High Performance Computing : Assignment 3

Parallel algorithm for Binary Search and Breadth First Search.

Oral questions

1. Is it possible to parallelize the linear search ?

   Yes start different process from different ends

2. What is the Parallel formulation of Depth First Search?

3. work request and response messages in message passing architectures, and locking and extracting work in shared address space machines. Whenever a processor finds a goal node, all the processors terminate. If the search space is finite and the problem 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 (either via explicit messages or by locking) 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. We refer to the processor that sends work as the ***donor*** processor and to the processor that requests and receives work as the ***recipient*** processor.

4. Which decomposition method is used in parallelizing the above searching methods?

   Divide & Conquer

5. Explain parallel formulation of Best First Search.

important component of best-first search (BFS) algorithms is the *open* list. It maintains the unexpanded nodes in the search graph, ordered according to their *l*-value. In the sequential algorithm, the most promising node from the *open* list is removed and expanded, and newly generated nodes are added to the *open* list.

In most parallel formulations of BFS, different processors concurrently expand different nodes from the *open* list. These formulations differ according to the data structures they use to implement the *open* list. Given *p* processors, the simplest strategy assigns each processor to work on one of the current best nodes on the *open* list. This is called the ***centralized strategy*** because each processor gets

work from a single global *open* list. Since this formulation of parallel BFS expands more than one node at a time, it may expand nodes that would not be expanded by a sequential algorithm. Consider the case in which the first node on the *open* list is a solution. The parallel formulation still expands the first p nodes on the *open* list. However, since it always picks the best p nodes, the amount of extra work is limited. There are two problems with this approach:

The termination criterion of sequential BFS fails for parallel BFS. Since at any moment, p nodes from the *open* list are being expanded, it is possible that one of the nodes may be a solution that does not correspond to the best goal node (or the path found is not the shortest path). This is because the remaining p - 1 nodes may lead to search spaces containing better goal nodes. Therefore, if the cost of a solution found by a processor is c, then this solution is not guaranteed to correspond to the best goal node until the cost of nodes being searched at other processors is known to be at least c. The termination criterion must be modified to ensure that termination occurs only after the best solution has been found.

Since the *open* list is accessed for each node expansion, it must be easily accessible to all processors, which can severely limit performance. Even on shared-address-space architectures, contention for the *open* list limits speedup. Let t_{exp} be the average time to expand a single node, and t_{access} be the average time to access the *open* list for a single-node expansion. If there are n nodes to be expanded by both the sequential and parallel formulations (assuming that they do an equal amount of work), then the sequential run time is given by $n(t_{access} + t_{exp})$. Assume that it is impossible to parallelize the expansion of individual nodes. Then the parallel run time will be at least nt_{access}, because the *open* list must be accessed at least once for each node expanded. Hence, an upper bound on the speedup is $(t_{access} + t_{exp})/t_{access}$.

6. State parallel run time of above mentioned searching methods
7. Compare sequential and parallel search.
8. How to parallelise Fibonacci search?
9. Which algorithmic strategy is used in BFS, DFS?
10. What are the application of Best First Search?
11. Is it possible to implement above searching methods in cluster environment?

Practice problem

1. Implement parallel searching using cluster of Raspberry Pi

2. Check the performance of this program by varying number of nodes in a cluster and plot the graph.
3. Implement parallel Depth First Search.
4. Implement Parallel Best First Search
5. Implement Fibonacci Search
6. Implement radix sort.
7. Implement any searching algorithm
8. Take a input from file
9. Use Random function
10. Provide a input at run time

Similarly print output in a file