Boosting the Stragglers:

Runtime Redistribution in a Distributed System

Andy Evenson

Abstract

Many distributed algorithms are limited by the performance of the slowest machine in the system. For instance, if an algorithm requires all processes to reach a barrier before it can continue, then the progress of the whole is determined by the last process to reach that barrier. One possible solution to this problem would be to reduce the load on the slower machines by giving some of their work to the faster ones. In practise, however, this is very difficult to implement. Deciding how to distribute the load often requires global knowledge, direct supervision by an omniscient machine, and heavy calculation. Furthermore, if we do decide to take a job from a slow machine and give it to a fast machine, we must first guarantee that the latter machine has all of the necessary resources (Data, information, communication ability, etc.) to complete the job.

This paper examines a specific setting where workload redistribution is desirable, describes a simple solution, and analyses its effectiveness. The setting is a large parallel database in which 100 client machines work concurrently to read a file that is stored across 100 servers. The solution requires only local communication (each server must communicate with two clients, and each client with two servers), and only very minor calculations. It changes dynamically with time and can be shown to converge on the best possible solution. We present it here in several variations, and discuss the strengths and weaknesses of each.

Introduction

In a distributed system, a single task can be broken down into several pieces which are then performed simultaneously. This allows resources (such as disk space) to be pooled together, and it can dramatically reduce response time. For instance, a distributed program could use a file much bigger than any single machine could store, because it can spread the file across all the machines in the system. And though ten machines can do ten different jobs simultaneously, the average response time will be much lower if they all work together on a single job before moving on to the next.

There are, however, many difficulties in coordinating the efforts of more than one machine. Not least of these difficulties is the fact that faster machines often end up waiting for slower machines. By way of analogy, consider the teacher evaluation process at UW-Madison: Teachers hand out evaluation forms in class but are required to leave the room while the forms are being filled out. So some student must volunteer to collect the forms in the teacher's absence. But volunteers are scarce, because every student thinks to herself, "If I volunteer to collect these forms, then I'll have to stick around until the last student has finished writing flowery comments -- and who knows how long that could take!" Likewise, most distributed programs contain "barrier points", where all machines must stop and wait until the last one catches up.

Specifically, I would like to consider a large parallel database, in which 100 client machines perform searches based on information provided by 100 server machines. The data is stored in a huge file that is spread across all of the servers and each data block is stored exactly twice (ie mirrored once for reliability) on different servers (say Servers i and i-1). In a typical operation, each client will read the data stored on one server, process it in some way, and send the results on to a single designated machine for correlation. Thus client 1 will read from server 1, client 2 from server 2, and so on. Obviously the correlation step cannot happen until all of the clients are finished reading and processing, so we would like to somehow ensure that they finish simultaneously.

It turns out that the real limiting factor in performance is not the network speed nor the client processing speed, but the bandwidth at which the servers can supply the data to the clients. If even one server has a slow disk or is bogged down by outside requests and takes extra time to supply his client with data, the entire operation is slowed down by that same amount of time. The rest of this paper will focus on two things: first, describing several variations of a simple, easily implemented solution to this problem; and second, analysing those solutions in terms of their robustness, effectiveness, and provability.

(Maple Definitions)

1. Description of the Solution

The key to this solution (originally suggested by Remzi Arpaci-Dusseau), is the fact that every data block can be obtained from two possible sources. We have a very large file spread evenly across the storage disks of 100 server machines, and for reliability purposes every data block appears twice. Thus Server i has 2% of this large file stored on its disk: 1% that was designated to Server i, and 1% that was designated to Server i+1 but is mirrored on Server i. That means that if say Server 8 suddenly slows way down, Server 7 could step in and help out by supplying some of the data that Server 8 was supposed to supply.

Let's set up the situation a little more carefully with an example: Suppose our parallel database contains sales records for a large company like Wal-Mart. A request comes in from a user who wants to know how many bags of cheddar pretzels were purchased at Wal-Mart on Tuesday. We'll assume here that the file TuesdayPurchases.dat contains a record of every purchase made on Tuesday at any Wal-Mart store in the country, and that this file has been broken into 100 pieces and stored as above.

To handle this request, each of the client machines will send a request to its respective server (so Client 3 sends a request to Server 3) for the part of TuesdayPurchases.dat which is stored on that server (1% of the total). The servers supply the information as fast as they can, and as the clients receive it, they count the number of records which indicate a cheddar pretzel purchase. When each client finishes this process, it sends its total count to a predesignated client which then tally's all of the results. Clearly, the final result is obtained only after ALL of the clients are finished.

THE PROBLEM:

Now suppose that one server (say Server 15) is really slow (it has a stomache ache from all those pretzels). Then Client 15 will take a longer time to finish (since it gets its data less quickly than the others do.) In this case, it would make sense for Server 14 to help out.
Server 14 has a back-up copy of Server 15's data, so Server 14 is qualified to service Client 15. Thus it can spend part of its time sending data to Client 15.

This means that Client 14 receives data at a reduced rate, but it increases the rate (or bandwidth) at which Client 15 receives data. By sharing some of Server 15's load, Server 14 can average out the bandwidths (C_{14} and C_{15}) at which Clients 14 and 15 receive data. Thus the calculation is no longer constrained by Server 15's rate, but is now constrained by the average of the two rates of Servers 14 and 15 (which is higher than Server 15's rate.)

Note that server 13 could now notice that Client 14 has slowed down, and it could help Client 14 in the same way that Server 14 helped Client 15. This would give Server 14 a little more leeway to help Client 15 even more; and we get a chain reaction. If we had one central machine that communicated with all of the others, we could use it to calculate exactly how much bandwidth each server (Server i) should 'give' to Client i+1 for optimal load distribution. However, this requires a lot of overhead, a central communication system, and fairly involved calculations. Instead, we will implement this in a much simpler way requiring only local communication (Server i is connected only to Clients i and i+1).

IMPLEMENTATION OF THE SOLUTION

We will give all of the decision making to the servers. Each client (Client i) sends requests simultaneously to servers i and i-1 (the two servers that have the right data for Client i). So each server receives requests from two Clients and decides what priority to give each (ie decides how to split its bandwidth between them). To make this decision, the server relies on information that the clients pass along with their requests. This information must be some measure of the progress that a client is making. That way, the server can divide its bandwidth in such a way as to even out the progress of the two clients it can service.

MEASURING PROGRESS

So, the solution we have just described leaves two decisions open:

- How do we want to measure progress?

- Given a progress report, how does the server make its decision?

Depending on our answer to these two questions, we can get several variations with different strengths and weeknesses.

2. Progress vs. Rate of Progress

We can measure the progress that the clients are making in two Fundamentally different ways:

- First, we can measure **RATE OF PROGRESS** by looking at the bandwidth C_i at which each client is receiving data. So we can look at the overall rate of progress as a vector containing all of the values of C_i :

```
> Rate := matrix(5,1,
>    [C[1],C[2],C[3],`...`,C[100]]);
```

$$Rate := \begin{bmatrix} C_1 \\ C_2 \\ C_3 \\ ... \\ C_{100} \end{bmatrix}$$

- Second, we can measure **ACTUAL PROGRESS** by looking at the number of data blocks D_i that each client has yet to read:

```
> Progress := matrix(5,1,
>     [D[1],D[2],D[3],`...`,D[100]]);
```

$$Progress := \begin{bmatrix} D_1 \\ D_2 \\ D_3 \\ \ldots \\ D_{100} \end{bmatrix}$$

[>

Both methods have pros and cons. Actual progress is a more accurate measure of what still needs to be done, but we can only use it if we know how many total bytes need to be read. That means we must know:

- the size of the file being read
- how much of it is stored on each server
- the entire file will be read

If any of these three conditions fails, we cannot measure actual progress and we must instead use rate of progress as our unit of measurement. For instance, if a search engine is designed to halt and return when fifty matches have been found, then it is impossible to know how much data will be read before the goal is reached. In that case, the best we can do is try to distribute the server bandwidths in such a way that the clients receive data at the same rate.

In the remainder of this paper, we will show that with successive iterations the rate of progress method can be made to converge quickly on the actual progress method. Thus, in cases where we do not have enough information to measure actual progress, it is good enough (ie it still gives you the same order of magnitude of performance benefits) to measure rate of progress.

3.Markov Chains:

2 basic assumptions:

- Client i can only speak with servers i, i-1.
- At some starting point, each server splits its bandwidth exactly evenly between its 2 clients.

Let's look a little more closely at the types of algorithms we can use based on the **rate of progress** paradigm. If we consider the vector C defined above, we can analize how C changes with time. First of all, we'll assume that at some starting point all the servers begin by splitting their available bandwidth exactly evenly between the two clients they serve. This not only starts things out more efficiently, but it allows servers to move their bandwidth either direction as they see fit. (If a server began with all of it's bandwidth directed at one client, then it could only move bandwidth one direction.)

Each time a server receives new information from the two clients it can service, that server can reevaluate how it divides its bandwidth between the two clients. The server can decide to try to exactly even out the two clients' progress, or it can be more conservative and only change by a fraction of that amount. The reason we might be conservative is that the servers on either side are doing the same thing, so there is a potential of overkill. That is, if Client 2 is making slow progress, then Servers 1 and 2 will both come to the rescue. Server 1 will take bandwidth from Client 1 to give to Client 2, and Server 2 will take bandwidth from Client 3 to give to Client 2. The danger here is that Client 2 might receive twice as much extra bandwidth as it needed.

To analyze this, we can look at what happens to the vector C at each step of this process. Each C_i will be replaced by the following formula:

```
> Next_C[i] := (C[i+1] + (k-2)*C[i] + C[i-1])/k;
```

$$Next\_C_i := \frac{C_{i+1} + (k-2)\, C_i + C_{i-1}}{k}$$

where k determines how "conservatively" the servers try to even out the client bandwidths. Note that we must have $2 \leq k$, and the value $k = 2$ corresponds to the most unconservative state (where the servers each try to singlehandedly even out the bandwidths of their two clients.)

We can represent this change to the vector C as a multiplication by a **"transition matrix."** And indeed the step by step changes in C represent a stochastic process (specifically a Markov chain.)

A Few Examples of Transition Matrices:

The entries in these matrices tell us how evenly the server bandwidths have been distributed among the clients. For instance, in the second column, the ones in the first and third place mean that after one step, the bandwidth received by Client 2 is equal to the average of the bandwidths of Servers 1 and 2.

Matrix 1: Servers try to even the bandwidths out. k = 2.

```
> T := Transition (10, 1, 0);
```

$$T := \begin{bmatrix} 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \end{bmatrix}$$

If you are running this in Maple, you can execute the commands below to see the effects of multiple steps, which tells us what happens over time. Notice that the matrix never really stabilizes.

Maple Users Only:

Matrix 2: A bit more conservative. k = 3.

```
> T:=Transition (10,1,1);
```

$$T := \begin{bmatrix} 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \end{bmatrix}$$

Matrix 3: A happy medium: k = 2.5.

```
> T:=Transition (10,2,1);
```

$$T := \begin{bmatrix} 1 & 2 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 2 \\ 2 & 1 & 2 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 2 & 1 & 2 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 2 & 1 & 2 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 2 & 1 & 2 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 2 & 1 & 2 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 2 & 1 & 2 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 2 & 1 & 2 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 2 & 1 & 2 \\ 2 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 2 & 1 \end{bmatrix}$$

<u>Matrix 4: The general case.</u>

```
> T:=Transition (10,P,Q);
```

$$T := \begin{bmatrix} Q & P & 0 & 0 & 0 & 0 & 0 & 0 & 0 & P \\ P & Q & P & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & P & Q & P & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & P & Q & P & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & P & Q & P & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & P & Q & P & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & P & Q & P & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & P & Q & P & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & P & Q & P \\ P & 0 & 0 & 0 & 0 & 0 & 0 & 0 & P & Q \end{bmatrix}$$

<u>Matrix 5: Suppose each data block is mirrored on TWO extra servers, rather than just one. Then, each client can get data from three different servers, and each of those servers can have two more clients. So at each step, a client's perceived bandwidth C(i) is affected by those of six other clients! We can rig things so that we get the following matrix:</u>

```
> size := 10:
> B := matrix( size,size):
> for i from 1 by 1 to size do for j from 1 by 1 to size do
  B[i,j] := 0 od od:
> for i from 1 by 1 to size do
> for t from -3 by 1 to 3 do
> B[i,1+((i-t-1) mod size)] := 1:
> od od:
> print(B);
```

$$\begin{bmatrix} 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \end{bmatrix}$$

[>

4.What Markov Chains Tell us:

We can use Markov Theory to show that our methods are robust. In other words, the above algorithms really do converge on the same behavior as if we had been able to measure actual progress. The main result of interest is the following theorem, which can be found in any text on probability:

Theorem: Let T be an nxn transition matrix in a Markov Process. Suppose there is an integer m such that every entry in T^m is strictly positive. Then there is exactly one vector $C = (C_1, C_2, ..., C_n)$ such that $C\,T = C$, and every row of T^l converges to C (as l increases).

We can also find an upper bound on the difference between an entry in a row of T^l and an entry of C. This upper bound is equal to $(1 - n\,d)^s$, where d is the smallest element of T^m, and s is the integer part of $\frac{l}{m}$.

In English: The vector C is equal to the best case scenario of even distribution, and we approach this scenario geometrically with time.

It turns out that these particular matrices converge much faster than the general case upper bound matrix given above. And we can show this simply by graphing the data, but the theorem above gives us a way of saying that we will ALWAYS tend toward the right solution.

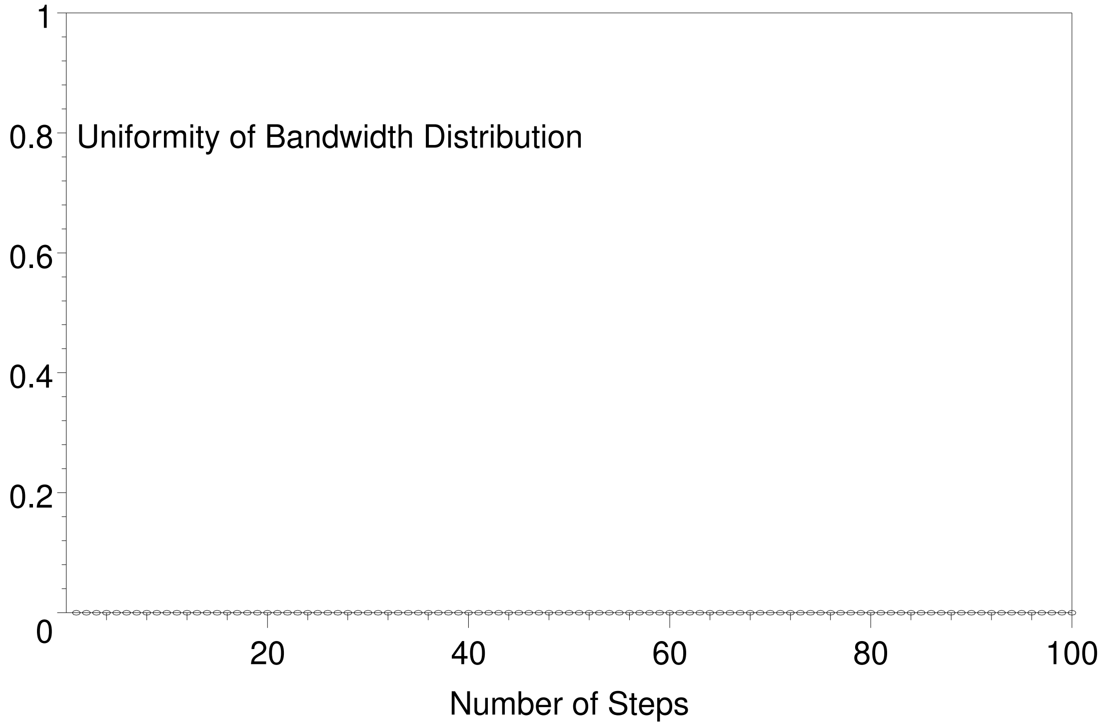
5.Convergence:

Here are a few graphs to show how quickly the rate of progress paradigm converges to the actual progress paradigm for various values of k.

```
> MinEntryData := proc(k, n :: integer, size::integer)
> local T,S,R,i,j;
> T := Transition (size, 1/k, (k-2)/k);
> S := T;
> for j from 1 by 1 to n do
> R[j] := size * min(seq(S[i,1], i=1..size));
> S := multiply (S,T);
> od:
> [seq([j,R[j]],j=1..n)];
> end:
```

```
> Data := MinEntryData (2, 100, 20):
> graph := listplot (Data, axes = box, font =
  [HELVETICA,BOLDOBLIQUE,12], axesfont =
  [HELVETICA,BOLDOBLIQUE,12], labelfont =
  [HELVETICA,BOLDOBLIQUE,12], labels = ["Number of Steps",""],
  style = point, symbol = circle, title = "Figure 1: Convergence
  for k = 2"):
> labelplot := textplot ([1,.8,"Uniformity of Bandwidth
  Distribution"],align = RIGHT,font =
  [HELVETICA,BOLDOBLIQUE,12]):
> display([labelplot,graph], view = [0..100,0..1]);
```

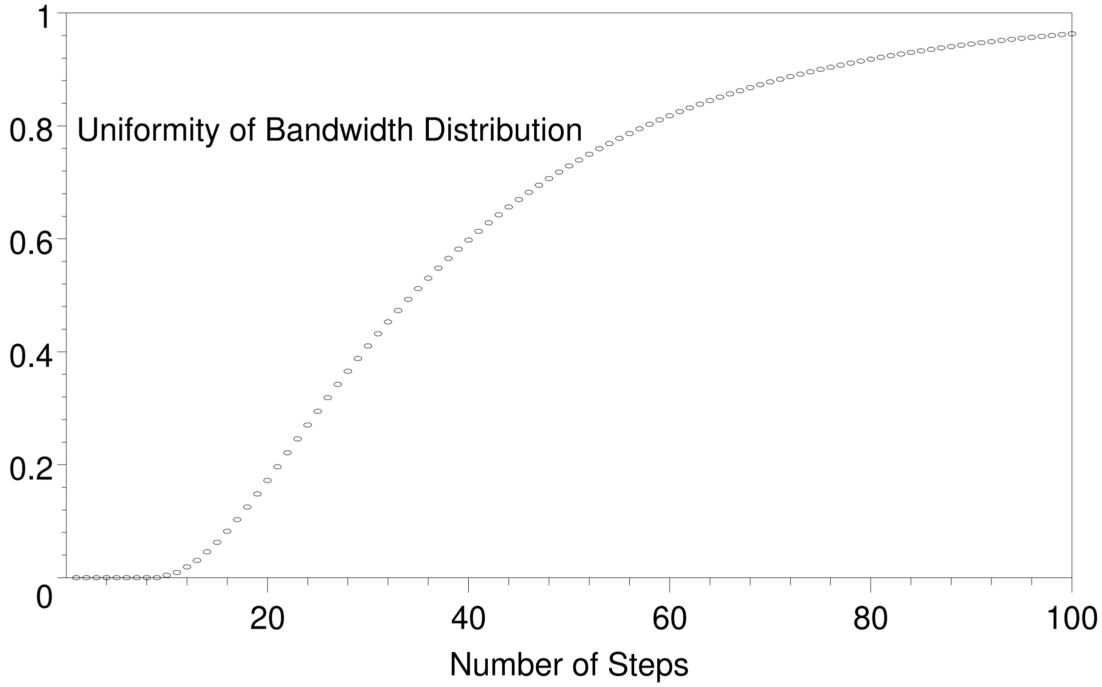
Figure 1: Convergence for k = 2



Notice that for k=2, we do not get any convergence: all of the values in the graph are zero. This is because the transition matrix has zero entries at every step.

```
> Data := MinEntryData (2.5, 100, 20):
> graph := listplot (Data, axes = box, font =
  [HELVETICA,BOLDOBLIQUE,12], axesfont =
  [HELVETICA,BOLDOBLIQUE,12], labelfont =
  [HELVETICA,BOLDOBLIQUE,12], labels = ["Number of Steps",""],
  style = point, symbol = circle, title = "Figure 1: Convergence
  for k = 2.5"):
> labelplot := textplot ([1,.8,"Uniformity of Bandwidth
  Distribution"],align = RIGHT,font =
  [HELVETICA,BOLDOBLIQUE,12]):
> display([labelplot,graph], view = [0..100,0..1]);
```

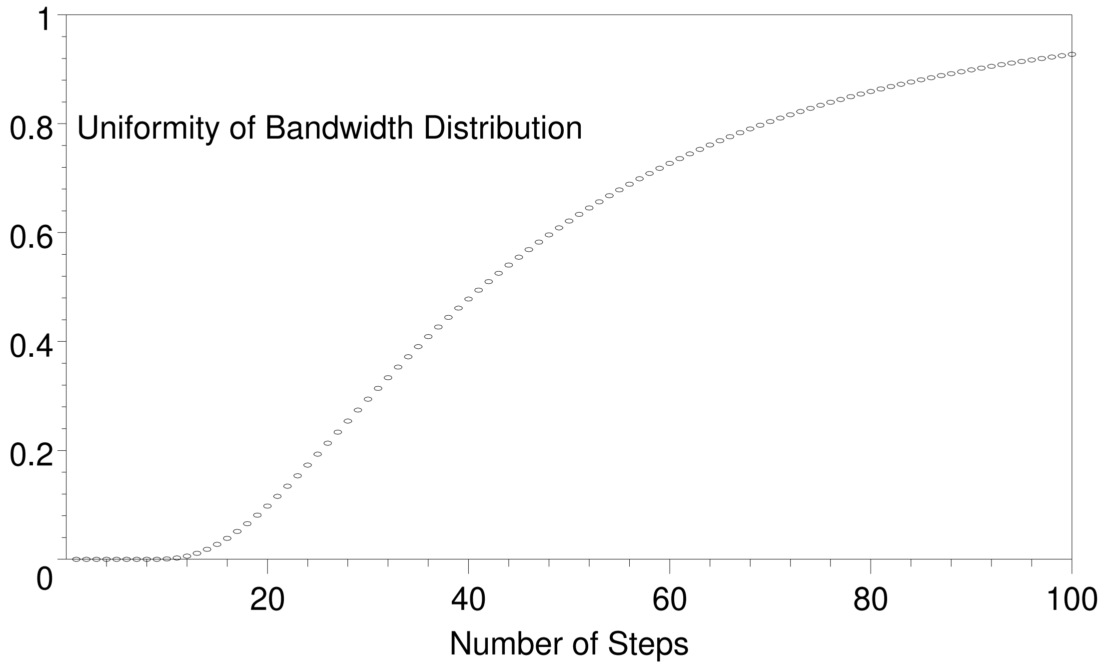
Figure 1: Convergence for k = 2.5



For k = 2.5 we get our fastest rate of convergence.

```
> Data := MinEntryData (3, 100, 20):
> graph := listplot (Data, axes = box, font =
  [HELVETICA,BOLDOBLIQUE,12], axesfont =
  [HELVETICA,BOLDOBLIQUE,12], labelfont =
  [HELVETICA,BOLDOBLIQUE,12], labels = ["Number of Steps",""],
  style = point, symbol = circle, title = "Figure 1: Convergence
  for k = 3"):
> labelplot := textplot ([1,.8,"Uniformity of Bandwidth
  Distribution"],align = RIGHT,font =
  [HELVETICA,BOLDOBLIQUE,12]):
> display([labelplot,graph], view = [0..100,0..1]);
```

Figure 1: Convergence for k = 3



>

6.Conclusion:

Based on the analysis described above, as well as much more which would not fit within this paper, we can achieve good results in redistributing a distributed load, using only very simple algorithms. We need only require each server machine to communicate with two clients, and the only information that must be collected is (readily apparent) information about the bandwidth at which a client receives data. The algorithms that we derive using this simple paradigm can be made to converge to nearly the same performance that we would see if we had information about the ***actual*** progress of each client rather than just that client's ***rate*** of progress.