Parallel Search Algorithm for Discrete Optimization Problems

CS 5204 Final Project Report
Kuan Yao
229-81-3553
kuyao@vt.edu

In this project, I implemented one parallel search algorithm (Depth First Search algorithm) to solve a typical discrete optimization problem, an 8-puzzle problem. The main purpose of the project is to demonstrate the speedup gained by the parallel computing to solve large search problem. The project implemented the dynamic load balance, task migration and termination detect algorithms. The program was running on IBM SP2 (Scalable Parallel Processor). The underlying communication layer was using MPI (Message Passing Interface).

Discrete Optimization Problem

A discrete optimization problem can be expressed as a tuple (S, f). The set S is a finite or countable infinite set of all solutions that satisfy specified constraints. This set is called the set of feasible solutions. The function f is the cost function that maps each element in set S onto the set of real numbers R.

$$f : S \rightarrow R$$

The objective of a DOP is to find a feasible solution x_{opt}, such that $f(x_{opt}) < f(x)$ for all $x \in S$. In most problems of practical interest, the solution set S is quite large. Consequently, it is not feasible to exhaustively enumerate he elements in S to determine the optimal element x_{opt}. Instead, a DOP can be reformulated as the problem of finding a minimum-cost path in a graph from a designated initial node to one of several possible goal nodes.

8-Puzzle Problem

The 8-puzzle problem consists of 3*3 grid containing eight tiles, numbered one through eight. One of the grid segments (called the "blank") is empty. A tile can be moved into the blank position from a position adjacent to it, thus creating a blank in the tiles' original position. Depending on the configuration of the grid, up to four moves are possible: up, down, left, and right. The initial and final configurations of the tiles are specified. The objective is to determine a shortest sequence of moves that transforms the initial configurations to the final configuration.

The 8-puzzle can be naturally formulated as a graph search problem.

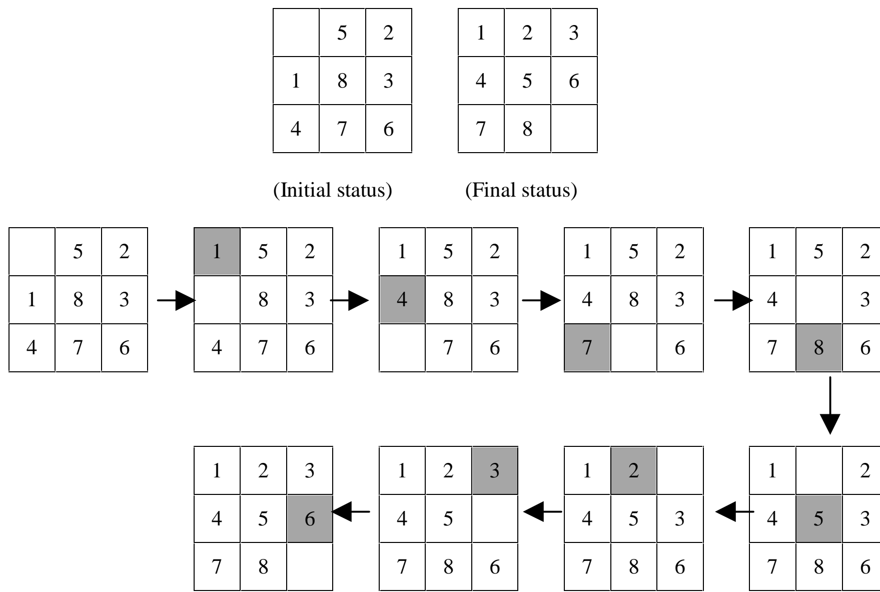

Figure 1. An 8-puzzle problem instance: initial configuration; final configuration; and a sequence of moves leading from the initial to the final configuration.

A common sequential search algorithm used for solving this problem is Depth-First Search algorithm. DFS begins by expanding the initial node and generating its successors. In each subsequent step, DFS expands one of the most recently generated nodes. If this node has no successors (or cannot lead to any solutions), DFS backtracks and expands a different node.

The figure-2 shows the execution of Depth-First Search for solving the 8-puzzle problem. The search starts at the initial configuration. Successors of this state are generated by applying possible moves. During each step of the searching, a new state is selected, and its successors are generated. The DFS algorithm expands the deepest node in the tree. In step 1, the initial state A generates states B and C. In step 2, the DFS algorithm selects state B and generates states D, E, and F. Note that the state D can be discarded, as it is a duplicate of the parent of B. In step 3, state E is expanded to generate states G and H. Again G can be discarded because it is a duplicate of B. The search proceeds in this way until the algorithm backtracks or the final configuration is generated.

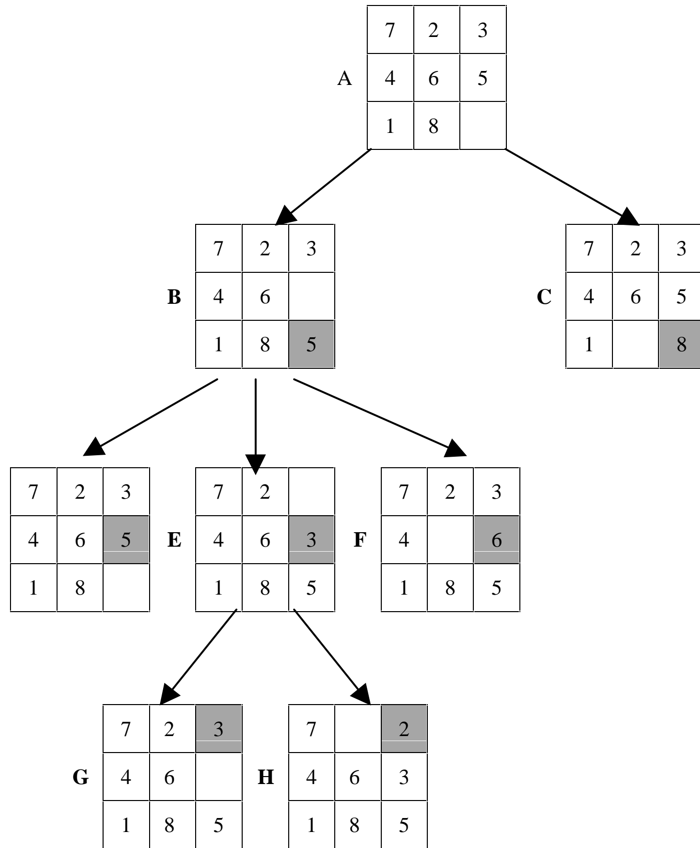

Figure 2. States resulting from the first three steps of depth-first search.

In each step of the searching, up to three untried alternatives are stored since these are untried alternatives. Let *m* be the amount of storage required to store a state, and *d* be the maximum depth, *l* be the maximum number of successors at each step, then the total space requirement of the DFS algorithm is $\Theta(mdl)$. A major advantage of DFS is that its storage requirement is linear in the depth of the state space being searched.

Parallel Depth-First Search

The critical issue in parallel depth-first search algorithm is the distribution of the search space among the processors. Consider the tree above. Note that the left subtree (rooted at node A) can be searched in parallel with the right subtree (rooted at node B). By statically assigning a node in the tree to a processors. It is possible to expand the whole subtree rooted at the node without communicating with another processor. The static allocation yields a good parallel search algorithm. However, like the digram above, the processor exploring the subtree rooted at node C expands considerable fewer nodes than does the other processor. Due to this imbalance in the workload, one processors is idle for a significant amount of time, reducing efficiency. Using more processors worsens the imbalance.

In dynamic loading balancing, when a processor runs out of work, it gets more work from another processor that has work. Although the dynamic distribution of work results in communication overhead for work requests and work transfers, it reduces load imbalance among processors.

A parallel formulation of DFS based on dynamic load balancing is as follows. Each processor performs DFS on a disjoint part of the search space. After a processor finishes searching its part of the search space, it requests an unsearched part from other processors. Whenever any processor finds a goal node, all the processors terminate. If the search space is finite and has no solutions, then all the processors eventually run out of work, and the algorithm terminates. Since each processor searches the state space depth-first, unexplored states can be conveniently stored as a stack. Each processor maintains its own local stack on which it executes DFS. When a processor's local stack is empty, it requests untried alternatives from another processor's stack. In the beginning, the entire search space is assigned to one processor, and other processors are assigned null search spaces (that is, empty stacks). The search space is distributed among the processors as they request work.

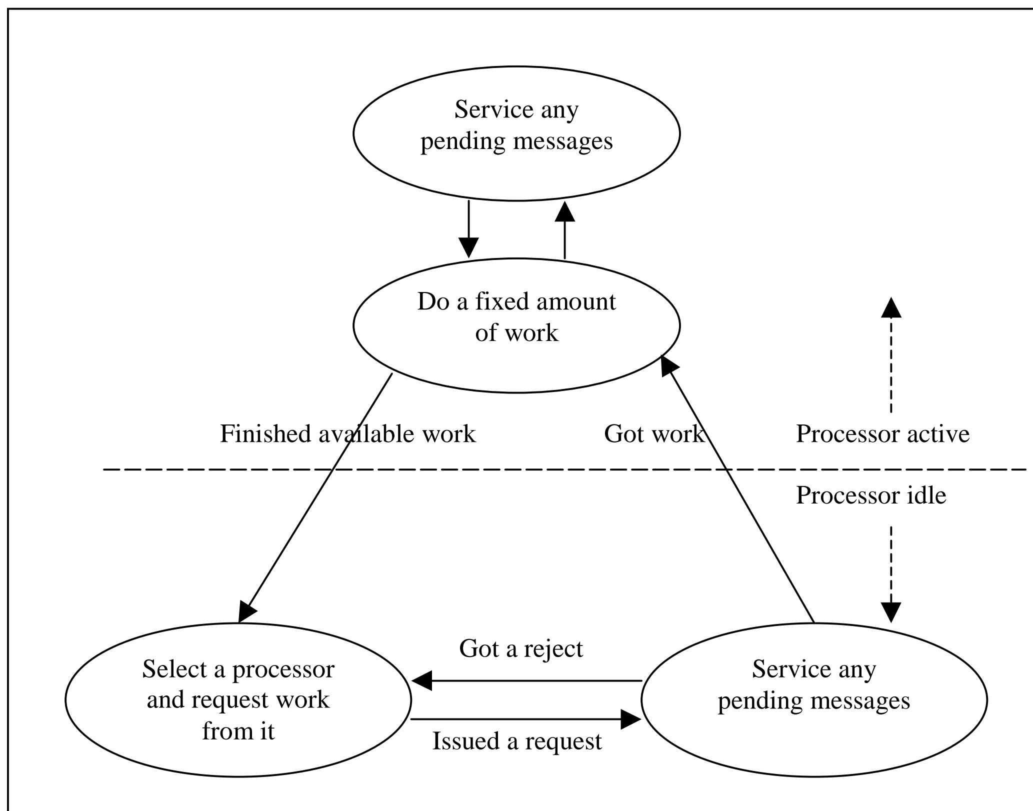

Figure 3. A generic scheme for dynamic load balancing.

As illustrated in the figure above, each processor can be in one of two states: active (that is, it has work) or idle (it is trying to get work). If the idle processor receives work from

other processors, it becomes active. If it receives a reject message (because the other processor doesn't have any work), it selects another processor and sends a work request to that one. This process repeats until the processor gets work or all the processors become idle.

In the active state, a processor does a fixed amount of work (expands a fixed number of nodes) and then checks for pending work requests. When a work request is received, the processor partitions its work into two parts and sends one part to he requesting processor. When a processor has exhausted its own search space, it becomes idle. This process continues until a solution is found or until the entire space has been searched. If a solution is found, a message is broadcast to all processors to stop searching. A termination detection algorithm is used to detect whether all processors have become idle without finding a solution.

When the work is transferred, the sender need to determine how much work it should migrate. In this project, the Donor simply sends the job at the bottom of the stack when it migrating job. However, this may results a quickly new imbalance since if too little work is sent, the recipient quickly becomes idle; if too much, the sender becomes idle. Ideally, the stack is split into two equal pieces such that the size of the search space represented by each stack is the same.

Load-balancing schemes used in this project is **Asynchronous Round Robin**. That is, each processor maintains an independent variable, *round_robin*. Whenever a processor runs out of work, it uses this variable as the label of a donor processor and sends it a work request. The value of the *round_robin* incremented (*modulo p*) each time a work request is sent. The initial value of target at each processor is set to processor 0. Since the work requests are generated independently by each processor, it is possible for two or more processors to request work from the same donor at nearly the same time.

When all the processors are run out of work, they should detect this situation and terminate after that. **Dijkstra's Token Termination Detection** Algorithm was used in this project. Visualize the p processors as being connected in a ring. Processor P_0 initiates a token when it becomes idle. This token is sent to the next processor in the ring, P_1. At any state in the computation, if a processor receives a token, the token is held at the processor until the computation assigned to the processor is complete. On completion, the token is passed to the next processor in the ring. If the processor was already idle, the token is passed to the next processor. Note that if at any time the token is passed to processor P_i, then all processors $P_0 \ldots P_{i-1}$ have completed their computation. Processor P_{p-1} passes its token to processor P_0; when it receives the token, processor P0 knows that all processors have completed their computation and the algorithm can terminate.

However, this simple scheme should be modified because after a processor goes idle, it may receive more work from other processors. In the modified scheme, the processors are also organized into a ring. A processor can be in one of two states: *black* or *white*. Initially, all processors are in state *white*. As before, the token travels in the sequence P_0, $P_1 \ldots P_{p-1}$, P_0. If the only work transfers allowed in the system are from processor P_i to P_j

such that $i < j$, then the simple termination scheme is still adequate. However, if processor P_j sends work to processor P_i, the token must traverse the ring again. In this case processor P_j is marked *black* since it causes the token to go around the ring again. Processor P_0 must be able to tell by looking at the token it receives whether it should be propagated around the ring again. Therefore the token itself is of two types. A *white* (or *valid*) token, which when received by processor P_0 implies termination; and a *black* (or *invalid*) token, which implies that the token must traverse the ring again. The detail procedures are show below:

1. When it becomes idle, processor P_0 initiates termination detection by making itself white and sending a white token to processor P_1.
2. If processor P_i sends work to processor P_j and i > j then processor P_i becomes black.
3. If processor P_i ahs the token and P_i is idle, then it passes the token to P_{i+1}. If P_i is black, then the color of the token is set to black before it is sent to P_{i+1}. If P_i is white, the token is passed unchanged.
4. After Pi passes the token to P_{i+1}, P_i becomes white.
5. The algorithm terminates when processor P_0 receives a white token.

Testing the Program.

The program was running on IBM SP2 machine. The Scalable Parallel Processor (SP2), is actually many powerful processors in one, all interconnected by high speed data links. Each of the fourteen processors in the Virginia Tech SP2 is equivalent to a processor on the 3090 mainframe, home of VTVM1. The communication layer API used is MPI (Message Passing Interface)

To make the parallel execution more apparently, I inserted some delay in each search step. The time was measured from P_0 first being initialized a job to either P_0 has found a path or P_0 got a PATH_FOUND message or a *white* token.

Number of Processors	Execution Time	Speedup
1	6.578092	**-**
2	1.892748	**3.48**
4	1.120812	**5.87**
6	0.655067	**10.04**
8	0.888851	**7.40**

I attached the data I collected. I got pretty good speed up when using 2, 4, 6, 8 processors. Due to the overhead, the minimum time is when using 6 processors. From the log file, you may see the jobs are migrated in random way.

The superlinear speedup has been achieved when the processor number are 2, 4, 6. This is due to the sequential search algorithm may not be the best or some other factors.

For this problem, if the maximum depth was pre-specified as 7, there may not be a path exists. The last log segment shows this case: all processors run out of work and are passing the token that doing the termination algorithms and terminate at last.

Summary

This project is trying to show the ability of parallel computing. It has some place could be improved. For example, because there are up to three moves in each search step, we can predict the next move, which is much close to our goal. An Admissible Heuristic Function could be used to get the weight of each move by calculating the Manhattan distance between the destination node and the next node. Also, in dynamic loading scheme, using Random Polling may get high efficiency over Round Robin.

I satisfied with the job migration, termination detected algorithm works well. And be happy to see the speedup.

Reference:

1. Introduction to Parallel Computing: Design and Analysis of Algorithms, Vipin Kumar, Ananth Grama, Anshul Gupta George Karypis

Appendix: data collection

```
**********************************
using one processor
**********************************
$ mpirun -np 1 demo
p0 found the path
A path found in : START-->DOWN-->DOWN-->RIGHT-->UP-->UP-->RIGHT-->DOWN-->DOWN
Total time is 6.578092 seconds
p0: total 254 steps searched


**********************************
use two processors
**********************************
$ mpirun -np 2 demo
migrate a job from processor 0 to processor 1
p1 found the path
processor 1 report that the path has been found
p1: total 136 steps searched
A path found in : START-->DOWN-->DOWN-->RIGHT-->UP-->UP-->RIGHT-->DOWN-->DOWN
Total time is 1.892748 seconds


**********************************
use four processors
**********************************
bash$ mpirun -np 4 demo
migrate a job from processor 0 to processor 3
migrate a job from processor 2 to processor 1
migrate a job from processor 0 to processor 2
migrate a job from processor 0 to processor 1
migrate a job from processor 1 to processor 0
migrate a job from processor 3 to processor 1
migrate a job from processor 3 to processor 0
migrate a job from processor 0 to processor 3
migrate a job from processor 0 to processor 2
migrate a job from processor 1 to processor 2
p2 found the path
p3: total 79 steps searched
p1: total 73 steps searched
p2: total 75 steps searched
processor 2 report that the path has been found
A path found in : START-->DOWN-->DOWN-->RIGHT-->UP-->UP-->RIGHT-->DOWN-->DOWN
Total time is 1.120812 seconds
p0: total 48 steps searched


**********************************
use eight processors
**********************************
bash$ mpirun -np 8 demo
migrate a job from processor 0 to processor 7
migrate a job from processor 1 to processor 5
migrate a job from processor 0 to processor 1
migrate a job from processor 0 to processor 3
migrate a job from processor 0 to processor 4
migrate a job from processor 0 to processor 6
migrate a job from processor 0 to processor 2
migrate a job from processor 1 to processor 0
migrate a job from processor 2 to processor 6
migrate a job from processor 1 to processor 0
migrate a job from processor 0 to processor 2
migrate a job from processor 0 to processor 4
migrate a job from processor 0 to processor 6
migrate a job from processor 0 to processor 1
migrate a job from processor 0 to processor 3
migrate a job from processor 5 to processor 0
migrate a job from processor 7 to processor 6
migrate a job from processor 6 to processor 5
```

```
migrate a job from processor 7 to processor 0
migrate a job from processor 1 to processor 6
migrate a job from processor 5 to processor 1
migrate a job from processor 7 to processor 1
migrate a job from processor 0 to processor 7
migrate a job from processor 0 to processor 6
migrate a job from processor 6 to processor 1
migrate a job from processor 5 to processor 0
p1: total 46 steps searched
p2: total 17 steps searched
p3: total 17 steps searched
p4: total 19 steps searched
p7: total 60 steps searched
p6: total 37 steps searched
p5 found the path
p5: total 49 steps searched
migrate a job from processor 0 to processor 1
processor 5 report that the path has been found
A path found in : START-->DOWN-->DOWN-->RIGHT-->UP-->UP-->RIGHT-->DOWN-->DOWN
Total time is 0.888851 seconds
p0: total 39 steps searched


**********************************
use six processors, due to the
overhead, using six processors can get the
fastest execution time.
**********************************
bash$ mpirun -np 6 demo
migrate a job from processor 0 to processor 1
migrate a job from processor 1 to processor 5
migrate a job from processor 0 to processor 3
migrate a job from processor 0 to processor 4
migrate a job from processor 0 to processor 2
migrate a job from processor 1 to processor 2
migrate a job from processor 1 to processor 4
migrate a job from processor 1 to processor 0
migrate a job from processor 3 to processor 1
migrate a job from processor 2 to processor 0
migrate a job from processor 1 to processor 0
p2: total 38 steps searched
p5 found the path
p3: total 44 steps searched
p4: total 42 steps searched
processor 5 report that the path has been found
p5: total 41 steps searched
A path found in : START-->DOWN-->DOWN-->RIGHT-->UP-->UP-->RIGHT-->DOWN-->DOWN
p1: total 44 steps searched
Total time is 0.655067 seconds
p0: total 32 steps searched


**********************************
if the max search step is 7, it may
not find the path, I expect it use
termination detection algorithms to
stop the execution. use four processors
see the 'passing token' and 'path not
found' message.
**********************************
bash$ mpirun -np 4 demo
migrate a job from processor 0 to processor 3
migrate a job from processor 1 to processor 2
migrate a job from processor 0 to processor 1
migrate a job from processor 1 to processor 0
migrate a job from processor 0 to processor 2
migrate a job from processor 0 to processor 2
migrate a job from processor 3 to processor 0
migrate a job from processor 1 to processor 2
migrate a job from processor 2 to processor 1
migrate a job from processor 0 to processor 3
```

```
migrate a job from processor 2 to processor 0
migrate a job from processor 3 to processor 1
migrate a job from processor 0 to processor 2
migrate a job from processor 3 to processor 2
passing token from 1 to 2
migrate a job from processor 3 to processor 0
passing token from 0 to 1
passing token from 1 to 2
passing token from 2 to 3
passing token from 3 to 0
passing token from 0 to 1
p1: total 38 steps searched
passing token from 2 to 3
passing token from 3 to 0
cann't find a path in 7 steps
p2: total 34 steps searched
p3: total 48 steps searched
Total time is 0.725413 seconds
p0: total 39 steps searched
```