Marked-union algorithm for Load Balancing in Hypercubic Distributed Hash Tables with Heterogeneous Processors

Junning Liu and Micah Adler *

Abstract. There has been a considerable amount of recent research on load balancing for distributed hash tables (DHTs), a fundamental tool in Peer-to-Peer networks. Previous work in this area makes the assumption of homogeneous processors, where each processor has the same power. Here, we study load balancing strategies for a class of DHTs, called hypercubic DHTs, with heterogenous processors. We assume that each processor has a size, representing its resource capabilities, and our objective is to balance the load density (load divided by size) over the processors in the system. Our main focus is the offline version of this load balancing problem, where all of the processor sizes are known in advance. This reduces to a natural question concerning the construction of binary trees. Our main result is an efficient algorithm for this problem. The algorithm is simple to describe, but proving that it does in fact solve our binary tree construction problem is not so simple. We also give upper and lower bounds on the competitive ratio of the online version of the problem.

1 Introduction

Structured Peer-to-Peer(P2P) systems have been increasingly recognized as the next generation application mode of the Internet. Most of the constructions of current structured P2P systems, such as CAN [17], Chord [19], Pastry [18] and TAPESTRY [21] etc. are *distributed hash tables*(DHTs), which determine where to store a data item by first hashing its name to a prespecified address space, and then partitioning this address space across the processors of the P2P system in a manner that allows a specific hash address to be found relatively easily. There has been a considerable amount of research on various aspects of these DHT systems. The aspect of DHT systems that motivates our work is load balancing. This is crucial to a DHT: a major design goal of P2P systems is to provide a scalable distributed system with high availability and small response time.

In this paper, we consider load balancing in a DHT consisting of heterogeneous processors. While most previous work has analyzed DHTs under the assumption that all processors are identical, real DHTs consist of processors with significantly different characteristics in terms of computational power, bandwidth availability, memory, etc. One way to extend results for the homogeneous

* Department of Computer Science, University of Massachusetts, Amherst, MA 01003-4610, USA. This work supported by NSF grants EIA-0080119, CCR-0133664, and ITR-0325726. Email: {liujn,micah}@cs.umass.edu

case to heterogeneous systems is to use *virtual processors*: powerful processors pretend to be multiple less powerful processors. This approach has the drawback that a processor must maintain pointers to other processors for each virtual node it represents, and most of these pointers will be distinct. Since each pointer is a connection that must be kept alive in the presence of processor arrivals and departures, this can be quite expensive. Other existing techniques for dealing with heterogeneous processors include data migration. Data migration can be a good approach when data is highly dynamic, but is usually unfavorable when processors are highly dynamic. In fact, when processors arrive and leave frequently, the migration process may have a cascading effect of migration [3] that further deteriorates bandwidth usage and data consistency. Data migration can also introduce an overhead cost for locating the data.

This paper tries to balance the load without using virtual servers or migration methods. We study one variant of the DHT, the *hypercubic hash table*(HHT), a variant of CAN [17] described in [2]. The HHT partitions the address space using a full binary tree. The leaves of the tree correspond to the processors of the DHT. We assign a 0 to every left branch in the tree, and a 1 to every right branch. Each processor stores all hash addresses whose prefix matches the string obtained by following the path from the root to the leaf for that processor, and thus the fraction of the address space stored at a processor i is $1/2^{l_i}$, where l_i is the distance in the tree from i to the root.

In the hypercubic hash table, the arrival of a new processor i is handled by *splitting* an existing leaf node j. To do so, the leaf node for j is replaced by an internal node with two children i and j. The address space is then reallocated as appropriate for this change. Deletions are handled in an analogous manner. [2] analyzed distributed techniques for choosing which node of the tree to split in order to keep the resulting tree as balanced as possible. In particular, the objective was to minimize the ratio between the maximum fraction of the address space stored at any processor, and the minimum fraction of the address space stored at any processor.[1]

For heterogeneous processors, we want to store larger fractions of the address space on some of the processors, and thus we no longer want as balanced a tree as possible. We assume that every processor i has a single measure of its power, which we call its size, and denote by 2^{s_i} (it will be more convenient for us to use the logarithm of the sizes). Instead of balancing the load across the processors, we now wish to construct a tree that balances the *load density*, where the load density on processor i is defined to be $ld_i = L\frac{1}{2^{l_i}}\frac{1}{2^{s_i}} = \frac{L}{2^{s_i+l_i}}$, L is the total load and l_i is the height (distance from the root) of node i in the tree. Our goal is to minimize the quantity $\frac{\max_i ld_i}{\min_i ld_i}$. This criteria is a natural extension

[1] Note that an equal partition of the address space does not necessarily imply an equal amount of data at each processor. However, when the number of data items is large compared to the number of processors, and the data items have approximately the same size, then this will be a good estimate of the load balance. The case of heterogeneous processors where the number of data items is close to the number of processors has been studied in [5].

of the load balancing criteria of [2], and characterizes the application's major requirements. An alternative criteria would be to minimize the maximum load density. A drawback to considering the maximum load density without regard to the minimum load density is that it can result in processors with a large size not having a large fraction of the address space: i.e., powerful processors may be underutilized. In fact, our main result actually demonstrates that it is possible to minimize both criteria simultaneously: we describe an algorithm that finds the tree with the minimum load density ratio, but this tree also minimizes the maximum load density.

While the eventual goal of a load balancing technique would be to develop a distributed and online algorithm, it turns out that minimizing the load density ratio with heterogeneous processors is challenging even in a centralized and offline setting. Thus, in this paper we focus on that setting; developing distributed algorithms that utilize the techniques we introduce for the centralized setting is an interesting and important open problem. The centralized problem reduces to the following simple to state algorithmic problem: given a set of nodes $S = \{p_1, p_2, \ldots, p_n\}$ with weights $s_1, \ldots, s_n$, construct a full binary tree with leaf set S. Let the *density* of node i in this tree be $s_i + l_i$. The binary tree should minimize the difference between the maximum density of any node and the minimum density of any node. This is a quite natural question concerning the construction of binary trees, and thus we expect that algorithms for this problem will have other applications. We also point out that the requirement that the tree be full (i.e., that every non-leaf node have degree exactly 2) is necessary for our application: if any node has degree 1, there will be a portion of the address space not stored at any processor.

This algorithmic problem is reminiscent of building optimal source codes; for that problem Huffman's algorithm (see [6] for example) provides an optimal solution. In fact, if we use the simpler optimization criteria of minimizing the maximum density, then Golumbic's minimax tree algorithm [11], a variant of Huffman's algorithm, provides the optimal solution. In this variant, when two nodes are merged into a single node, the new node has a size equal to the maximum of the two merged nodes plus one (instead of the sum of the two nodes as in Huffman's algorithm). Similarly, if we must maximize the minimum density, then using the minimum of the two merged nodes plus one gives the optimal solution. What makes our problem more difficult is that we must simultaneously take both the maximum density and the minimum density into account. We point out that there are simple example inputs for which it is not possible to construct a tree that simultaneously minimizes the maximum density and maximizes the minimum density.

Our main result is the introduction and analysis of a polynomial time algorithm, called the *marked-union* algorithm, that, given a set of nodes with integer weights, finds a tree that minimizes the difference between the maximum density and the minimum density. This algorithm is in fact based on the maximin version of Golumbic's algorithm, and as a result also maximizes the minimum density (which corresponds to minimizing the maximum load density on any pro-

cessor). The difference between our algorithm and Golumbic's is that ours relies on an additional tie breaking rule that determines which nodes to merge. This tie breaking rule is crucial: Golumbic's algorithm without the tie breaking rule can result in a density difference that is considerably worse than the optimal.[2]

The fact that the marked-union algorithm does in fact minimize the density difference is not obvious, and in fact the proof of this is somewhat involved. As further evidence that minimizing the density difference is not as easy to deal with as the minimax and the maximin optimization criteria, we point out that Golumbic's algorithms for those problems also work for the case of non-integer s_is. On the other hand, the marked-union algorithm does not always return the optimal solution for non-integer inputs, and in fact the complexity of that case is an open problem. Nevertheless, the marked-union algorithm does provide a 2-approximation (which is a 4-approximation to the original load balancing problem, before we moved to using the logarithm of the sizes and the load.)

We also provide lower and upper bounds on the competitive ratio of the online version of this problem. In this scenario, the algorithm has access to the entire current tree, and, on an arrival, must decide which node to split to achieve as good load balancing as possible. We assume that both the adversary and the algorithm have access to a list of possible sizes before the sequence arrives, and that the cost of an input sequence is the final load imbalance. This model captures the system's long term stable status load balancing needs. We demonstrate that in this scenario, the trivial algorithm of keeping the tree as balanced as possible, without regard to the sizes, is essentially the best that one can do for the worst case performance of this online scenario.

This paper is organized as follows: In Section 2, we define the problem more formally and present the marked-union algorithm. In Section 3, we prove the optimality of the marked-union algorithm. In Section 4 we describe our results for the competitive analysis of the online algorithm. Finally, in Section 5 we discuss future work. Due to space limits, most of our proofs are in the Appendix.

1.1 Related work

There has been a number designs on how to build a scalable structured P2P system, including CAN [17], HHT [2], Chord [19], Viceroy [13], Pastry [18], Tapestry [21], Distance Halving DHT [14], and Koorde [12]. Most of these DHTs achieve a $\log n$ ratio of load balancing with high probability, although many also consider schemes for improving this to $O(1)$, sometimes at the cost of other performance measures. CAN [17] allows a measure of choice to make the load more balanced. This technique is analyzed in [2] for the closely related HHT; that paper demonstrates that it achieves a constant load balance ratio with high

[2] For example, on an input of 2^n-1 nodes of the same size, if ties are broken arbitrarily, then the density difference can be as bad as $n-1$, whereas the optimal for this input is 1. We also point out that while this tie breaking rule does not reduce the maximum load density in the system, it can significant reduce the number of processors that must deal with this maximum load density.

probability. All of the work mentioned above considers only the homogenous scenarios.

Work on heterogenous load balancing has just begun. [16] and [1] provide heuristics for dealing with heterogenous nodes. In [16], migration and deletion of virtual servers is used in a system based on Chord. This system is studied using simulations. [1] uses the P-Grid Structure. There, each node is assumed to know its own best load and the system reacts accordingly to balance the load. This provides further motivation for our offline algorithm: it demonstrates that determining the optimal allocation for the processors currently in the system is a useful tool for load balancing of heterogenous P2P systems.

Perhaps the closest work to ours is [7], which introduces a protocol and proves that it balances the heterogenous nodes' load with high probability for a variant of the Chord DHT, assuming that a certain update frequency is guaranteed. As they also point out, this system relies on moving data items to arbitrarily destinations, which can lead to difficulties with data location.

To the best of our knowledge, there has been no competitive analysis of load balancing problems for structured P2P systems.

There has been a considerable amount of work on Huffman codes and its variants, which is related to our problem of constructing binary trees. [20] showed that if the input is pre-sorted, the running time of Huffman's algorithm can be reduced to be $O(n)$. [15] found a structure(*lattice*) on all the binary Huffman codes for a finite alphabet and showed that the tree's imbalance can be a partial index for the lattice. Their result showed Huffman coding can be viewed as optimizations over a lattice. [11] introduced the minimax tree algorithm already mentioned. [10] generalizes the Huffman code to a non-uniform encoding alphabet with the same goal of minimizing the expected codeword length. They gave a dynamic programming algorithm that finds the optimal solution for some cases, as well as a polynomial time approximation scheme for others.

There is recently some work studying constrained version of optimization problems arising in binary tree construction. [8] studied the problem of restructuring a given binary tree to reduce its height to a given value h, while at the same time preserving the order of the nodes in the tree and minimizing the displacement of the nodes. Their result provides an explicit tradeoff between the worst-case displacement and the height restriction. [9] considers the problem of constructing a nearly optimal binary search trees with a height constraint. Interestingly, they do not take the probabilities of the nodes as input but instead use a given binary search tree and the rank of the nodes' probabilities.

2 Problem statement and algorithm

We next provide a more formal description of the problem. Suppose we have m different possible processor sizes, $S_m > S_{m-1} > \ldots S_i > \ldots > S_1 > 0$. Each size is an integer, and there are $n_i > 0$ processors for size S_i. Define a ***solution tree*** to be a full binary tree plus a bijection between its leaf nodes and the input processors. Let T be any solution tree. Denote the depth of a processor R in a

tree T as l_R, the size of it as s_R (the root node has a depth of zero). The *density* of processor R is $d_{T,R} = s_R + l_R$. [3] The *density difference* of T is the maximum density minus the minimum density. Our goal is to find the *optimal solution tree* T^* that achieves the minimum density difference $\text{Minimum}_T(\max_R d_{T,R} - \min_R d_{T,R})$.

For this problem we design the marked-union algorithm. In order to describe it, we start with some notation. This algorithm will repeatedly merge *nodes*. To start with, the set of input processors form the set of nodes. We refer to these initial nodes as *real nodes*. Each node maintains a minimum density attribute $d_{min} = x$. We sometime refer to a node as (x). The real node for a processor has the processor's size as its d_{min}.

The algorithm proceeds via a series of *unions*: an operation that consumes two nodes, and produces a *virtual node*. The nodes it consumes can be real or virtual. When a union operation consumes nodes (d_{1min}) and (d_{2min}) the new node it generates will be $(\min(d_{1min}, d_{2min}) + 1)$.

The aspect of our algorithm that makes it unique as a variant of Huffman's algorithm is the concept of a *marked node*. A virtual node is labeled as marked when it is formed if either of two conditions hold: (1) the two input nodes that create that node have different values of d_{min}, or (2) either of the two input nodes was already marked. The marked-union algorithm will use this marking to break ties; this will lead us to the optimal solution.

We are now ready to describe the algorithm as below:

The Marked-Union Algorithm:

(1) **Initialization:**
Sort the input nodes in decreasing order of size. Construct the working queue using the nodes in this order (from left to right).
(2) **Processing the nodes**
(3) While (more than one node remains in the working queue) {
(4) Union the rightmost two nodes, resulting in a new node $V(d)$;
(5) Insert $V(d)$ into the working queue in the following order:
From left to right, nodes decrease according to d_{min}; for nodes with the same d_{min}, marked nodes appear to the left of unmarked nodes, otherwise, ties are broken arbitrarily.
(6) } end while loop

Fig. 1. Marked-Union algorithm.

Note that all the input nodes start as unmarked real nodes and we will prove in Lemma 1 later that there will never be more than one marked node. The algorithm's running time is just $O(N \log N)$, where $N = \sum_{i=1}^{m} n_i$. Also, we can

[3] We will use the density for the offline analysis, but use the load density for the cost function of online competitive analysis. Note their relationship is $d_{T,R} = \log L - \log ld_{T,R}$

use it to return a solution tree: treat each real node as a leaf node, and on any union operation, the resulting virtual node is an internal node of the tree with pointers to the two nodes it consumed. Our main result will be to show that this tree is an optimal solution tree, which we call the *best tree*.

We also point out that we can also get the maximum density difference of the resulting tree at the same time. To do so, we modify the algorithm by adding a new attribute to any marked node which stores the maximum density of any node in its subtree. If we represent a marked node as $V(d_{min}, d_{max})$, modify the union operation as: $(d_{1min}) \cup (d_{2min}) = (\min(d_{1min}, d_{2min}) + 1, \max(d_{1min}, d_{2min}) + 1)$ when $d_{1min} \neq d_{2min}$; and $(d) \cup (d_{min}, d_{max}) = (\min(d, d_{min}) + 1, \max(d, d_{max}) + 1)$ when $d_{1min} = d_{2min} = d$. As was already mentioned, we do not need to define a union operation for the case where two marked nodes are consumed. By defining union this way, it is easy to see that d_{min} will always be the minimum density in the subtree rooted at that node, and d_{max} will always be the maximum density. Thus, the density difference of the final marked node $d_{max} - d_{min}$ will be the maximum density difference of the tree. If the final node is unmarked, all nodes have the same density.

2.1 Marked-union with the splitting restriction

Before we prove the optimality of the tree resulting from this algorithm, we show that this tree can be constructed under the restriction that it is built up using a sequence of splitting operations. In other words, we assume that the nodes must be inserted into the tree one at a time, and each such insertion must be handled by splitting a node of the tree. In order to do so, we first sort the processor by size. We assume that the input processor sequence is $p_m, p_{m-1}, \ldots, p_2, p_1$ with sizes satisfying $s_m \geq s_{m-1} \geq \ldots \geq s_2 \geq s_1$. We next run the marked-union algorithm and record the resulting depth l_i^* for each processor p_i.

We then process the whole input sequence from left to right. For the first node, we use it as the root node of the solution tree. For each subsequent node p_j, choose the leftmost node p_i of the already processed input sequence that has a current depth $l_i < l_i^*$ in the solution tree thus far. As we demonstrate in the Appendix in the proof of Theorem 1, such a node must exist. Split the node p_i and add p_j as its new sibling. When all processors have been processed, return the final tree. Let T^* be this tree.

Theorem 1. *In the tree T^*, each processor p_i will have $l_i = l_i^*$.*

3 The optimality of the marked-union algorithm

Theorem 2. *The marked-union algorithm returns an optimal solution tree w.r.t. the minimum density difference.*

This section is devoted to proving Theorem 2. We start with a high level overview of this proof. The first step (Section 3.1) is to define the concept of rounds. Very roughly, there is one round for each different size processor that

appears in the input, and this round consists of the steps of the algorithm between when we first process nodes of that size and when we first process nodes of the next larger size from the original input. Once we have defined rounds, we prove a number of properties concerning how the virtual nodes in the system evolve from one round to the next.

The second step of the proof (Section 3.2) defines *regular trees*, a class of solution trees that satisfy a natural monotonicity property. There always exists some optimal solution that is a regular tree, and thus we prove a number of properties about the structure of these trees. In particular, we examine sets of subtrees of a regular tree. There is one set of subtrees for each processor size, and these subtrees are formed by examining the lowest depth where that processor size appears in the tree. The third step (Section 3.3) uses the results developed in the first two steps to show that the tree produced by the marked-union algorithm has a density difference that is no larger than any regular tree. We do so via an induction on the rounds of the algorithm, where the virtual nodes present after each round are shown to be at least as good as a corresponding set of subtrees of any regular tree.

3.1 Analysis of marked-union algorithm

Now let us do the first step of the proof for Theorem 2. We first introduce the definition of Rounds for the purpose of analyzing the algorithm. Note that rounds is not a concept used in the algorithm itself.

Let us divide the algorithm into m *rounds*, each round contains some consecutive union operations: Round_1's start point is the start point of the algorithm, before any union operation is performed. Round_i's start point is the end point of Round_{i-1}; its end point is right before the first union operation that will either consume a real node of size S_{i+1} or will generate a virtual node with a size bigger than S_{i+1}. All unions between these two points belongs to Round_i . Round_m 's end point is when we have a single node left as the algorithm halts.

Lemma 1. *There can be at most one marked node throughout one run of the algorithm, and the marked node will always be a node with the smallest d_{min} in the working queue.*

Let Δ_i be $d_{max} - d_{min}$ of the marked node before Round_{i+1} after Round_i , and $\Delta_i = 0$ if there is no marked node at that time. We will prove in Lemma 2 that after each Round all virtual nodes will have the same d_{min}, Let d_i be the d_{min} of the virtual nodes after Round_i.

Lemma 2. *Before Round_{i+1}, after Round_i, we have two possible cases:*

a) *Single node $(d_i, d_i + \Delta_i)$, with $d_i < S_{i+1}, \Delta_i \geq 0$;*

b) *A set of virtual nodes with the same d_{min}, which are $k (k \geq 0)$ unmarked virtual nodes $V(d_i)$ and another marked or unmarked node $V(d_i, d_i + \Delta_i)$, $d_i = S_{i+1}, \Delta_i \geq 0$.*

Lemma 3. *There are two possible cases for Δ_i:*
case a): $\Delta_i = \max(S_i - d_{i-1}, \Delta_{i-1})$
case b): $\Delta_i = 1$, $\Delta_{i-1} = 0$, *and* $d_{i-1} = S_i$

If after Round_i, case b) happens, then we call Round_i a *b-round*, if not, then Round_i is a normal round. Note the first marked node appears at case b) and this case can only happen once: there could be at most one b-round throughout one run of the algorithm.

Corollary 1. $\Delta_i = 0 \Rightarrow \Delta_{i-1} = \Delta_{i-2} = \ldots = \Delta_1 = 0$, $d_{j-1} = S_j \ \forall\, j \leq i$.

Theorem 3 (Gol76). *The marked-union algorithm maximizes the final d_{min}.*

3.2 Definitions and Conclusions about Regular Trees

Now let us do the second step of the proof of Theorem 2. We first introduce the concept of a special class of solution trees then prove some properties of it. A *regular tree* is a Solution Tree which satisfy that for any two processors, if $s_i > s_j$ then $l_i \leq l_j$.

Observation 1 *There exists an optimal solution tree that is a regular tree.*

For any regular tree, let $l_{i,\max}$ be the maximum depth of the leaf nodes with Size S_i. By the definition of regular tree, it is easy to see $l_{m,max} \leq l_{m-1,max} \leq \ldots \leq l_{2,max} \leq l_{1,max}$.

Then if we look through across the regular tree at depth $l_{i,\max}$, the nodes we get are some internal nodes or leaf nodes. For each of the internal nodes I_V, all its descendants and I_V form a subtree with I_V as the root; for each leaf node with a processor size of $s < S_i$, it can also be viewed as a subtree of one node, and thus we have a set of subtrees. Define $k_{i-1}(k_{i-1} \geq 1)$as the number of these subtrees. Define the set of these subtrees as $V_{i-1} = \{\nu_{i-1,1}, \nu_{i-1,2}, \ldots, \nu_{i-1,j}, \ldots, \nu_{i-1,k_{i-1}}\}$. Furthermore, we define $k_m = 1$: V_m has only one subtree, the regular tree itself.

From its definition, all subtrees of V_i have the same depth in the regular tree:$l_{i+1,\max}$(the depth of a subtree means its root's depth). From the definition of Regular Tree and $l_{i,\max}$, we know all the leaf nodes with size S_i lie in a depth inclusively between $l_{i+1,\max}$ and $l_{i,\max}$ in the regular tree, thus $\nu_{i,1}, \nu_{i,2}, \ldots, \nu_{i,j}, \ldots, \nu_{i,k_i}$ can be viewed as being constructed from $\nu_{i-1,1}, \nu_{i-1,2}$, $\ldots, \nu_{i-1,j}, \ldots, \nu_{i-1,k_{i-1}}$ plus the n_i leaf nodes of size S_i. Notice that in V_1, we don't have subtrees of V_0 below, they will only be formed by leaf nodes of size S_1. Also the roots of $\nu_{i-1,1}, \nu_{i-1,2} \ldots \nu_{i-1,j}, \ldots \nu_{i-1,k_{i-1}}$ all have the same relative depth of $l_{i,\max} - l_{i+1,\max}$ in its residing subtree in V_i. This depth is also the max relative depth in V_i for real nodes with size S_i, and there is at least one leaf node with size S_i that lies at such a relative depth in one of the subtrees of V_i.

Let *relative density* be a node's density with respect to some subtree of the regular tree, i.e., if a node of size S_i has a depth of l_i in a regular tree, and the node is also in some subtree with l_r as its root's depth in the regular tree, then the node's relative density with respect to the subtree is $(l_i - l_r) + S_i$. From above

we know the set of subtrees in V_i all have the same root depth at the regular tree, so we can talk about relative density with respect to a set of subtrees. Let $d_{min,i}$ be the minimum relative density w.r.t. V_i of all the leaf nodes that lie in any of the subtrees of V_i. Let $d_{max,i}$ be the maximum relative density. Let $\Delta_{d_i} = d_{max,i} - d_{min,i}$.

Claim 1 *For any regular tree,*

$$\Delta_{d_i} \geq \Delta_{d_{i-1}} \tag{1}$$

$$\Delta_{d_i} \geq |S_i - d_{min,i-1}| \tag{2}$$

Corollary 2. *If after Round$_i$, we have a single virtual node in our algorithm, then compared with the V_i of any regular tree on the same input,*

$$d_i \geq d_{min,i} \tag{3}$$

3.3 Comparing marked-union's output to regular trees

Now we have prepared enough for the proof of Theorem 2. By Observation 1, there exist at least one regular tree that is optimal. If we can prove no regular trees on the same input can beat our algorithm, then we prove marked-union algorithm is optimal. So the Lemma below is what we need to prove.

Lemma 4. *For any given input, let* $\min(\Delta_{d_i})$ *be the minimum Δ_{d_i} of all regular trees for the same input. For any Round$_i$, if it is the last round or it is not the b-round, then* $\min(\Delta_{d_i}) \geq \Delta_i$.

Lemma 4(proof in Section B of Appendix) is the core part of the whole optimality proof, combined with Observation 1, it shows that the marked-union algorithm's output will be no worse than any regular tree in terms of density difference, since there exist at least one optimal solution tree which is also a regular tree. We have proved that the marked-union algorithm returns an optimal tree, or it is an optimal algorithm that minimize the solution tree's density difference.

4 Competitive Analysis

The marked-union algorithm is designed for the offline case, where we know the entire set of available processors before any decisions need to be made. We here consider the online case, where the algorithm maintains a current tree, and processors arrive sequentially. After each such arrival, the algorithm must make a decision on how to deal with this arrival without any knowledge of future arrivals. We further assume that the algorithm is restricted to only performing split operations: only one other processor currently in the system can be effected on an arrival. This would allow the DHT to efficiently rebalance any data currently in the system. We study the worst case degradation of performance due to incomplete information of the coming input, using competitive analysis [4]. We provide a model for competitive analysis in our scenario. We also demonstrate that it is not possible for a deterministic algorithm to do much better than simply keeping the tree as balanced as possible. This suggests the need for randomization as is used in most existing load balancing schemes for DHTs.

4.1 The Model

We describe our model in terms of three parties: the adversary (which chooses the input), the online player (the algorithm), and the third party. At the start of the online problem, the third party specifies some common knowledge, known as *problem parameters*, to both the online player and the adversary. In our model, the common knowledge is a set of possible processor sizes. The adversary then serves the online player with a sequence of processor sizes. The only information the online player has prior to an arrival is that the size of that arrival must be in the set of possible sizes given by the third party. The adversary also determines when the input sequence halts; the online player only sees this when it actually happens. Without a restriction on possible sizes (as is imposed by the third party), the adversary can make the performance of any online algorithm arbitrarily bad, and thus this aspect of the model gives us a way to compare the performance of online algorithms. Furthermore, it is reasonable to assume that real DHTs have some knowledge of the arriving processor's sizes, e.g. the range of the sizes.

Since the load density (as opposed to density) represents our real performance metric for DHTs, we use load density in this section. In particular, we use the load density ratio $2^{d_{max}-dmin}$ of the final solution tree when the adversary decides it is time to halt. We point out that instead of the cost of the final solution tree, other choices to consider would be a sum of costs or the maximum cost [4]. However, for our case, when the tree is growing, it is not in a stable situation yet. This transition period should be negligible because it is not a typical working state for the system; we are really concerned with the long term effects, for which our model is more appropriate.

4.2 Bounds for competitive ratios

For minimization optimization problems, the competitive ratio [4] for an online algorithm is defined as the worst case ratio, over all allowed input sequences, between the cost incurred by an online algorithm and the cost by an optimal offline algorithm. As was already mentioned, we use the original load density ratio as the cost function. Since the node density we used earlier is the log of a node's load density, the log value of competitive ratio is then the difference of density differences between the online algorithm and the optimal offline algorithm. We sometimes use the log value of the real competitive ratio for convenience. Define the set of input processor sizes which the third party specifies as $\Re = \{S_i | 1 \leq i \leq m\}$, with $S_m > S_{m-1} > \ldots > S_2 > S_1$. Define $\Delta S = S_m - S_1$, i.e., the largest size difference for the processors specified by the third party. Since the case where the third party only specifies a single sized processor is very easy to analyze, we only consider the heterogeneous case where $m > 1$. Define $S_{max} = S_m$ as the largest processor's size, $S_{min} = S_1$ as the smallest processor's size. We first provide the lower bound for the competitive ratio.

Theorem 4. *For any deterministic online algorithm for this load balancing problem, the competitive ratio is at least* $2^{\Delta S}$.

We point out that the proof in Appendix for this is made more complicated by the fact that it is described for an adversary that has a further restriction. In particular, if we further limit the adversary that it must introduce at least one processor of each size given by the third party, then the bound still holds. This demonstrates that the lower bound really is a function of the difference between the largest processor and the smallest processor. Of course, any lower bound for this restricted scenario still holds for the unrestricted scenario.

We next point out that a very simple algorithm can almost match this lower bound. In particular, the lower and upper bounds are within a factor of 2.

Theorem 5. *There is an online algorithm with a competitive ratio of* $2^{\Delta S+1}$ *for this load balancing problem.*

From this we also know an upper bound for the best competitive ratio any online algorithm could have.

5 Future work

Since our marked-union algorithm efficiently solves the centralized offline optimization problem, the most important open problem is to design a distributed online algorithm. It is possible that algorithms based on ideas from the marked-union algorithm may provide some direction. Although the online lower bound is somewhat discouraging, it is possible that one can do better using randomization. Also, it seems likely that stochastic models of input sequences are easier to deal with. Finally, looking at the case of multiple, independent measures of a processor's power looks like an interesting but challenging question to pursue.

References

1. K. Aberer, A. Datta, and M. Hauswirth. The quest for balancing peer load in structured peer-to-peer systems, 2003.
2. Micah Adler, Eran Halperin, Richard M. Karp, and Vijay V. Vazirani. A stochastic process on the hypercube with applications to peer-to-peer networks. In *Proceedings of the thirty-fifth ACM symposium on Theory of computing*, pages 575–584. ACM Press, 2003.
3. Masaru Kitsuregawa Anirban Mondal, Kazuo Goda. Effective load balancing of peer-to-peer systems. In *IEICE workshop on Data engineering*, number 2-B-01, 2003.
4. Allan Borodin and Ran El-Yaniv. *Online Computation and Competitive Analysis.* Cambridge University Press, Cambridge, 1998.
5. John Byers, Jeffrey Considine, and Michael Mitzenmacher. Simple load balancing for distributed hash tables.
6. Thomas M. Cover and Joy A. Thomas. *Elements of Information Theory.* Wiley Series in Telecommunications. John Wiley & Sons, New York, NY, USA, 1991.
7. Matthias Ruhl David R. Karger. New algorithms for load balancing in peer-to-peer systems. Technical Report LCS-TR-911, UC Berkeley, July 2003.

8. William S. Evans and David G. Kirkpatrick. Restructuring ordered binary trees. In *Symposium on Discrete Algorithms*, pages 477–486, 2000.
9. Gagie. New ways to construct binary search trees. In *ISAAC: 14th International Symposium on Algorithms and Computation (formerly SIGAL International Symposium on Algorithms), Organized by Special Interest Group on Algorithms (SIGAL) of the Information Processing Society of Japan (IPSJ) and the Technical Group on Theoretical Foundation of Computing of the Institute of Electronics, Information and Communication Engineers (IEICE))*, 2003.
10. Mordecai J. Golin, Claire Kenyon, and Neal E. Young. Huffman coding with unequal letter costs (extended abstract).
11. M. Golumbic. Combinatorial merging. *IEEE Transactions on Computers*, 24:1164–1167, 1976.
12. M. Frans Kaashoek and David R. Karger. Koorde: A simple degree-optimal distributed hash table.
13. Dahlia Malkhi, Moni Naor, and David Ratajczak. Viceroy: A scalable and dynamic emulation of the butterfly. In *Proceedings of the 21st ACM Symposium on Principles of Distributed Computing*, 2002.
14. Moni Naor and Udi Wieder. Novel architectures for p2p applications: the continuous-discrete approach, 2002.
15. D. Stott Parker and Prasad Ram. The construction of huffman codes is a submodular ('convex') optimization problem over a lattice of binary trees.
16. Ananth Rao, Karthik Lakshminarayanan, Sonesh Surana, Richard Karp, and Ion Stoica. Load balancing in structured p2p systems, 2003.
17. Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard Karp, and Scott Shenker. A scalable content addressable network. In *Proceedings of ACM SIGCOMM 2001*, 2001.
18. Antony Rowstron and Peter Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. *Lecture Notes in Computer Science*, 2218:329–350, 2001.
19. Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proceedings of the 2001 conference on applications, technologies, architectures, and protocols for computer communications*, pages 149–160. ACM Press, 2001.
20. J. van Leeuwen. On the construction of Huffman trees. In *Proceedings of the 3rd International Colloquium on Automata, Languages, and Programming*, pages 382–410, 1976.
21. B. Y. Zhao, J. D. Kubiatowicz, and A. D. Joseph. Tapestry: An infrastructure for fault-tolerant wide-area location and routing. Technical Report UCB/CSD-01-1141, UC Berkeley, April 2001.

A Marked-Union with splitting restriction

Proof of Theorem 1:

At any time-step, let $\wp = \{i | p_i$ has already been processed by the algorithm$\}$. We first prove that throughout the course of the algorithm, $\forall i \in \wp$, $l_i \leq l_i^*$. We prove this by induction on the number of nodes that have been processed. The base case is when only one node has been processed, and follows easily from the fact that there is only a single node in the tree. Next, assume the inductive hypothesis

and that there is still at least one node to process. Note that at this step, there must be some node for which $l_i < l_i^*$. Otherwise, by the inductive hypothesis, it must be the case that $\forall i \in \wp$, $l_i = l_i^*$. Furthermore, since T^* is a full binary tree, it must be the case that $\sum_{i \in \wp} \frac{1}{2^{l_i}} = 1$. However, since the marked-union algorithm also produces a tree, it must be the case that $\sum_i \frac{1}{2^{l_i^*}} = 1$. Since there is some $i \notin \wp$, this is a contradiction.

Let i be such that $l_i < l_i^*$. After node p_i is split, l_i has increased by 1, and thus it is still the case that $l_i \leq l_i^*$. Furthermore, let p_j is the node that is inserted at this step. In the marked-union algorithm, the fact that p_i is to the left of p_j in the initial working queue implies that $l_i^* \leq l_j^*$, so $l_j = l_i \leq l_i^* \leq l_j^*$. This completes the inductive step. Thus, in the final tree T^*, $\forall i$, $l_i \leq l_i^*$. This, combined with the fact that $\sum_i \frac{1}{2^{l_i}} = \sum_i \frac{1}{2^{l_i^*}} = 1$ implies that $l_i = l_i^*$. This last step can be proved by contradiction: if there exists a node with $l_i < l_i^*$, then $\sum_i \frac{1}{2^{l_i}} > \sum_i \frac{1}{2^{l_i^*}} = 1$.

B Optimality of Marked-Union algorithm

Proof of Lemma 1:

We prove this by induction on the number of union operations. Before the first union operation, there is clearly at most one marked node, which will be a node with the smallest d_{min}. Assume that after the kth union, this is still true. For the $(k+1)$st union, if it does not generate a marked node then neither of the two input nodes of this union is marked, thus the assumption will still be true after the $(k+1)$st union. If the $(k+1)$st union generates a marked node, since a union operation always consumes the two smallest d_{min} nodes, if there is no previous marked node, the two nodes consumed by the $(k+1)$st union must have different densities, $d_{min1} > d_{min2}$. The resulting marked node must have a $d_{min} = d_{min2} + 1 \leq d_{min1}$, less than or equal to the larger input node's min density, so it has to be the minimum d_{min} after the removal of the two input nodes. If there is already one marked node, then from the assumption it has a smallest $d_{min} = d_s$, since the algorithm always inserts the marked node into a position which is leftmost within all nodes with the same d_{min}, we know the $(k+1)$st union will consume all nodes with a min density of d_s, or after this union, there will be no node with $d_{min} = d_s$ left in the queue. Also the resulting new marked node has $d_{min} = d_s + 1$, and so must be the smallest d_{min} of the resulting working queue. Thus for all possibilities, the assumption is still true after the $(k+1)$st union.

Proof of Lemma 2:

According to the definition of "Rounds", Round$_i$ can not have virtual nodes with $d_{min} > S_{i+1}$. Also *Round*$_i$ will not end if it has more than one node with $d_{min} \leq S_{i+1}$, and with at least one of them has $d_{min} < S_{i+1}$, so it either ends with only one virtual node with $d_{min} < S_{i+1}$, or one or more virtual nodes with

the same $d_{min} = S_{i+1}$. From Lemma 1, there is at most one marked node, which is the only node that can have $d_{max} - d_{min} > 0$. Thus we complete the proof.

Let N_{ri} be the number of real nodes with size S_i, $N_{ri} = n_i$. Let N_{vi} be the number of virtual nodes after Round$_{i-1}$, before Round$_i$. Let $N_i = N_{ri} + N_{vi}$.

Claim 2 *if* $\Delta_{i-1} = 0$, $d_{i-1} = S_i$, $N_i = k2^{(S_{i+1}-S_i)}$, *for some* $k > 0$ *then* $\Delta_i = 0$ *and* $N_{v(i+1)} = k$.

This is easy to see. Since all nodes are paired up, our algorithm ensures we get zero density difference and $N_{v(i+1)} = k$ after Round$_i$.

Proof of Lemma 3:

After Round$_{i-1}$, before Round$_i$, by Lemma 2, there are two possibilities as below:

The first case is when $d_{i-1} < S_i$, according to Lemma 2, there is a single node $(d_{i-1}, d_{i-1} + \Delta_{i-1})$ then the first union in Round$_i$ is between this node and $R(S_i)$, generates a marked node: $V(d_{i-1} + 1, \max(d_{i-1} + \Delta_{i-1}, S_i) + 1)$, it has $|d_{min} - d_{max}| = \max(S_i - d_{i-1}, \Delta_{i-1})$. Also according to Lemma 1, the number of marked node can at most be one, we monitor this node within *Round$_i$* after the first union of this round, denote the marked node as $V(d, d + \Delta)(\Delta = \max(S_i - d_{i-1}, \Delta_{i-1}))$.
When $d_{i-1} < d \leq S_i$, the marked node could only involve in a union with $R(S_i) = R(d + r)$, $0 \leq r \leq S_i - d_{i-1} - 1 < \Delta$, get the union result as $V(d + 1, d + 1 + \Delta)$; when $d > S_i$, by Lemma 1 and because d_{min} can only increase by one after each union, the marked node can only involve in a union with $V(d)$ when we have even number of the smallest nodes or $V(d+1)$ when we have odd number of the smallest nodes, this also will not increase the density difference. So we proved density difference of the marked node will not increase after the first union during Round$_i$, thus $\Delta_i = \Delta = \max(S_i - d_{i-1}, \Delta_{i-1})$. this belongs to case a).

The second case is when $d_{i-1} = S_i$. Now including the first union in Round$_i$, all unions with the marked node during Round$_i$ is the second case of union described above, so if $\Delta_{i-1} \geq 1$, $\Delta_i = \Delta_{i-1} > S_i - d_{i-1} = 0$, it is also case a); if $\Delta_{i-1} = 0$, there is no marked nodes, we may get one node marked during Round$_i$ when we do not have even number of nodes with the smallest d_{min}, this is a result of a union like $(d + 1) \cup (d) = (d + 1, d + 2)$. If we have $N_i = k2^E, k > 0, E = S_{i+1} - S_i$ or $N_i = 2^E, 0 \leq E \leq S_{i+1} - S_i$, this type of union will never happen, we get $\Delta_i = 0$, this is also case a); otherwise, if we do not have such a N_i, there will be exactly one such union operation, raise the Δ from zero to 1 and get one node marked, after it during *Round$_i$*, by Lemma 1 and same argument we did above in the first part, this marked node $V(d, d + 1)$ could only involve in unions with a node like $V(d)$ or $V(d + 1)$, the density difference $\Delta = 1$ will not increase anymore during Round$_i$, so in this case we get $\Delta_i = 1$, this is case b).

Proof of Claim 1:

Because V_i can be viewed as being formed with all subtrees in V_{i-1} and all leaf nodes of size S_i, all subtrees of V_{i-1} lie at the same relative depth in their corresponding subtrees in V_i. So for the leaf nodes in V_{i-1} subtrees, their relative densities' range w.r.t. V_{i-1} will not change from their relative densities' range w.r.t. V_i. This is because their relative densities will increase by the same amount: $(l_{i,\max} - l_{i+1,\max})$. Since leaf nodes in V_{i-1} is also a subset of the leaf nodes in V_i, so $\Delta_{d_i} \geq \Delta_{d_{i-1}}$.

Also there is at least one leaf node with size of S_i lies in some $\nu_{i,j}$ with a relative depth of $(l_{i,\max} - l_{i+1,\max})$ and if we look at the leaf node in V_{i-1} that has the minimum relative density of $d_{min,i-1}$, its relative density in V_i will increase by $(l_{i,\max} - l_{i+1,\max})$, thus we know $\Delta_{d_i} \geq |S_i - d_{min,i-1}|$.

Claim 3 *If $\Delta_{i-1} = 0$, and $d_{i-1} = S_i$, and if in some regular tree, $\Delta_{d_i} = 0$, then $N_{vi} = k_{i-1}$, and from this we get $N_i = n_i + k_{i-1}$.*

Proof. First we know $\Delta_{d_i} = 0 \Rightarrow \Delta_{d_{i-1}} = \ldots = \Delta_{d_1} = 0$ (by (1)). $\Delta_{i-1} = 0 \Rightarrow \Delta_{i-2} = \ldots = \Delta_1 = 0$ and $d_{j-1} = S_j$ for $\forall j \leq i - 1$(by Corollary 1). Then with the help of Claim 2 we can prove it by induction on rounds.

For the base case in our regular tree, there are no V_0 subtrees, so $k_0 = 0$, thus $N_{v1} = 0 = k_0$, so the base case stands. Assume $N_{vj)} = k_{j-1}$ for $\forall j \leq t < i$. Since $\forall j \leq i$, $\Delta_{d_j} = 0$, then all leaf nodes with size S_t lie at the same depth in the regular tree, all leaf nodes with size S_{t+1} also lie at the same depth, and their depths difference has to be $S_{t+1} - S_t$, so $k_t 2^{(S_{t+1}-S_t)} = n_t + k_{t-1}$. Then $N_t = n_t + N_{v\,t} = n_t + k_{t-1} = k_t 2^{(S_{t+1}-S_t)}$, together with $\Delta_{t-1} = 0$, $d_{t-1} = S_t$, by Claim 2, we have $N_{v\,(t+1)} = k_t$ this is the induction round. Thus we proved $N_i = n_i + k_{i-1}$ inductively.

Proof of Corollary 2:

We prove this by contradiction. Assume after Round_i, $d_i < d_{min,i}$, then arbitrarily union the subtrees in V_i into one new tree, this tree is formed with all the nodes of size $S \leq S_i$, call the leaf nodes in this new tree's minimum density as d_{min}, then $d_{min} \geq d_{min,i} > d_i$, which means for all the nodes with $S \leq S_i$ our algorithm failed to maximize the minimum density, contradicted with Theorem 3(that our algo. Maximize Min density). So the assumption is wrong, we always have $d_i \geq d_{min,i}$.

Proof of Lemma 4:

We prove it by induction on the number of rounds. Assume that $\min(\Delta_{d_{i-1}}) \geq \Delta_{i-1}$ is correct for Round_{i-1}, we go through all the possible cases below to show either the next round or the round after next will also satisfy it, $\min(\Delta_{d_i}) \geq \Delta_i$ or $\min(\Delta_{d_{i+1}}) \geq \Delta_{i+1}$(if the next round is a b-round and is not the last round).

If the next round Round_i is not a $b-$Round, then according to Lemma 3, only Case a) is possible after Round_i, so we have $\Delta_i = \max(S_i - d_{i-1}, \Delta_{i-1})$. In case

of $\Delta_i = \Delta_{i-1}$, we know from (1) for any regular tree, $\Delta_{d_i} \geq \Delta_{d_{i-1}}$, also from the assumption $\min(\Delta_{d_{i-1}}) \geq \Delta_{i-1}$, so $\Delta_{d_i} \geq \Delta_{d_{i-1}} \geq \Delta_{i-1} = \Delta_i$. This holds for any regular tree, so we have $\min(\Delta_{d_i}) \geq \Delta_i$. In case of $\Delta_i = S_i - d_{i-1} > \Delta_{i-1}$, this means we have $d_{i-1} < S_i$, by Lemma 2 we have a single node after Round_{i-1} before Round_i . From (3), for any regular tree, $d_{i-1} \geq d_{min,i-1}$, also from (2) $\Delta_{d_i} \geq |S_i - d_{min,i-1}|$, so $\Delta_{d_i} \geq |S_i - d_{min,i-1}| \geq S_i - d_{i-1} = \Delta_i$, this holds for any regular tree, so $\min(\Delta_{d_i}) \geq \Delta_i$.

If the next round is a $b-$Round, then we know $\Delta_i = 1$ while $\Delta_{i-1} = 0$, the induction assumption will not be helpful here. There are two sub possibilities:

First, When $i < m$, we prove $\min(\Delta_{d_{i+1}}) \geq \Delta_{i+1}$, or the round after i—Round_{i+1} satisfies the induction condition. Since after Round_i we met case b), $\Delta_i = 1$ while $\Delta_{i-1} = 0$, so for Round_{i+1} it can only be case a). If $\Delta_{i+1} = S_{i+1} - d_i$, then with the same argument as we did earlier we can prove here $\min(\Delta_{d_{i+1}}) \geq \Delta_{i+1}$. If $\Delta_{i+1} = \Delta_i = 1$, we can prove the induction step by contradiction. Assume $\min(\Delta_{d_{i+1}}) \geq \Delta_{i+1}$ is not true, then there exist a regular tree, with $\Delta_{d_{i+1}} = 0$. Thus all leaf nodes of size S_{i+1} have to be in the same level, they all lie at the same depth as all V_i's subtrees' roots have in V_{i+1}'s subtrees. Also all leaf nodes with size S_i have to lie at a depth of $S_{i+1} - S_i$ in V_i's subtrees. Similarly all V_{i-1}'s subtrees' roots lie in the same depth in V_i's subtrees, which is also $S_{i+1} - S_i$, so $n_i + k_{i-1} = k_i 2^{(S_{i+1} - S_i)}$. At the same time, since Round_i is case b), so $d_{i-1} = S_i$, $\Delta_{i-1} = 0$, by Claim 3 in Appendix, $N_{vi} = k_{i-1}$, so $N_i = k_i 2^{(S_{i+1} - S_i)}$, since $\Delta_{i-1} = 0$, by Claim 2 in Appendix , we get $\Delta_i = 0$,contradicted with $\Delta_i = 1$! so, $\min(\Delta_{d_{i+1}}) \geq \Delta_{i+1}$ is true here.

Second, when $i = m$, round_i is a $b-$Round while it is the last round. We can still prove by contradiction. Assume $\min(\Delta_{d_i}) \geq \Delta_i$ is not true, then there exist a regular tree, with $\Delta_{d_i} = 0$, since $i = m$, V_m has just one tree, the regular tree itself, so because $\Delta_{d_m} = 0$, we know all leaf nodes of size S_m and the roots of the subtrees V_{m-1} lie at the same depth of the regular tree, and no leaf nodes above, so $n_m + k_{m-1} = 2^N$, N is a positive integer, also by Claim 3 in Appendix, we have $N_{vm} = k_{m-1}$, $N_m = 2^N$, then by Claim 2 in Appendix we know $\Delta_m = 0$, there is no way $\Delta_m = 1 (i = m)$, contradiction!

For the base round, we can think there is a $\Delta_0 = 0$, then either the first round is case a) condition of Lemma 3, or it is a $b-$Round, but no matter under which condition, all the argument above still holds. So we have the base case also stands, thus the proof is completed.

C Competitive Analysis

Proof of Theorem 4:

We build the adversary as: For the given set $\Re$, the adversary will serve in a $S_1, S_m, S_{m-1}, S_{m-2}, \ldots, S_2, S_1$ pattern.

The adversary will first serve a big wave of S_{min} processors in a number of $2^h + 1, h \gg \Delta S$. At this point, the online algorithm will construct a solution tree with these $2^h + 1$ processors. Let the depth of shallowest leaf node in this tree

be h_{min} and the depth of the deepest node be h_{max}. Thus the depths of all the leaf nodes satisfy $\sum_{h_{min} \le l \le h_{max}} \frac{1}{2^l} = 1$.

If $\Delta h = h_{max} - h_{min} > 3\Delta S$, the adversary will serve one processor of size $S_m, S_{m-1}, S_{m-2}, \ldots, S_2$ each, followed by S_1 in a number of $n_1 = (2^{\Delta S+1} - \sum_{i=2}^{m} 2^{S_i - S_1} - 1)$ then stop the input. For the online algorithm, there is at least one processor of the smallest size S_1 that has a depth of at most $h_{max} - (\Delta S + 1)$ in the final solution tree. The reason is that after serving one processor of size $S_m, S_{m-1}, S_{m-2}, \ldots, S_2$ each, the shallowest processor of S_1 could has a depth of at most $h_{min} + (\Delta S - 1)$. Then since $0 < n_1 < 2^{\Delta S}$, with the last wave of S_1 input, the shallowest S_1 size processor's depth can at most be $h_{min} + (\Delta S - 1) + \Delta S$. While the deepest S_1 size processor is still at a depth of h_{max}. Since $h_{max} - [h_{min} + (\Delta S - 1) + \Delta S] > \Delta S + 1$, the final solution tree has a density difference of at least $\Delta S + 1$, the load density ratio is at least $2^{\Delta S+1}$. For the offline algorithm, let $h_1 = \lfloor log_2(2^h / 2^{\Delta S+1} + 1) \rfloor$, let $h_2 = h_1 + \Delta S + 1$. After the first wave of $2^h + 1$ size S_1 input, the offline algorithm we choose will keep one S_1 at a depth of h_1, and the rest 2^h processors at a depth of either h_2 or $h_2 + 1$. Since $\sum_{i=2}^{m} 2^{S_i - S_1} + n_1 + 1 = 2^{\Delta S+1} = 2^{S_m - S_1 + 1} \Rightarrow \sum_{i=2}^{m} \frac{1}{2^{S_m - S_i + 1}} + \frac{n_1 + 1}{2^{\Delta S+1}} = 1$. This means the single S_1 size processor left behind can be split to a depth of h_2, so do the last wave of the n_1 S_1 size processors. And for any of the single processor S_i, $2 \le i \le m$, it will be split to a depth of $h_2 - (S_i - S_1)$. Thus the final solution tree can have a load density ratio of at most 2. So in this case the ratio between the online and offline cost is $2^{\Delta S+1} \div 2 = 2^{\Delta S}$.

Otherwise if $\Delta h \le 3\Delta S$, the adversary will serve $2^{h-\Delta S} - 1$ S_m size processors, followed by one processor for each size of $S_{m-1}, S_{m-2}, \ldots, S_3, S_2$. Then finally the adversary serves $n_2 = 2^{\Delta S} - 1 - \sum_{i=2}^{m-1} 2^{S_i - S_1}$ S_1 size processors, then stop. The optimal offline algorithm for this input will construct a tree with one S_1 at depth 1 and the rest with a depth of h for the first wave of S_1 input. Then use the second S_m wave to split the shallow S_1 processor and the joining S_m processors to a depth of $h - \Delta S$. Finally with the third wave of one processor each for $S_{m-1}, S_{m-2}, \ldots, S_3, S_2$ and n_2 S_1 processors continue to split the single S_1 left behind and the last wave of S_1 processors to the depth of h. And each of the S_i $(2 \le i \le m-1)$ size processor will be split to a depth of $h - (S_i - S_1)$ thus the final density difference for the optimal offline algorithm for this input is zero, the load density ratio is 1. We show the online algorithm's cost has to be $2^{\Delta S}$ below.

Given that the online algorithm's solution tree for the first wave of input satisfies $\Delta h \le 3\Delta S$, it is easy to see that from the second wave, the best the online algorithm could do is to use each arriving S_m processors to split a current shallowest node, for nodes of the same depths, select a S_1 size instead of a S_m size to split unless there is no S_1 size processors for the shallowest depth. Then at the end of the S_m input, denote the deepest S_m processor's depth as h_m, there will be at least $(2^{h-\Delta S} - 1)/2^{h_m - h_{min}} \ge 2^{h-\Delta S-1}/2^{\Delta h} \ge 2^{h-4\Delta S-1}$ S_1 size processors with a depth less than or equal to h_m. Since $h \gg \Delta S$, we have $2^{h-4\Delta S-1} > 2^{\Delta S}$. Since the total number of the rest input is only $m - 1 + n_2 < 2^{\Delta S}$, we know in the final solution tree not all the S_1 processors that are not deeper than h_m can

be split to a depth bigger than h_m. So the resulting solution tree will have the smallest density as at most $S_1 + h_m$, and the largest density as at least $S_m + h_m$, or a load density ratio of at least $2^{S_m - S_1} = 2^{\Delta S}$. The ratio between the online and offline cost then is $2^{\Delta S} \div 1 = 2^{\Delta S}$.

Since for any online algorithm, it will be either of the above two cases, we have proved that the ratio between any online algorithm's load density ratio cost and optimal offline algorithm's load density ratio in the worst case will be at least $2^{\Delta S}$. In other words, $2^{\Delta S}$ is a lower bound for the competitive ratio of any online algorithm.

Proof of Theorem 5:

We describe an algorithm with a cost of at most $2^{\Delta S+1}$ for all possible inputs.

Conservative Algorithm(**ConA**): The first processor will be placed at the root of the tree. After this, for any arriving processor the algorithm will select a shallowest node to split, and within the shallowest nodes, select the smallest density(or smallest size) processor to split.

It is easy to see that this algorithm will give a solution tree with a depth difference of at most 1, thus the density difference is at most $\Delta S + 1$ and the load density ratio is at most $2^{\Delta S+1}$.

Since the density difference for any algorithm is nonnegative, including the optimal offline algorithm, the load density ratio is then at least 1 for any algorithm. Thus we know the competitive ratio for ConA is at most $2^{\Delta S+1}$.