A Simpler Analysis of the Karp-Zhang Parallel Branch-and-Bound Method

Abhiram Ranade

Computer Science Division
University of California
Berkeley, CA 94720

Abstract

Karp and Zhang presented a general method for deriving randomized parallel branch-and-bound algorithms from sequential ones. They showed that with high probability the resulting algorithms attained optimal speedup to within a constant factor, for large enough problems. We present an alternate analysis of their method. Our analysis is considerably simpler, and gives good bounds even for small problem sizes.

1 Introduction

Branch-and-bound algorithms are the most frequently used method in practice for solving combinatorial optimization problems [1,4]. Because of the importance of combinatorial optimization problems in operations research and computer science, and also because these problems are typically very compute intensive, they are a natural candidate for parallel computation.

Karp and Zhang [2] presented a general method for deriving randomized parallel branch-and-bound algorithms from sequential ones. They showed that the resulting algorithms behaved well with high probability on all sufficiently large problem instances (as compared to the number of processors in the parallel computer, defined formally later). In particular, they showed that the execution time of their algorithms was overwhelmingly unlikely to exceed a certain inherent lower bound by more than a constant factor.

In this paper we present an alternate analysis of Karp and Zhang's method. Our analysis is considerably simpler, and gives good bounds also for substantially smaller problem sizes.

In the next section we present the model. In section 3 we state the main result and the algorithm. Section 4 presents the analysis. Section 5 concludes with some remarks and open problems.

2 Model

Karp and Zhang model a generic branch-and-bound optimization problem as a rooted tree H and cost function c over the nodes in H. The goal is to find the leaf of least cost in H, given that the cost function is monotonic, i.e. $c(\text{parent}(v)) < c(v)$, for all nodes v except the root. We shall assume for convenience, as Karp and Zhang do, that no two nodes have the same cost. The input to the algorithm is the root of H, and the other nodes in H are generated as required during execution using a procedure called *node expansion*. When this procedure is applied to a node v it either determines that v is a leaf, or generates the children of v and evaluates their costs. A node can be expanded only if it is the root, or if it has been generated previously during the expansion of its parent.

The computational model consists of p processors numbered 1 through p, each with a local memory, and connected together by a complete network. There is no shared memory, and processors can exchange information only by communicating through the network. There is no global control, but the processors operate synchronously in steps, each of which consists of a computation substep followed by a communication substep. In a computation substep a single processor can perform one node expansion, and an unlimited number of operations of other types. In a communication substep a processor can send out a constant number of messages to other processors. A processor can receive an arbitrary number of messages in a single communication substep. Each message is long enough to accomodate the cost of a node and other information required to subsequently process that node.

The computational model thus charges only for node expansions, and communication. Other operations, e.g. those required for queueing messages are provided for free. This is based on the assumption that in real applications the cost of communication and node expansion would dominate.

3 Main Result

Our main result is that the "local best-first" algorithm of Karp and Zhang [2] described below gives good speedup:

Theorem 1 *Let $\widetilde{H}$ denote the set of nodes in a branch-and-bound tree having cost less than the cost of the least cost leaf, with $\left|\widetilde{H}\right| = n$. Let h denote the length of the longest path from the root to any node in $\widetilde{H}$. Then for any constant k_1 there exists a constant k_2 independent of $\widetilde{H}$ and the number of processors p such that with probability $1-n^{-k_1}$ the local best-first parallel algorithm completes execution in time $k_2(\frac{n}{p}+h+\log n+\log p)$.*

Note that the time required must be at least $\max(n/p, h) = \Omega(n/p+h)$. This is because (1) any parallel algorithm must expand all the n nodes in $\widetilde{H}$, requiring n/p time using p processors, and (2) the h nodes on the path from the root to the farthest node in $\widetilde{H}$ can only be expanded sequentially.

The lower bound is attained whenever the problem size $n = \Omega(p \log p)$. The earlier analysis by Karp and Zhang required $n = \Omega(p^3)$ for attaining the lower bound.

The theorem is proved in section 4. Our proof of the theorem is also substantially simpler than the corresponding proof by Karp and Zhang.

3.1 Local Best-First Algorithm

Karp and Zhang's algorithm is based on the "best-first" heuristic used in sequential branch-and-bound algorithms. The sequential best-first algorithm maintains a priority queue of unexpanded nodes, ordered by their cost. At each step, the node with the least cost is expanded, and the resulting children, if any, are put back into the queue. The algorithm terminates when a leaf is encountered. In the natural parallel extension, named "global best-first", p least cost nodes are expanded at each step rather than just one.

While this is sufficient to find the least cost leaf with minimum number of node expansions, implementing the shared priority queue is hard, as Karp and Zhang point out.

Karp and Zhang instead partition the set of unexpanded nodes among the local memories of the processors, with each processor repeatedly expanding the least cost node from its local partition. They key innovation is that each child resulting from an expansion is sent to the partition of a *randomly* chosen processor. For simplicity, in this paper we give a slightly modified synchronous version of their algorithm, although the result is also valid for their original algorithm.

Specifically, processor i maintains two data structures in its local memory, a priority queue $q(i)$, and a bound $b(i)$ that is known to be the cost of some leaf. Initially all queues are empty, except for one that contains the root, and all bounds are initialized to ∞. The algorithm consists of repeated execution of a loop, each iteration of which consists of $\log p$ invocations of a *node expansion phase* followed by a single *termination detection phase*:

1. Node expansion phase: Processor i picks the node with the least cost from its queue $q(i)$ and expands it. If the node turns out to be a leaf, the bound $b(i)$ is updated if necessary. Else, for each child, a random number j is chosen independently and uniformly from $[1, p]$, and the child is sent to processor j. The nodes received by each processor are stored in its queue. Each node expansion phase executes in a single step.

2. Termination detection phase: The processors first determine the cost of the least cost leaf generated till then by any processor. Then this cost is broadcast to all processors. Each processor then determines if it has any unexpanded nodes with lower cost. If no processor has nodes with a lower cost, then the processors terminate execution. Each termination detection phase can easily be seen to execute in $O(\log p)$ steps.

For the analysis we shall only consider the node expansion phases, and completely ignore the termination detection phases. This is justified because in every iteration of the loop only a constant fraction of the time is spent in termination detection.

4 Analysis

The only use of randomness in the algorithm is in the choice of queues for the nodes. The probability space consists of p^n equally likely elementary outcomes, one for each possible queue assignment. The key insight in the proof is in defining events in the probability space called *delay sequences* that must occur whenever execution takes a long time. Once this correspondence is established, Theorem 1 can be proved by counting all possible delay sequences and estimating their probability.

Let ancestors(v) to denote the set of nodes on the path from the root to a node v, both endpoints included. Let, $h(v) = |\text{ancestors}(v)|$. Note that $\max_{v \in \widetilde{H}} h(v) = h$. Let ancestor$(v, i)$ denote the ith node on the path, with ancestor$(v, 1) =$ root and ancestor$(v, h(v)) = v$.

Definition 1 *An (s, Q, t, T) delay-sequence consists of 4 components:*

1. *A tree node $s \in \widetilde{H}$.*
2. *A sequence Q of queues $q_1, ..., q_{h(s)}$.*
3. *A time interval $[1, t]$.*
4. *A sequence T of disjoint time intervals numbered $T_1, \ldots, T_{h(s)}$ from the earliest to the latest such that $[1, t] = \cup_{i=1}^{i=h(s)} T_i$.*

The delay sequence is said to occur during a particular execution iff each ancestor(s, i) *is processed in q_i, and there exists $V \subseteq \widetilde{H}$ such that:*

1. *V and* ancestors(s) *are disjoint.*
2. *Each node in V arrives into q_i during the interval T_i for some i.*
3. $|V| \geq t - h - \log p$.

Lemma 1 *Suppose the execution time is t. Then some (s, Q, t, T) delay sequence occurs.*

Proof: We shall construct the required sequences.

As s we choose any of the nodes in $\widetilde{H}$ that get expanded in the last iteration of the loop described in the previous section. Clearly, some such node must exist since the last iteration was necessary. The sequence Q is constructed by setting q_i to be the queue in which ancestor(s, i) is expanded.

Let t_i be the time at which ancestor(s,i) is expanded in q_i. Let $t'_0 = 1$, and for $i > 0$ define t'_i such that $T'_i = [t'_i, t_i]$ is the longest interval at each instant of which q_i contained some node from $\widetilde{H}$. Because of the monotonicity of the cost function, we know that ancestor$(s,i) \in \widetilde{H}$. Since ancestor(s,i) stays in q_i during $[t_{i-1}+1, t_i]$, we know that $t'_i \leq t_{i-1}$. Thus the intervals T'_i cover the interval $[1, t_{h(s)}]$. Define

$$T_i = T'_i - \cup_{j=i+1}^{h(s)} T'_j$$

By construction, the intervals T_i are disjoint, and cover the interval $[1, t_{h(s)}]$.

Let V be the set of all nodes expanded in q_i during T_i for any i, excluding ancestors(s). We know that q_i contains some node from $\widetilde{H}$ at each instant in T_i. Thus the nodes expanded at each instant in T_i must also belong to $\widetilde{H}$, since the nodes not in $\widetilde{H}$ have higher cost. Since q_i is known not to contain nodes from $\widetilde{H}$ before the beginning of T_i, these nodes must also have arrived into q_i during T_i. The total number of arrivals is at least $\sum |T_i| \geq t_{h(s)}$, of which at most h can belong to ancestors(s). We know that $t_{h(s)} \geq t - \log p$, since s was expanded in the last loop iteration.[1] Thus $|V| \geq t - h - \log p$.

Q, T and V defined above satisfy all the requirements of a delay sequence, except that the intervals in T may not cover the entire interval $[1, t]$. This is fixed by suitably extending $T_{h(s)}$. ■

Figure 1 illustrates the construction. Suppose $h(s) = 5$, and suppose that ancestors(s) are expanded at times 1, 2, 6, 9, 10 and 11. If corresponding t'_is are respectively 1, 2, 3, 5, 8 and 7, then the intervals T_i are respectively are $[1,1]$, $[2,2]$, $[3,4]$, $[5,6]$, $[]$ and $[7,11]$.

Lemma 2 *Any fixed delay sequence (s, Q, t, T) occurs with probability at most* $p^{-h(s)} \left(\frac{ne}{p(t-h-\log p)}\right)^{t-h-\log p}$.

Proof: In order for the delay sequence to occur ancestor(s,i) must be placed in the ith element q_i of Q. This occurs with probability $p^{-h(s)}$, since there are p choices for each of the ancestors. In addition we require an appropriate sequence V.

[1]Remember that we are assuming for the purpose of the analysis that the termination detection phases do not exist.

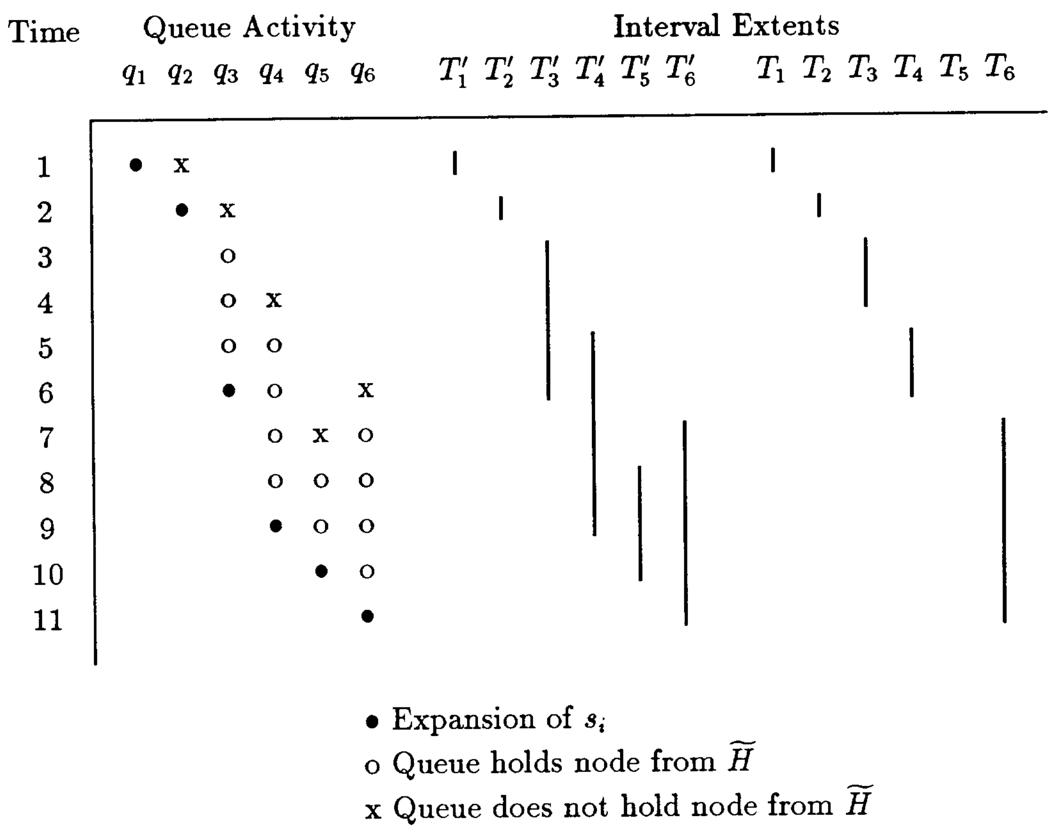
Time
Queue Activity
q_1 q_2 q_3 q_4 q_5 q_6
Interval Extents
T_1' T_2' T_3' T_4' T_5' T_6'
T_1 T_2 T_3 T_4 T_5 T_6
1
2
3
4
5
6
7
8
9
10
11
● Expansion of s_i
o Queue holds node from $\widetilde{H}$
x Queue does not hold node from $\widetilde{H}$

Figure 1: Construction of a delay sequence

For each node v let x_v denote the random variable taking value 1 if node v arrives into a queue q_i during the interval T_i for some i, and 0 otherwise. Define

$$X = \sum_{v \in \tilde{H}-\text{ancestors}(s)} x_v$$

An appropriate V exists iff $X \geq t - h - \log p$. All the $n - h(s)$ variables x_v are independent and identically distributed, and take a value 1 with probability $1/p$. Thus $X \geq t - h - \log p$ with probability at most

$$\binom{n - h(s)}{t - h - \log p} \left(\frac{1}{p}\right)^{t-h-\log p} \leq \left(\frac{ne}{p(t - h - \log p)}\right)^{t-h-\log p}$$

using $\binom{n-m}{r} \leq \binom{n}{r} \leq \left(\frac{ne}{r}\right)^r$. Since this is independent of where the ancestors of s are placed, the lemma follows by multiplying the two probabilities. ■

Proof of Theorem 1: We count the possible (s, Q, t, T) delay sequences for a fixed t and s such that $h(s) = i$. Let n_i denote the number of possible choices for s. The i queues in Q can be chosen in p^i ways. The partition T can be chosen in fewer than $\binom{t+i}{i} < 2^{t+i} < 2^{t+h}$ ways. The total number of delay sequences with $h(s) = i$ is thus $n_i p^i 2^{t+h}$.

Noting that $\sum_i n_i = n$, and using lemma 2, the net probability that some delay sequences occurs is at most:

$$\sum_i n_i p^i 2^{t+h} p^{-i} \left(\frac{ne/p}{t - h - \log p}\right)^{t-h-\log p} \leq \left(e 2^{\frac{t+h+\log n}{t-h-\log p}} \frac{n/p}{t - h - \log p}\right)^{t-h-\log p}$$

For arbitrary k_1, this can be made smaller than n^{-k_1} by choosing $t = k_2(\frac{n}{p} + h + \log p + \log n)$ for some k_2 independent of n and p. ■

We get fairly small constants when n/p is large. For example, for any fixed $\epsilon > 0$, the probability that $t \geq (2e + \epsilon)n/p$ can be made as small as necessary by choosing n/p sufficiently larger than $h + \log p + \log n$.

5 Remarks

In the algorithm we have given above processors need to operate synchronously for the purpose of termination detection. The original algorithm of Karp and Zhang [2] uses a randomized termination detection technique, which enables processors to operate asynchronously. Our analysis is also applicable to their algorithm with little modification. Karp and Zhang also show how their algorithm and its analysis can be adapted for a PRAM. The analysis presented here holds for the adapted PRAM algorithm with minor modifications and with similar results.

The processor model used in this paper consists of a complete network. It would be interesting to extend these results to a sparse network of processors, e.g. a butterfly network possibly using dynamic tree embedding techniques of [3].

Acknowledgements

I am very grateful to Sandeep Bhatt, Dick Karp and Yanjun Zhang for many discussions.

References

[1] E. Balas. Branch and bound methods. In E.L. Lawler *et al*, editor, *The Traveling Salesman Problem.* John Wiley and Sons, 1985.

[2] Richard Karp and Yanjun Zhang. A randomized parallel branch-and-bound procedure. In *Proceedings of the ACM Annual Symposium on Theory of Computing*, pages 290–300, 1988.

[3] Tom Leighton, Mark Newman, Abhiram Ranade, and Eric Schwabe. Dynamic Tree Embeddings on Butterflies and Hypercubes. In *Proceedings of the ACM Symposium on Parallel Algorithms and Architectures*, pages 224–234, June 1989.

[4] C. H. Papadimitriou and K. Steiglitz. *Combinatorial Optimization: Algorithms and Complexity.* Prentice–Hall, 1982.