Branch and Bound

Template for Parallel Programming on Clusters

Design Document

Project – Parallel Compting (CS F422)

Rajath Shashidhara	2012B5A7589P	rajath.shashidhara@gmail.com
Ajinkya Kokandakar	2012B3A7463P	ajinkya.kokandakar@gmail.com
Varad Gautam	2012A8TS237P	varadgautam@gmail.com

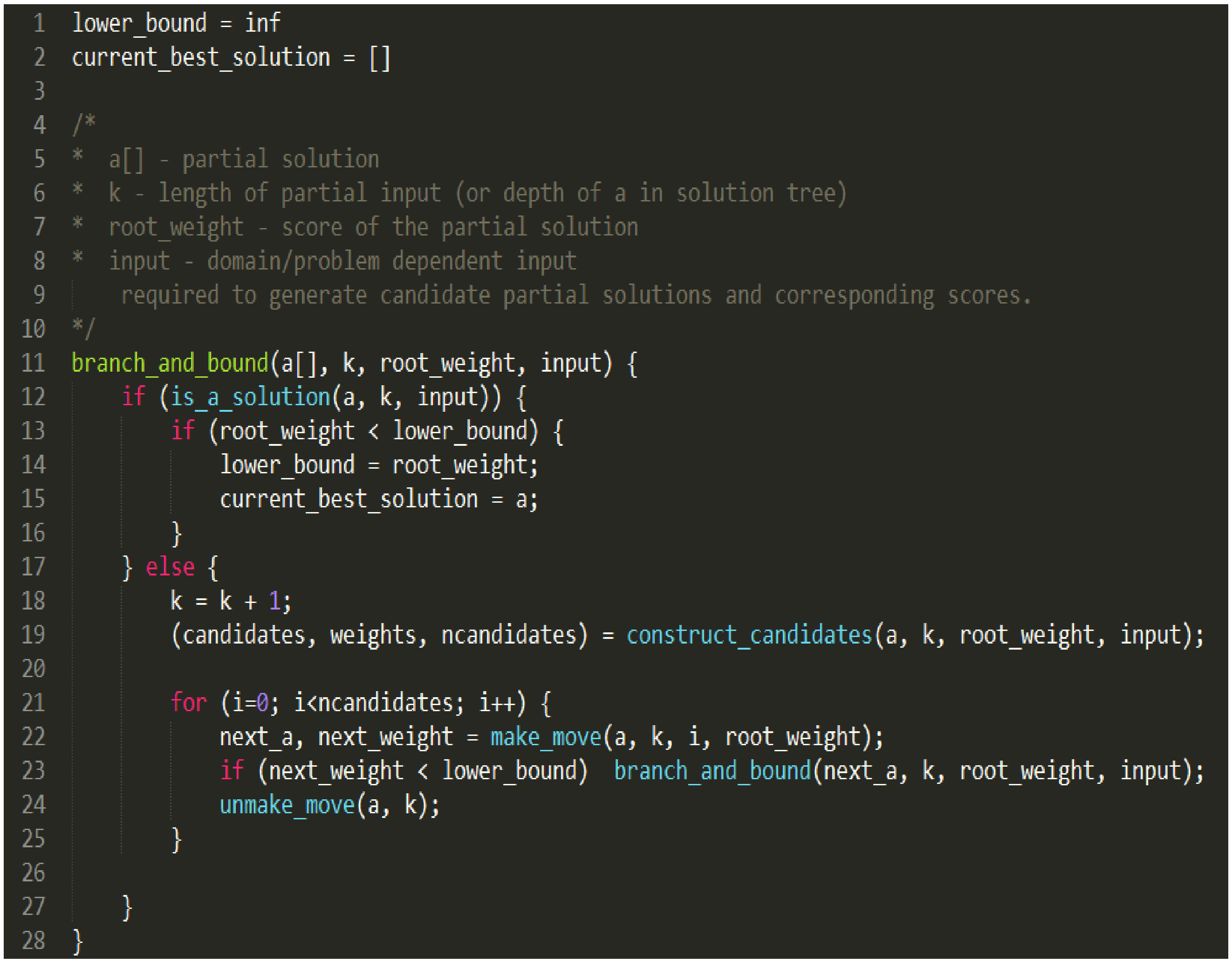
Problem Description

Branch and bound is an algorithm/design paradigm to solve discrete combinatorial optimization problem. Branch and Bound involves searching the solution space by systematic enumeration of possible solutions.

Like backtracking, the sequential solution for branch and bound involves recursively constructing and exploring prospective solutions to find the optimal one. However, instead of brute-forcing to check the optimality of a (partial) solution, the branch and bound algorithm relies on a cost function that is evaluated for each candidate solution and compared against the current best solution, the "bound". This allows eliminating suboptimal (partial) candidates from further processing and saves on execution time.

The sequential implementation for Branch and Bound is a Depth First Search of the solution space. Branches (corresponds to partial solutions) having a score, which is worse than the current best lower bound are pruned during the search.

Pseudo-code for template (sequential) branch and bound :



```
lower_bound = inf
current_best_solution = []

/*
*  a[] - partial solution
*  k - length of partial input (or depth of a in solution tree)
*  root_weight - score of the partial solution
*  input - domain/problem dependent input
    required to generate candidate partial solutions and corresponding scores.
*/
branch_and_bound(a[], k, root_weight, input) {
    if (is_a_solution(a, k, input)) {
        if (root_weight < lower_bound) {
            lower_bound = root_weight;
            current_best_solution = a;
        }
    } else {
        k = k + 1;
        (candidates, weights, ncandidates) = construct_candidates(a, k, root_weight, input);

        for (i=0; i<ncandidates; i++) {
            next_a, next_weight = make_move(a, k, i, root_weight);
            if (next_weight < lower_bound)  branch_and_bound(next_a, k, root_weight, input);
            unmake_move(a, k);
        }

    }
}
```

We intend to solve the following problems using our template (parallel) algorithm

1) Graph Vertex Coloring
2) Travelling Salesman Problem

Both the above problems are intractable (they do not have a known polynomial time algorithm) and the best known sequential algorithm (based on branch and bound) has exponential time complexity.

Parallelization

The proposed solution involves the parallelization of the branch and bound algorithm for a cluster of multicore computers. The proposed solution makes use of both distributed parallelism and shared memory parallelism.

In the above pseudo-code, note that the method construct_candidates() is problem dependent, therefore not amenable for parallelization in a template algorithm. It is expected that a problem specific implementation of construct_candidates() is supplied, and no attempt to parallelize construct_candidates() is made.

In our parallel algorithm, different branches of the solution tree are explored simultaneously. This may lead to some unproductive exploration of the search space, as opposed to the sequential algorithm, since we have deviated from a pure DFS based approach.

Decomposition

The primitive task in this algorithm is to construct possible candidates for extending a partial solution until no extension is possible. If no extension is possible it must be evaluated to determine if the obtained solution is feasible and update the bound in case the solution is better than the previous bound.

Major Design Concerns

When attempting to design a distributed algorithm for branch and bound, the following issues must be addressed :

1) Initialization – partitioning task/data across nodes at the start.
2) Bound Sharing – the bound of the current best solution has to shared with other processes for pruning.
3) Load Balancing – dynamically redistribute the skewed load amongst processes. This has to be done both at the level of processes (distributed

paradigm) and threads (shared memory paradigm).

4) Termination Detection – co-operative processes must determine if the search space is exhausted and quit.

In the proposed design, each node has one control and communication thread and c-1 worker threads where c is the number of cores per node. The control and communication thread manages issues requiring co-operation from other processes – initialization, bound sharing, load balancing, termintion detection, clean up and exit.

Worker Threads and Load Balancing

A shared queue (also called the work stealing queue) of primitive tasks is maintained at each node. This queue is used to schedule tasks to threads on a single node. Each idle thread obtains a task (partial solution) from the queue, and generates candidate [partial] solutions by extending the partial solution. The candidate [partial] solutions are merged into the queue for further processing.

The queue must satisfy the following requirements :

- Thread–safety
- Each task is associated with a priority (score assigned to each partial solution) – solution tree is expanded in the best first fashion for effective pruning.
- Efficient bulk insert – each thread produces a large number of new tasks (size comparable to the queue size) to be merged into the queue.
- Efficient bulk extract – large number of tasks (with varying priorities) must be extracted from the queue for load balancing with neighbours.
- Queue must not overflow – it is not possible to predict the number of extensions to a partial solution. Therefore, the queue must not be of fixed size.
- Efficient pruning of tasks – when a new lower bound is reached, tasks having a score greater than the bound must be pruned.

Interface –

1. partial_solution min_task(queue q);
2. void merge_queues(queue q1, queue q2);
3. void prune_tasks(queue q, float new_bound);

4. void extract_tasks(queue q, int num_tasks);
5. void insert_task(queue q, partial_solution sol, float score);

Regular array based heap implementation of priority queue does not meet the criteria listed above. Instead, we have chosen to implement a tree-based heap data structure called *“leftist heap”*. The above listed operations can be very efficiently implemented using the idea of leftist heaps.

As per “Data structures and their algorithms, Harry R. Lewis and Larry Denenberg”, the leftist heap is defined as :

Leftist heap is an unbalanced binary tree with properties :

for each node p:

1. key(p) ≥ key(parent(p))

   Partial order is maintained on the set of elements.

2. dist(right(p)) ≤ dist(left(p))

   [an external node has fewer than two children]

   dist(p) is the number of edges on the shortest path from node p to a descendent external node.

   The shortest path to a descendent external node is through the right child.

[For detailed explanation : refer to http://www.dgp.toronto.edu/people/JamesStewart/378notes/10leftist/]

Operation	**Time Complexity**
min_extract	O(log(N)) [worstcase]
insert	O(log(N)) [worstcase]
merge	O(log(N)) [worstcase]
prune_tasks	O(N) [worstcase] [average & amortized cost will be much lesser]
bulk_extract	O(log(N) – m) [m is the number of tasks to be extracted – this will produce a list of tasks with varying scores for effective load balancing – this will also help in balancing the tree].

We have decided to use coarse grained locking for two reasons -

1. Ease of design and implementation due to time constraints.
2. Several approaches to design fine grained concurrency in priority queues has failed to a large extent.

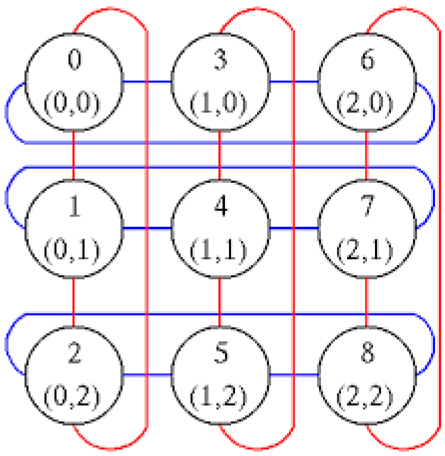
   [Refer : http://arxiv.org/pdf/1509.07053.pdf . We may look into skiplist based lock free queues if time permits.]

Initialization

Before the dynamic load balancing takes effect, in order to have balanced distribution of work, all the nodes must be initialized with roughly uniform number of partial solutions to start processing. This is achieved by hierarchical distribution of data to nodes.

This initialization phase uses tree like communication pattern using processor ranks. The root of the tree beings with the empty solution and generates candidate partial solution extensions. The partial solutions are then distributed to the children nodes. At every internal node if there is only one solution then it is expanded and the partial solutions are distributed among the children and the node. Thus the initial distribution take place logarithmically. (It is important to note that the internal node keeps some partial solutions its own queue). The leaf nodes do not do any expansion in the initialization phase. Tasks distributed to nodes are used to populate the shared priority queue to distribute tasks amongst the threads.

Communication Patterns and Virtual Topologies



All the communications in the algorithm are local. Cost of local communication is lesser than global communication as the cost of synchronization of processes for global communication. Also, the communication between processes is structured.

For the purpose of all the communications, a toroid virtual topology will be used. A toroid topology is a modified version of the mesh topology. Connecting the end nodes in a mesh topology with the other end in both axes leads to a toroid topology.

The distributed algorithms for bound sharing, load balancing and terminiation detection make use of this virtual toplogy.

Bound Sharing

A cylindrical communication pattern is used to communicate the bound. When a node obtains a feasible solution, the control and communication thread will initiate the communication of this bound.

Let node 4 in the figure be the node that initiates the bound communication. The node sends the bound to all its neighbours i.e. communicates it in all four directions. The nodes on the vertical containing the initiator i.e. node 3 and node 5, send this in three directions left, right and the direction opposite to the one from which the node received the bound. The nodes not on the vertical axis communicate the bound only in the direction opposite the one from which it received the bound. If a node receives a bound which it has already communicated to its neighbours, it is ignored. This has the effect of simulating a broadcast using a spanning tree rooted at the initiator.

The aforementioned cylindrical communication pattern can be easily achieved on the toroid virtual topology. The advantage of the toroid topology is that all nodes are symmetric in the toroid topology and any initiator can be visualized as being the centre of its axis. In other words, the toroid is isotropic and homogenous.

Load Balancing

The load balancing is handled by the control and communication thread. As with bounds communication, load balancing can be intuitively modelled by a global queue storing partial solutions. However, this is inefficient in a distributed computing environment. The proposed design is diffusive load balancing.

In Diffusive load balancing, candidate solutions in the priority queue can only be shared with its neighbours.

Diffusive load balancing is a continuous process. The control and communication thread triggers load balancing when any worker thread inserts partial solutions in the queue.

The load balancing protocol also the toroid topology. Thus every node can perform load balancing four neighbours.

Every node can perform load balancing only wit its four neighbours. The C&C thread sends a communication request to all its neighbours. The neighbours respond with ACCEPT/REJECT. If the node is ready to accept then it also sends local queue size. A node can respond with REJECT only if it is already in a round of load balancing with some other node. Initiation of load balancing requires at least 8 point to point communications. The number of partial solutions distributed to each neighbouring nodes is in inverse proportion to their queue lengths. The current score of the partial solutions are also considered when selecting exactly which partial solutions are to be distributed. The selected partial solutions are then removed from the queue and distributed to the neighbouring nodes.

[Reference for detailed explanation : Corradi, Antonio, Letizia Leonardi, and Franco Zambonelli. "Diffusive load-balancing policies for dynamic applications." *IEEE concurrency* 1 (1999): 22-31.]

Termination Detection

The parallel branch and bound must terminate when the queue in all nodes is empty. This is a non-trivial problem as the process must communicate amongst each other to determine if everybody has finished processing and this process is inconvenienced by the dynamic load balancing and scheduling algorithms.

The distributed termination detection algorithm is based on the Dijkstra's DTD algorithm. The communication pattern used is a ring (the virtual toroid topology can be reused to recreate the ring communication pattern).

Process 0 (a designated process) always initiates the termination detection algorithm when it is left with no tasks even after four rounds of load balancing (once with each of its neighbours). A message is passed around in a ring, requesting each node to stop load balancing. If a node still has tasks left, the message is rolledback along the ring to restart load balancing. If a node has no tasks left, it stops load balancing and forwards the message to the next neighbour on the ring. When the message returns back to process 0, a second message is sent. Each tasks adds its queue length to the message and forwards it to the next node in the ring. If the count is zero, when the message arrives at process 0, the program has to terminate and an intruction to quit is ciruculated along the ring.

[Reference : Dijkstra, Edsger W., Wim HJ Feijen, and A_J M. Van Gasteren. "Derivation of a termination detection algorithm for distributed computations." *Information processing letters* 16, no. 5 (1983): 217-219.]

References

Along with the references mentioned above, the one listed below were extensively used to arrive at the current design.

- Skiena, Steven S. *The algorithm design manual: Text*. Vol. 1. Springer Science & Business Media, 1998.

- Goodrich, Michael T., and Roberto Tamassia. *Algorithm design: foundation, analysis and internet examples*. John Wiley & Sons, 2006.
- Herlihy, Maurice, and Nir Shavit. *The Art of Multiprocessor Programming, Revised Reprint*. Elsevier, 2012.
- Quinn, Michael J. *Parallel computing: theory and practice*. McGraw-Hill, Inc., 1994.
- Grama, Ananth, Anshul Gupta, and George Karypis. *Introduction to parallel computing: design and analysis of algorithms*. Vol. 400. Redwood City, CA: Benjamin/Cummings, 1994.
- Willebeek-LeMair, Marc H., and Anthony P. Reeves. "Strategies for dynamic load balancing on highly parallel computers." *Parallel and Distributed Systems, IEEE Transactions on* 4, no. 9 (1993): 979-993.
- Finkel, Raphael, and Udi Manber. "DIB—a distributed implementation of backtracking." *ACM Transactions on Programming Languages and Systems (TOPLAS)* 9, no. 2 (1987): 235-256.