A Scalable Fully Distributed Parallel Branch & Bound Algorithm on nCUBE

Chee Kane Chang, Solomon K.-C. Wu, Ying Teh Tsai, and Chuan Yi Tang

Department of Computer Science
National Tsing Hua University
Hsinchu, Taiwan, 30043, Republic of China

Correspondence Address:
Prof. Chuan-Yi Tang
Department of Computer Science
National Tsing Hua University
Hsinchu, Taiwan, 30043, Republic of China
Phone: 886-35-731077
Fax: 886-35-723694
E-mail: cytang@cs.nthu.edu.tw

A Scalable Fully Distributed Parallel Branch & Bound Algorithm on nCUBE

Chee Kane Chang, Solomon Kuency Wu, Ying Teh Tsai, and Chuan Yi Tang

Department of Computer Science
National Tsing Hua University
Hsinchu, 30043
Taiwan, R.O.C.

Correspondence e-mail address: cytang@cs.nthu.edu.tw

Abstract

The branch & bound is an important design strategy of algorithm to solve NP-complete combinatorial optimization problems. In this paper, we present a best-first branch & bound scheme for parallel computers to solve the traveling salesperson problem. The scheme is based on a simple nearest-neighbor dynamic balancing strategy which reduces the redundant search overhead but doesn't involve too much communication overhead.

Load-imbalancement is usually the main reason that degrades the performance of a parallel program. In branch & bound algorithms, because it is impossible to predict the size of each branch beforehand, the workload has to be balanced again and again at run-time. In this paper, we propose a dynamic load-balancing algorithm, which is much simpler and easier to implement, in comparison with [17], but still equally efficient.

To show the performance of our scheme, we solve the traveling salesperson problems on an nCUBE/2^{TM}[1] hypercube computer. Using 128 processors, we are able to achieve a speedup 102.02 for fifty-city, and a speedup 119.44 for sixty-city problem sizes. These results verify that our best-first branch & bound scheme is efficient.

Keywords: Parallel computing, dynamic load-balancing, branch & bound, traveling salesman problem, scalability.

1 Introduction

The branch & bound strategy is widely used to design algorithms for solving NP-complete problems. A survey for this strategy can be found in [9]. The branch & bound strategy,

---

[1] nCUBE/2 is a trademark of the nCUBE Corporation

due to its characteristics, is naturally suitable for parallelization. Since a branch & bound strategy is a kind of tree searching technique, we can let several processors simultaneously search through the different parts of the search tree. Much research about parallel branch & bound has been done [1, 4, 7, 8, 10, 12, 15].

There are two implementation models for parallel branch & bound algorithms: the centralized model and the distributed model. In this paper, we propose a fully distributed balancing scheme for efficient parallelization of the best-first branch & bound algorithm. In this scheme, we try to let all the processors expand nodes which have the same or nearly the same lower bounds. During the expansion, some of the processors will generate nodes which are best among all the processors. The main job of our balancing algorithm is to let these nodes be fairly distributed among the processors and expanded. We use a fully distributed nearest-neighbor to achieve this goal. The idea is very simple. Before select a best node for expansion, each processor, say p_i, sends the lower bound of its best node to all its neighbors. On receiving this lower bound, a processor will send its best node to p_i if this lower bound is greater than the one of its best node. Note that this scheme will let the best nodes among the processor be fairly distributed among the processors, but it does not involve much communication, because the message size is very small. Hence search overhead can be expected to decrease with low cost of communication. We will discuss this issue in detail in Section 3.

The remaining of this paper is organized as follows: Section 2 describes the branch & bound algorithm, and how to reduce search overhead is given in Section 3. The performance results of the scheme on the nCUBE/2 computer are reported in Section 4. Section 5 gives the conclusion and future researching direction.

2 Parallel Branch & Bound Algorithms

Our program works in an SPMD (Single Program Multiple Data) stye. It has an initialization step to let every processor derive its branch to work on. Then during processing, for a parallel branch & bound algorithm, it is necessary to broadcast the newly computed UB to other processors such that each processor can prune those nodes with lower bound greater or equal to this UB. This work can be done after step 10. Recall in Section 1 that we need a load balancing algorithm to balance the best node each processor. This work can be done after step 19, because the best lower bound of a processor alters only when it has expanded the best node.

When a processor run out of node (after step 20), it sends a ∞, which means it is in the idle state, to its neighbors and enters the idle state. If all the processors enter idle state, the latest idle processor will generate a termination signal and broadcast it to other processors. We adapt the *snapshot termination algorithm* of Huang [6] to do this work. Before receiving the termination signal and output the optimal solution (if any), a processor might receiving a node from its neighbor. If this occurs, the processor should put this node into its priority queue and go to step 5. We'll combine such circumstance with the balance procedure and explain how to implement it in Section 3. These two works (termination detection and node receiving) should be placed after step 20.

Therefore, our parallel branch & bound algorithm should be:

```
1    Create the initial subproblems
2    Insert these subproblems into priority queue $PQ$
3    Set the upper bound $UB$ to $\infty$
4    Set the best solution $BS$ to $\emptyset$
     Do
5        While $PQ$ not empty do
6            Delete the best node $SP$ from $PQ$
7            If $SP$ is feasible then
8                If cost of $SP < UB$ then
9                    $UB$ = cost of $SP$
10                   $BS = SP$
                     Broadcast $UB$ to all processors
11               EndIf
12           Else
13               Expand $P$ to obtain $SP_1, SP_2, \ldots, SP_r$
14               For $i = 0$ to $r$ do
15                   If $LowerBound(SP_i) < UB$ then
16                       Insert $SP_i$ into $PQ$
17                   EndIf
18               EndFor
```

```
            EndIf
            Invoke the Balancing Procedure /* balance with respect to lower bound */
        EndWhile
        Send (Inform, $\infty$) to all neighbors
        Invoke Balance Procedure /* receive nodes */
        Invoke the Snapshot Termination Algorithm
    Until detected TERMINATE signal
    Output $BS$,
```

3 Balancing Algorithm

From the above discussions, one can realize that the balancing of the number of unexamined nodes in each processor is not suitable for a parallel best-first branch & bound algorithm and that using the lower bound as the balancing criterion is better.

The following is our balancing algorithm. In this algorithm, the message comes in the form (A, B), where A represents the message type, and B is the message. There are three message types, *NewUpperBound*, *SubProblem* and *Inform*. They represent the newly computed bound, the subproblem (node), and the information which contains the lower bound of the second best node in a processor, respectively. The function *LowerBound*() returns the lower bound of a node, and the symbol *MySecondBest* represents the node having the second smallest lower bound in the processor's priority queue.

```
Procedure BALANCE
    While receiving (NewUpperBound, $NU$) do If $NU < UB$ then $UB = NU$

    While receiving (SubProblem, $SP$) do
        If LowerBound($SP$) $< UB$ then insert $SP$ to $PQ$
    EndWhile

    For $i = 0$ to $dims - 1$ do HasInform[$i$] = False

    While receiving (Inform, $LB$) from neighbor $i$ do
        BestNode[$i$].LB = $LB$
        HasInform[$i$] = True
    EndWhile

    Sort the array BestNode.LB in descending order
```

```
11  For $i = 0$ to $dims - 1$ do
12      If HasInform[$i$] then
13          If BestNode[$i$].LB > LowerBound(MySecondBestNode) then
14              Delete MySecondBestNode $SP$ from $PQ$
15              Send (SubProblem, $SP$) to neighbor $i$
16              BestNode[$i$].LB = LowerBound($SP$)
17          EndIf
18      EndIf
19  EndFor

20  Send (Inform, LowerBound(MySecondBestNode)) to all neighbors
EndProcedure
```

On receiving a new upper bound, the processor compares it with the current upper bound. If this new upper bound is less than the old one, it replaces the old one (step 1). When a node coming, the processor inserts it into its priority queue PQ if the lower bound of this node does not excess the current upper bound (steps 2–4). Steps 5–9 update the latest information which represent the second best lower bounds of the processor's neighbors. Then a loop is performed to send a node to each neighbor i if the neighbor has a lower bound greater than the lower bound of the second best node in the processor's priority queue (steps 10–19). Finally, the processor sends its lower bound of the second best node to all its neighbors.

The idea behind this algorithm is simple. Every time when a processor performed the branching rule, it will apply this algorithm. The key point is in steps 10–20. After receiving the lower bound information or additional tasks from each neighbor, the processor sorts and checks these lower bounds and sends a node to those neighbors which have lower bounds greater than the one of the second best node in the processor's priority queue. After checking all the neighbors' lower bounds, the processor multi-casts the current lower bound of its second best node to all its neighbors.

Here, we sort the lower bounds because a neighbor which has the larger lower bound is more likely to produce the redundant nodes than those of smaller lower bounds. Besides, the reason why we use the second best node instead of the first best node for checking the lower bound and sending it to other processors is that a processor would be preferable to expand the best node instead of sending it to another processor for expansion, which will cause additional communication.

4 Experimental Results

Here we report the experimental results of our algorithm. All the experiments were done on nCUBE/2 running the traveling salesperson problem. This hypercube computer has 128 processors, and each of them has 4 mega bytes memory. It is well known that a hypercube of dimension k has a diameter k and consists of 2^k processor, where each processor has k direct neighbors.

In our following description, the *runtime* is the execution time excluding the I/O time for data loading and solution outputing. The notation T_1 is used to denote the computation time when we solve the problem with only a single processor. P represents the number of processors and T_P represents the runtime when we parallelize the program onto P processors. We now present our experimental results for the traveling salesman problems below.

4.1 Results

We performed our experiments on different sizes of *asymmetric* traveling salesperson problem, in which the edge costs of in-coming and out-going may be different. All the edges costs are generated randomly with integers which distribute uniformly in the range $[0, 5t]$, where t is the problem size. The problem size (the number of cities) ranges from 30 to 60. For each problem size, we generate 16 data instances for our experiments. The graph for the traveling problems are all fully connected, which means that there is an edge between any two cities. Table 1 shows the average runtime for different sizes of traveling salesperson problems.

As one can see, the runtime increases as the problem size increases, and decrease as the number processor increases. Note also that, for a specific problem size, the runtime decreases nearly by half as we double the number of processors. This gives us the first impression that our results can achieve good efficiency.

Table 2 shows the average speedups for different sizes of traveling salesperson problems. We can see that, using 128 processors, it achieves a speedup of 102.017 for 50-city problems,

problem size	number of processors					
	1	8	16	32	64	128
30	14.98	2.13	1.21	0.81	0.69	0.74
40	124.01	16.39	8.40	4.45	2.52	1.74
50	515.79	66.77	33.83	17.42	9.07	5.06
60	3977.69	—	256.32	129.46	65.38	33.30

Table 1: The runtime for the asymmetric traveling salesman problems.

problem size	number of processors					
	1	8	16	32	64	128
30	1.000	7.024	12.386	18.577	21.707	20.373
40	1.000	7.566	14.756	27.889	49.204	71.179
50	1.000	7.724	15.246	29.616	56.891	102.017
60	1.000	—	15.518	30.724	60.841	119.438

Table 2: The speedups $S = \frac{T_1}{T_p}$ for the asymmetric TSP.

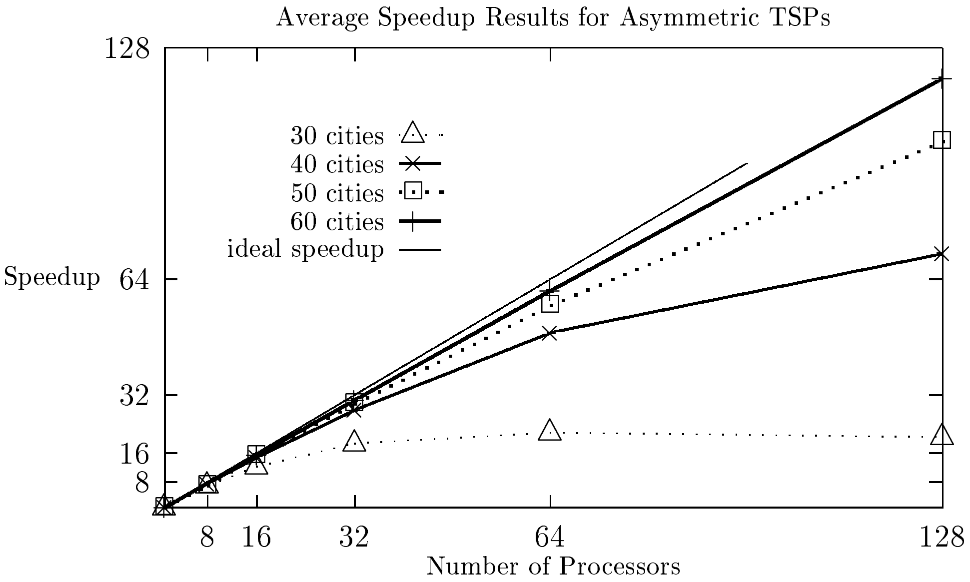
and a speedup of 119.438 for 60-city problems.

Figure 1 shows that our algorithm approaches to nearly linear speedups for large problems.

5 Conclusions and Future Work

We have presented a best-first branch & bound scheme for parallel computers and applied this scheme to solve the traveling salesperson problem on an nCUBE/2 computer. We also demonstrate that this parallel algorithm based upon the lower bound balancing scheme is efficient on a hypercube network of up to 128 processors, which achieves the speedup of 119.44. Although we only experiment our algorithms on 128 processors, we believe that our algorithms are also efficient on larger processors.

Our lower bound balancing algorithm only gather the information of neighbors in a hypercube machine for balancing purpose. It works well on a parallel computer, but it is


Average Speedup Results for Asymmetric TSPs
128
64
32
16
8
Speedup
30 cities
40 cities
50 cities
60 cities
ideal speedup
8
16
32
64
128
Number of Processors