Parallel State Space Searching Algorithms

Hart Lambur <hal2001@columbia.edu> and Blake Shaw <bs2018@columbia.edu>
Professor Keyes, APAM 4990 Class Project Write-up

May 13, 2004

Introduction

As a pedagogical exercise in parallel programming and scientific computing, we looked at the problem of state space searching. We focused on the standard example problem, the 15-puzzle, and developed original C++ code which quickly and efficiently finds solutions to the puzzle on a parallel machine, using two different algorithms. In this paper, we summarize the theory behind our two parallel implementations, and analyze our results using the Jumpshot profiling package. Furthermore, we discuss the problems with our scalability and performance data, and offer some conclusions on the success of this project, as well as some ideas about directions for future research.

Discrete Optimization Problems

The state space searching problem can be formalized as a discrete optimization problem (DOP). Given a tuple (S, f) where S is a finite or countably infinite set of solutions, and f is a cost function $f : S- > R$, the objective of a DOP problem is to find a feasible solution x_{opt} such that:

$$f(x_{opt}) \leq f(x), \qquad \forall\, x \,\epsilon\, S$$

The 15-Puzzle

As an example problem, we chose to focus on the 15-puzzle, a simple sliding tile puzzle on a 4 by 4 grid. The 15-puzzle has long been popular among mathematicians and AI researchers for its conceptual simplicity and large associated search space. An estimated 10 trillion unique states ($4^2!/2$) of the puzzle are possible, and finding an optimal (shortest) path to a given state is known to be an NP-complete problem. Describing the 15-puzzle in terms of the formal DOP definition, the S space is the set of possible moves from an initial to final configuration of the puzzle, and the cost function f is the number of moves in the sequence.

Finding a solution to the 15-puzzle involves searching the state space, which is best conceptualized as searching a tree of possible puzzle configurations. Three standard search algorithms exist for this purpose. Breadth-first-search (BFS) searches every node at a certain depth before searching the next depth. Depth-first-search (DFS) searches the tree as deeply as possible before backtracking and searching another path, essentially searching the tree from left to right. Best-first-serach (A*) assigns a heuristic value to each state and searches the nodes of the tree guided by that heuristic. In this project, we look at how to best parallelize DFS and A*, widely accepted as the two most useful of these searching techniques.

MPI Implementation

To parallelize DFS and A*, we wrote object-oriented code from scratch in C++ using MPI. The code was developed and tested on Columbia's AFQ cluster, a 64-node Beowulf machine. The details of our parallel implementation follow.

Parallel DFS

In a parallel implementation of DFS, our goal is to have each processor searching a unique part of the state space. Essentially, we want to divide the search space into N separate trees, where N is the number of processors. To accomplish this, the head processor distributes nodes to the other processors and every processor begins a serial depth-first search on its assigned section of the state space. An open list of unvisited nodes and a closed list of visited nodes are maintained locally by each processor. Because of the effect move ordering has on the 15-puzzle, maintaining a closed list of visited nodes eliminates redundant searching of previously visited paths through the tree. When a processor finds the desired goal node, it uses a non-blocking send to tell the other processors, and the search is halted.

Parallel A*

Like the DFS algorithm, parallel A* maintains open and closed lists of nodes. A*, however, sorts its open list by a heuristic value, and searches those nodes in order of how "good" they are, based on this heuristic. In a parallel implementation of A*, we want to keep each processor searching the "best" part of the tree. The general method used to accomplish this involves some sort of synchronization strategy, where each processor passes some of its best nodes to other processors. In our implementation, we adopted a simple ring synchronization strategy. Like our parallel DFS search, a head processor distributes nodes to every other processor and the search is started. However, at a specified rate each processor passes its "best" node along a virtual ring to its neighbor, thereby insuring every processor will get to search some of the "best" part of the tree. As in the DFS search before, when the desired goal node is found, a non-blocking send is used to tell all the other processors, and the search is halted.

Profiling and Performance

Using the Jumpshot profiling package developed at Argonne National Laboratory, we were able to construct some informative images of our message passing code. Unfortunately, due to the nonhomogeneous nature of our cluster and the difficulty of securing dedicated CPU time, accurate analysis of our performance and scaling results was nearly impossible. The details of our findings follow.

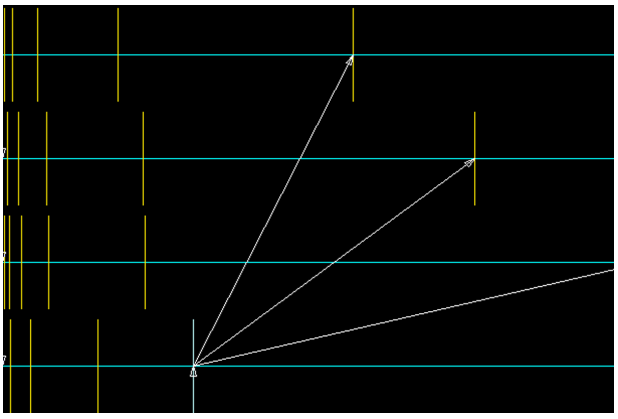

Figure 1: A Jumpshot profile of a randomly generated depth 12 puzzle solved using ID-DFS. The yellow lines marking the start of each depth bounded search clearly show how the time to search increases exponentially with the tree depth. The solution that processor four finds at ∼20 sec is communicated using a non-blocking send (the white arrows) to the other processors.

Jumpshot Profiling

The Jumpshot package was extremely useful in illustrating the inner workings of our code. Figure 1 shows a four-processor ID-DFS search, where we can see clearly how the time of each search increases exponentially as the depth bound is increased. The yellow lines represent `MPI_Test` calls executed at every increase in the depth bound of each processor's depth-first search. One can see clearly how the tree for small depth bounds (like 3 and 5) can be searched quite quickly, but as the depth bound increases (to 7, 9, 11 and finally 13) the time for each search increases exponentially. When the fourth processor finds a solution at ∼20 sec, it uses a non-blocking send to tell the other processors it has found a solution; since the other processors check only the

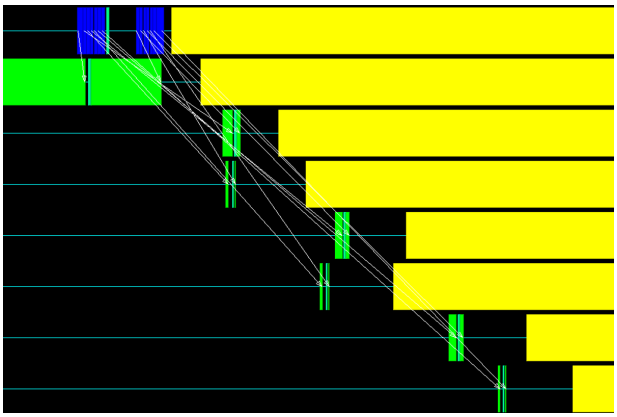

Figure 2: Close-up view of the initial distribution of nodes in an eight-processor A* search of a depth 30 puzzle. We can see how the head processor (at the top of the graph) initially sends a copy of the desired goal node to each processor (represented by the first group of blue lines) before distributing unique start nodes to the processors (the second group of blue lines). After receiving this information from the head processor, all processors enter an MPI_Barrier call to try to synchronize the processors before the parallel search begins.

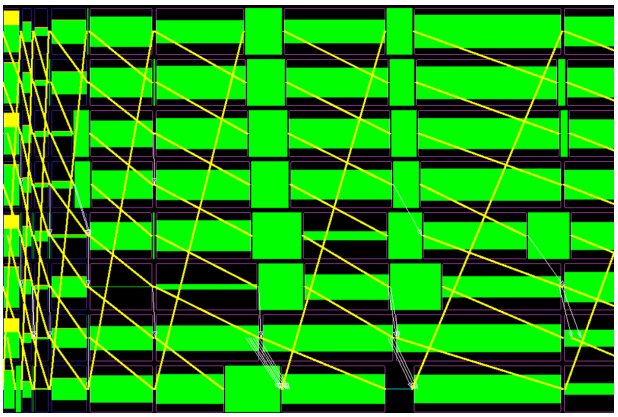

Figure 3: View of an eight-processor parallel A* search "in-progress." The orange lines represent MPI_Test commands used to check if a solution has been found by a different processor. The green blocks represent MPI_Recv commands used in the ring-passing communication strategy. We can see from the cyclic structure of the white "message-passing" arrows how nodes are passed from one processor to the next along this ring.

status of that send at the end of each depth-bounded search, the program does not terminate until each processor has completed its current depth-bounded search. In Figure 1, processor three takes a long time completing its depth 13 search (∼65 sec), preventing the program from terminating quickly; one obvious improvement would be to implement some sort of kill command to eliminate such inefficiencies. Unfortunately, an `MPI_Kill` command does not exist, but we hope to implement a similar feature in the future.

Figures 2 and 3 illustrate a typical eight-processor parallel A* search. Figure 2 shows how the head processor initially distributes the goal node and unique start nodes to each of the child processors. After receiving all the necessary information, an `MPI_Barrier` command is called to help synchronize the processors before the search is continued; this helps maintain a unified start time across the system. Figure 3 shows the search "in progress." The ring communication strategy is clearly visible: at a specified sync rate, nodes are passed to a neighboring processor along a ring, as demonstrated by the cyclic structure of the white "message-passing" arrow.

Performance and Scaling

The table below shows the relative time needed to find a solution to a given depth 10 puzzle for both the DFS and A* strategies, in both serial and four processor runs.

DFS NP 1	DFS NP 4	A* NP 1	A* NP 4
277.527	87.15	0.000575	0.005918

We can see that the four processor parallel DFS search offers a roughly 3.2x speedup over its serial counterpart; both, however, take far longer than an A* search. The intelligence of the A* algorithm is readily apparent and demonstrates how efficient algorithms are far more important than raw computation power in a parallel cluster or on a serial machine. For this small problem, the serial and parallel runs of A* finish in a fraction of a second. In

such a quick run, the communication overhead of the parallel A* run dominates the search time. To test the scalability of the A* algorithm, we used a far more challenging depth 30 puzzle, as demonstrated in the next table.

A* NP 1	A* NP 4	A* NP 8	A* NP 12
26.8331	3.14834	4.9506	0.330026

The above data shows obvious inconsistencies—no linear scaling factor exists for these times. These results demonstrate our frustration in securing undisturbed CPU time on the AFQ cluster; without submitting our jobs to the LSF queue (which was offline the week prior to our presentation), running parallel jobs using `mpirun` would produce inconsistent results. Given the nature of these results, we were discouraged from attempting any sort of extensive data collection such as averaging the time to completion for a varying number of processors over a large number of unique puzzles. This kind of extensive data collection would be needed before the scalability of our parallel A* algorithm could be determined; however, we can see that the parallel implementation offers significant performance gains over the serial version.

Conclusions

The experience of programming from scratch proved incredibly fruitful. We gained great insight into the inner workings of MPI, C++ and computational clusters as well as expanding our understanding of the state space searching problem. Furthermore, we learned firsthand just how difficult programming parallel computers can be, and how MPI-based algorithms are much harder to debug and understand than their serial counterparts.

While successful in what we accomplished, our project still leaves a great deal of room for improvement. A first priority would be developing an accurate testing and analysis suite to help us determine the scalability of our algorithms. This effort would likely involve an increased use of the LSF queue and its commands, something we have begun to understand but could better utilize. With an accurate testing suite, we could begin to optimize the performance of our code as well as develop and implement (and quite possibly invent) new synchronization strategies to help speed our A* searches.

References

[1] V. Cung and B. Cun. An efficient implementation of parallel a*, 1994.

[2] A. Grama, V. Kumar, and P. Pardalos. Parallel processing of discrete optimization problems. In *Encyclopaedia of Microcomputers*. Marcel Dekker Inc., New York, 1992.

[3] A. Y. Grama and V. Kumar. A survey of parallel search algorithms for discrete optimization problems, 1993.

[4] William Gropp, Ewing Lusk, and Anthony Skjellum. *Using MPI: Portable Parallel Programming with the Message Passing Interface,* 2nd edition. MIT Press, Cambridge, MA, 1999.

[5] V. Kumar, A. Grama, A. Gupta, and G. Karypis. *Introduction to Parallel Computing: Design and Analysis of Algorithms,* 2nd edition. Addison-Wesley, 2003.