Analyzing Isoefficiency of Partition Local Algorithms

Arjan Lamers[1,2] and Marko van Eekelen[2,3]

[1] First8 BV, Nijmegen, The Netherlands
[2] Open University of the Netherlands
[3] Radboud University Nijmegen, The Netherlands
a.lamers@first8.nl, marko@cs.ru.nl

Abstract. For network applications scalable performance is paramount. NoSQL database systems provide us with a tool to achieve scalable systems. In practice, scalability of performance can depend heavily upon the way of partitioning of the data models. To help making these engineering considerations we develop a first step towards a performance model that will incorporate the effects of partitioning. With this basic model we study the scalability properties of algorithms on sets of processors, each with its own statically assigned local part of a given data set. Within this model we calculate the isoefficiency of a special class of algorithms, *partition local* algorithms (that can run fully within a single partition). We study different algorithm complexity classes and show the validity of our results using a simulator.

1 Introduction

NoSQL databases are getting a lot of attention in the software industry. In NoSQL databases the strict ACID (Atomic, Consistent, Isolated and Durable) properties are no longer available for reasons of scalability. If you need a NoSQL database in your design, chances are you have lots of data and require quick response times. While they are subject to the CAP Theorem [Gilbert and Lynch, 2002]. This theorem states that from the three properties *consistency*, *availability* and *partitionability*, only two can be guaranteed at the same time in a distributed system. As a consequence (in most cases) consistency is sacrificed but you gain a scalable engineering option to handle large amounts of data in the form of BASE systems (Basically Available, Soft state, Eventually consistent).

Most of these BASE databases treat data as anonymous blocks, randomly distributing them around the grid. Within such a block, normally consistency is guaranteed again. Sometimes a block contains a simple value but often a fully structured JSON or XML document is used. Lately,

some NoSQL databases begin to offer features enabling even more control over the location of your data. It is left up to the developer what kind of data to put in such an atomic block, balancing the granularity of the data (and thus scalability) against the consistency requirements. Features like *data affinity* in Oracle's Coherence or Hazelcast, *routing* in Elastic Search and *grouping* in JBoss' Infinispan enable you to make sure that specific blocks are mapped to the same part of the grid. Other products, like Gigaspaces, are actually requiring this kind of explicit mapping for your full data model. If an operation has all its data in the same part of the grid, communication overhead will be reduced and thus response times will be better.

While work is being done on standardizing a sql-like language for NoSQL databases [Meijer and Bierman, 2011] in the form of e.g. UnQL and CQL, designing for these systems sometimes still feels more like an art than science. Deciding what data to group together impacts the performance of operations as well as what level of consistency can be guaranteed as is shown with a motivational example in Sect. 2. Important concepts in this area are introduced in Sect. 3.

So, how do you decide what to data to group together and are there effects on scalability? In a first step towards this goal, we examine in this paper how scalability relates to partition local algorithms (i.e. algorithms that can run fully within a single partition). We do that by mathematically analyzing the asymptotic isoefficiency of partition local algorithm in Sect. 4. We validate each result using a simulation. Finally, we discuss future work, related work and conclusions in Sect.s 5, 6 and 7.

2 Motivational example

As an example we will use a trivial implementation of a classified website. Usage of the website includes users being able to create advertisements (*ads*) and place them in categories. The website has a home page with the top ads per category. Visitors are able to shows ads per category, search for an ad and view the details of a specific ad. The data model and assumed sizes of the site are described in Fig. 1.

To partition this data model there are several possibilities. We distribute the tupels using a simple modulo function or some other consistent hashing on the tuple's primary key with the total number of partitions. This will evenly distribute the data: if we assume 100 partitions,

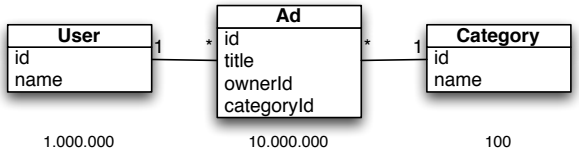

Fig. 1. UML class diagram with data sizes

this will mean that each partition will contain 100.000 ads, 10.000 users and a single category. Using a static partitioning scheme one cannot scale endlessly. When the number of processors increases, the distribution will become less even. When it reaches the size of the data set, the maximum scalability is achieved. With 10.000.000 processors we have one ad for each processors and no more ads to distribute. If you want to efficiently use more processors you need to add more data.

When the site wants to retrieve information of an ad given an adId, e.g. for the 'view ad' page, we can assume that the context of the 'view ad' operation knows the primary key of the ad. Thus we can easily determine which partition holds the correct data. Assuming all ads have an equal chance to be viewed, each 'view ad' operation will be directed to a single machine which can operate independently of others.

On the other hand, displaying the user's contact information could be more expensive: the user row associated with the ad might reside in a different partition. But, since the user is also partitioned on his primary key it is trivial to determine on which partition it is and retrieve the needed information. In the best case this is the local partition, in the worst case a single other partition needs to act. When creating or editing an ad or a user we touch the topic of *referential integrity*: the references to data must remain consistent. If referential integrity is needed, each update will need to check all relations with other tables. If any rows are residing in a different partition some form of transactions are required. Since it is inter-partition we have to take the CAP theorem into account and let go of some security. For example, using a two-phase commit (which can result in inconsistency) or paxos commits (potential unavailability).

For rendering the home page it gets more complicated. There are a number of ways of retrieving the top 10 ads per category but the data is scattered all over the partitions. The most obvious way of getting the ads is first to map (ask) to all partitions their list of categories and reduce

that to a single list. Secondly, broadcasting that list of categories we can ask for each partition their top 10 of ads for each category and reduce that back to 10 ads per category. It is not trivial to see if having the data scattered is increasing or decreasing performance.

The problem of retrieving the user's data when viewing an ad could be solved by partitioning an ad on the userId instead of the adId. The operations can than be routed on userId instead of adId. But that would mean that actually reaching the ad data given an adId is non-trivial anymore: no longer is it possible to determine in which partition the ad is residing if the userId is not known.

Another suggestion could be to partition ads on their categoryId. The same issue as with partitioning on the userId will appear but at least the home page can now be rendered partition-local: we can now broadcast to all partitions to give their top 10 ads per category. We know for sure that all ads in a category will be in the same partition so there is no need to broadcast a second time. We might even be able to cache them locally. Depending on the usage statistics this partitioning might actually be better for overall performance.

But now we have introduced two new bottlenecks. It is safe to assume that not all categories will have the same amount of ads. This would mean that the amount of ads will be significantly different between machines and thus we might reach the upper memory limit of a processor handling the biggest partitions. Also, since ad's are now tied to categories, we can also no longer scale to the number of ads in the system but instead we hit a limit already when the number of processors reached the number of categories, significantly lower than the number of ads.

Under certain assumptions other partitioning possibilities arise. If we assume that categories almost never change we could also make a local cache for them. Then, there is no need for partitioning on categories: all categories are now local. When categories do change we have to flush out all caches on all partitions but since that is rarely done it might be a good trade-off.

If we always know the userId when we have an adId, we could reconsider partitioning the ads on the userId. Alternatively we could choose to change the primary key of the ad to adId+userId, thereby forcing the application to always provide both. The tradeoff in a more skewed data distribution seems less worse than with categories: the cardinality of the users is closer to that of the ads than the categories were.

Summarizing, it is already apparent in this simple case that deciding on partitioning is not trivial. References between tables, use cases and cardinality of the types are factors to be considered even in a trivial example as given in this section.

3 Basic Concepts

Scalability is a property of a system so that it can be gracefully enlarged, in the sense that it can handle more data or more throughput. As Amdahl's law states, as soon as a program has a sequential part, adding more processors will have less and less impact. In other words, the efficiency of the systems becomes less. To keep the efficiency at a stable level, the problem size also has to increase. *Isoefficiency* [Grama et al., 1993] is a way of describing the increase in problem size needed to keep the efficiency of a parallel system constant while increasing the number of processors. In this paper we use isoefficiency as a measure for scalability.

Partitioning is a technique where data is divided into subsets. This is obviously required when the data set is simply too large for a single processor. Another incentive might be to create more locality of data access. A subset of the data is called a *partition*. In the context of a relational database this would mean that different rows or tuples in a given table will be distributed over several partitions. Each partition is assumed to have its own dedicated processor, thus making the data in the partition *private* to its processor.

Data which is available only to a single processor and is stored closely to that processor is defined as being *local*. For sake of simplicity in this paper there are no degrees of locality. If no other processor can access that data, that data is called *private* to that processor. Data local to another processor is defined as *remote* data. Data shared between processors is called *shared*. Shared data of course has to be protected for mutual exclusion problems. Local data does not require communication over unreliable connections or locking mechanisms to prevent mutual exclusion problems. Thus local data is expected to be faster then remote or shared data. In this paper we consider communication channels to be reliable and calculate a fixed overhead for it. Shared data is not considered: it is assumed that all data is partitioned and thus local.

A *mapping function* p uniquely maps a row (tuple) in the table to a partition name, given some *partition configuration*. If two different types

of tuples are mapped to the same partition name, they are considered to be in the same partition. In the most trivial case the partition configuration is the number of partitions and the partition function is a hash or modulo operation on some id. That number then points to a specific processor on which the partition resides. A mapping function which does not require communication to resolve the partition for a row, is called a *local mapping function*.

More complex configurations also occur; the distribution of tuples may be managed by a server and thus require a lookup in an index. A mapping function which requires a remote lookup to resolve the partition is a *remote* mapping function.

If an algorithm has all the data it needs inside a single partition, we call it a *partition local algorithm*.

A *routing function* takes a request and determines the best possible partition (if any) to execute that request. If the algorithm for a request is partition-local, the routing function should yield the partition containing the data for that operation.

If the data set allows to distribute the data evenly among any number of partitions in such a way that the chance of an operation needing a specific partition is equal for all partitions and each element occurs only in a single data set, that data set is *partitionable*.

Used conventions

cost function	c
the total dataset	D
partition	$p_{N,i}$
total amount of data	$\|D\|$
set of all partitions	P
nr of processors	N
routing function	r
constant cost of routing	c_r
all potential operations	$\mathbb{W}$
work for a single sequential operation	w
parallel individual workload	w^p
total number of instructions	W
maximum concurrency	$C(W)$
sequential time	T_1
parallel time	T_P
speedup	S_P
efficiency	E_P
total parallel cost	C_P
overhead	T_O

4 Mathematical Model

In this section we define a mathematical model. With the model we calculate the isoefficiency of different classes of algorithms. For each class we validate the result using a simulator.

4.1 Definitions

We define N as the number of servers. We assume that all partitions P are computationally independent and equal, e.g. each partition has its own server with the same specifications as the other servers (so, $N = |P|$ and server i manages partition i).

Large scale websites need to be able to run large amounts of similar instructions in parallel; each visitor of the website has its own operation w. W is the total workload, the number of operations the system should handle in a given period.

We express the scalability in terms of isoefficiency using the following common definitions: speedup ($S(N) = \frac{T_1}{T_N}$), efficiency ($E(N) = \frac{S_N}{N} = \frac{W}{C_N}$) and the total parallel cost ($C_N = N \cdot T_N = \frac{T_1}{E_N}$). The total overhead can thus be defined as $T_O = C_N - W$. As usual, the isoefficiency constant K is defined as $K = \frac{E}{1-E}$.

Given a set D of tuples:

$$D = \{t_0, t_1, t_2, ...\}$$

We define a *mapping function* which, given some configuration, maps each tuple to a *partition number*. For simplicity we assume the configuration is a natural number, e.g. the number of partitions.

$$m : \mathbb{N} \times D \rightarrow \mathbb{N}$$

A *partition* $p_{N,i}$ is now defined as the set of all tuples mapped to i:

$$p_{N,i} = \{t \in D | m(N, t) = i\}$$

The *partitioning* is defined as the collection of partitions:

$$P_N = \cup_{i=0}^{N-1} p_{N,i}$$

A *routing function* uniquely maps an operation to the number of the partition which is best equipped to handle that operation. Assuming that W is the set of all operations for all visitors, we can define r as:

$$r : \mathbb{N} \times \mathbb{W} \rightarrow \mathbb{N}$$

Via the mapping function, all tuples in D are uniquely assigned to a single partition:

$$\forall_{t \in D} \exists_{!i} \, [t \in p_{N,i} \,]$$

4.2 Validating the results

To validate the found isoefficiencies, we have written a simulator in Scala. This simulator executes the algorithm to test on a normal (single cpu) pc and produces an execution graph where each computation step in the algorithm is counted and stored as a node. Each node gets a (virtual) processor assigned based on the algorithm's requirements. E.g., a partition local algorithm gets the correct partition processor assigned.

This results in a directed acyclic graph (*dag*) like Fig. 2. Then, using a scheduler, it looks at what the total runtime would have been if we had the actual number of processors available. The simulator currently uses a scheduler which chooses the first available task. Experiments with a random scheduler or shortest-first scheduler produce comparable results. When plotting the execution dag when scheduled it will result in something like Fig. 3. The simulator splits up long running tasks to simulate preemptive multithreading on a processor.

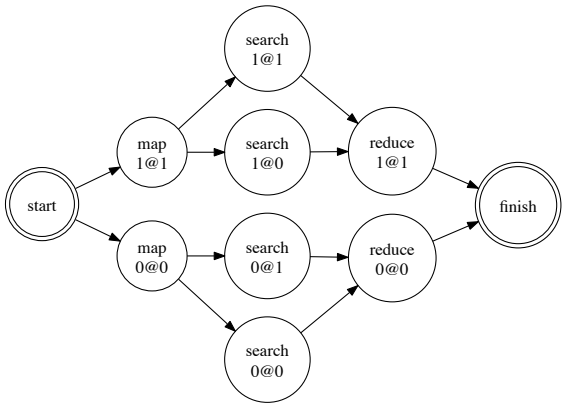

Fig. 2. Example execution dag

4.3 Partition local algorithms

Let's analyze a single, partition local algorithm's (PLA) computation time.

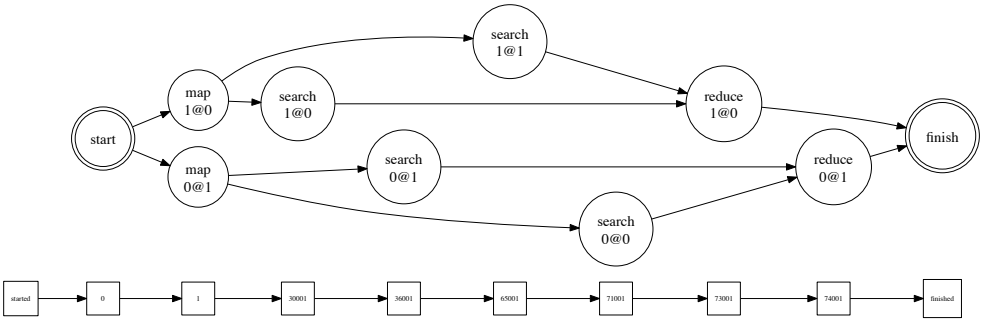

Fig. 3. Example scheduled execution dag

As a baseline we define the total time needed on a single server to be T_1. Assuming that the running time only depends on the size of the data, we can define a cost function for $T_1 = cost(w, |D|)$.

If we have a set of N servers and we distribute work round robin to these servers, we define i as the server the work is assigned to. That server does not necessary have the correct partition to handle the work thus we might have to route the work to another server. The total time needed to perform that instruction is thus the time for performing the route function, communicating the request and response and doing the actual work on the correct partition.

Throughout this paper we will assume that the routing function is something trivial as a modulo operation on some id, taking constant time: $\forall_{i,w} \left[cost(r(i, w)) = c_{route}\right]$.

For now, we'll assume that the communication cost (if needed) is also a constant $c_{comm} = c_{request} + c_{response}$. The time needed for communication is thus:

$$T_{comm}(w, i, N) = \begin{cases} 0 & \text{if } r(N, w) = i \\ c_{comm} & \text{if } r(N, w) \neq i \end{cases}$$

The expectation for T_{comm} is thus $c_{comm} - \frac{c_{comm}}{N}$.

The actual instruction is run against a subset of D, namely the partition data $p = p_{N,i}$: $cost(w, |p|)$. Assuming the costs only depend on the size of the data

Thus the expected total time needed on N servers is:

$$T_N = c_{route} + c_{comm} - \frac{c_{comm}}{N} + costs(w, |p|)$$

And a speedup of

$$S(N) = \frac{T_1}{T_N} = \frac{costs(w, |D|)}{c_{route} + c_{comm} - \frac{c_{comm}}{N} + costs(w, |p|)}$$

If $costs(w, |D|) \simeq costs(w, |P|) = O(1)$, naturally the speedup will be smaller than 1, thus only slower. So any PLA of $O(1)$ will not scale. Any performance increase should thus come from the smaller data size $|p| < |D|$. Looking at the simulated efficiency in Fig. 5 we indeed see that the increasing data size has no effect and that increasing the number of processors decreases the efficiency.

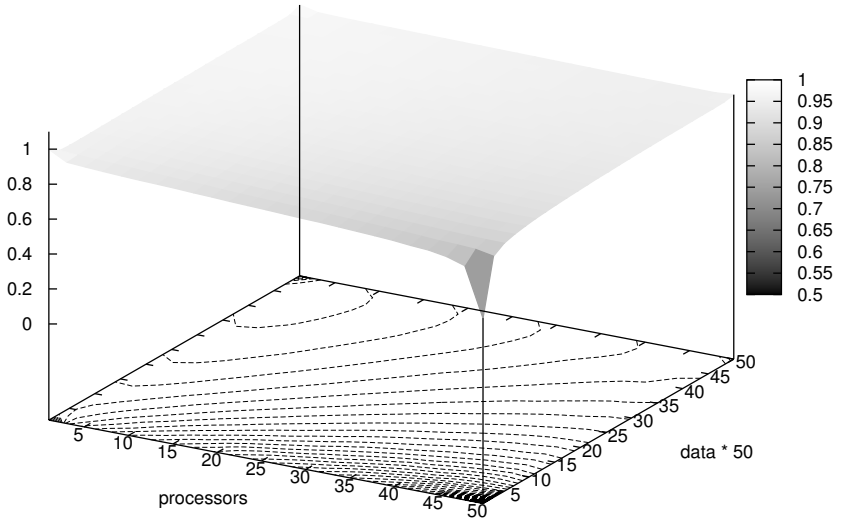

Fig. 4. Efficiency of O(D) PLA

linear PLA's The dataset size for most websites should be significantly larger than the number of processors. Assuming that the dataset is evenly distributed we can state $|p| = \frac{|D|}{N}$. Assuming a $O(|D|)$ costs function $costs(w, |D|) = a|D| + b$, we search for a speedup greater than 1, thus when $T_N < T_1$:

$$
\begin{aligned}
c_{route} + c_{comm} - \frac{c_{comm}}{N} + costs(w, |p|) &< costs(w, |D|) \\
c_{route} + c_{comm} - \frac{c_{comm}}{N} + a|p| + b &< a|D| + b \\
c_{route} + c_{comm} - \frac{c_{comm}}{N} + a\frac{|D|}{N} + b &< a|D| + b \\
N - \frac{c_{comm} a|D|}{N} &< 1 - \frac{c_{route} + c_{comm}}{a|D|}
\end{aligned}
$$

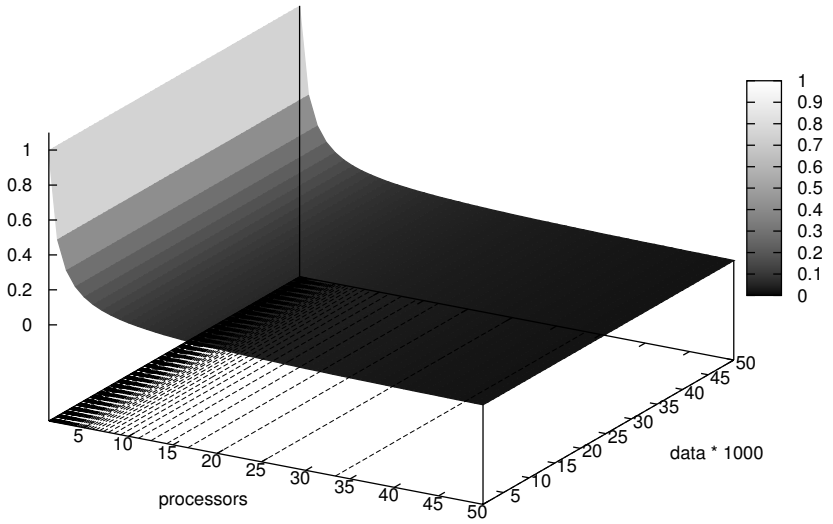

Fig. 5. Efficiency of O(1) PLA

When $|D|$ is sufficiently large this holds for any N and we should thus see some speedup. But how efficient is it?

$$
\begin{aligned}
T_O &= NT_N - W \\
&= N(c_{route} + c_{comm} - \frac{c_{comm}}{N} + costs(w, |p|)) - costs(w, |D|) \\
&= Nc_{route} + (N-1)c_{comm} + Na|p| + Nb - a|D| - b \\
&= Nc_{route} + (N-1)c_{comm} + Na\frac{|D|}{N} + Nb - a|D| - b \\
&= Nc_{route} + (N-1)c_{comm} + (N-1)b
\end{aligned}
$$

Since the overhead is linear for N, it's isoefficiency is $O(D)$. [4] The corresponding simulation is found in Fig. 4. Analyzing this figure, we find that the (simulated) isoefficiency is actually a bit better than $O(N)$ for the simulated ranges. Also, there is a local optimum with higher than 1 efficiency, meaning that for those ranges we have super linear scalability. This can be explained by caches. Since the simulator is run on a single computer, the cache does not grow with the number of processors.

logarithmic PLA's A common operation is a binary search or other index look-up. These algorithms often have a $O(logN)$ complexity: $costs(w, |D|) = alog|D| + b$. If we examine these with the same dataset assumption as in the previous paragraph we find:

$$
\begin{aligned}
c_{route} + c_{comm} - \frac{c_{comm}}{N} + costs(w, |p|) &< costs(w, |D|) \\
c_{route} + c_{comm} - \frac{c_{comm}}{N} + alog|p| + b &< alog|D| + b \\
c_{route} + c_{comm} - \frac{c_{comm}}{N} + alog\frac{|D|}{N} + b &< alog|D| + b \\
c_{route} + c_{comm} - \frac{c_{comm}}{N} + alog|D| - alogN &< alog|D| \\
\frac{c_{comm}}{N} + alogN &> c_{route} + c_{comm}
\end{aligned}
$$

For sufficiently large N, this holds. Note that c_{route} is in most cases very small anyway (e.g. a modulo function). When we look at the efficiency:

[4] Normally, for isoefficiency one would use the best known sequential algorithm's complexity. In this case it is likely to be more efficient than $O(D)$ since only a part of the data is needed. Since we assume partitioned data and focus on scalability we use $O(D)$ for $P = 1$ as well.

$$
\begin{aligned}
T_O &= NT_N - W \\
&= N(c_{route} + c_{comm} - \frac{c_{comm}}{N} + costs(w, |p|)) - costs(w, |D|) \\
&= Nc_{route} + (N-1)c_{comm} + Nalog|p| + Nb - alog|D| - b \\
&= Nc_{route} + (N-1)(c_{comm} + b) + Nalog|D| - NalogN + Nb - alog|D| - b \\
&= Nc_{route} + (N-1)(c_{comm} + b) + (N-1)(alog|D|) - NalogN
\end{aligned}
$$

Looking at the terms separately we find the isoefficiency for routing and communication to be $O(N)$. Looking at the second half of the overhead function we find:

$$
\begin{aligned}
W &= K((N-1)alog|D| - NalogN) \\
alog|D| &= KNalog|D| - KNalogN - Kalog|D| \\
log|D| &= KNlog|D| - KNlogN - Klog|D| \\
0 &= (KN - K - 1)log|D| - KNlogN \\
(KN - K - 1)log|D| &= KNlogN \\
log|D| &= \frac{KNlogN}{(KN - K - 1)} \\
W &= KNalog|D| - KNalogN - Kalog|D| \\
&= KNa\frac{KNlogN}{(KN - K - 1)} - KNalogN - Ka\frac{KNlogN}{(KN - K - 1)} \\
&= \frac{K^2N^2alogN - K^2aNlogN}{(KN - K - 1)} - KNalogN \\
&= O(\frac{K^2N^2alogN - K^2aNlogN}{(KN - K)} - KNalogN) \\
&= O(\frac{KN^2alogN - KaNlogN}{(N - 1)} - KNalogN) \\
&= O(-KalogN)
\end{aligned}
$$

Due to the minus sign, increasing the data set will only make things less efficient.

If we run a simulation with linearly increasing data, we get Fig. 6 which also does not show a stable efficiency. Even assuming an isoefficiency of $O(N^2)$ we see no sign of improved efficiency (Fig. 10).

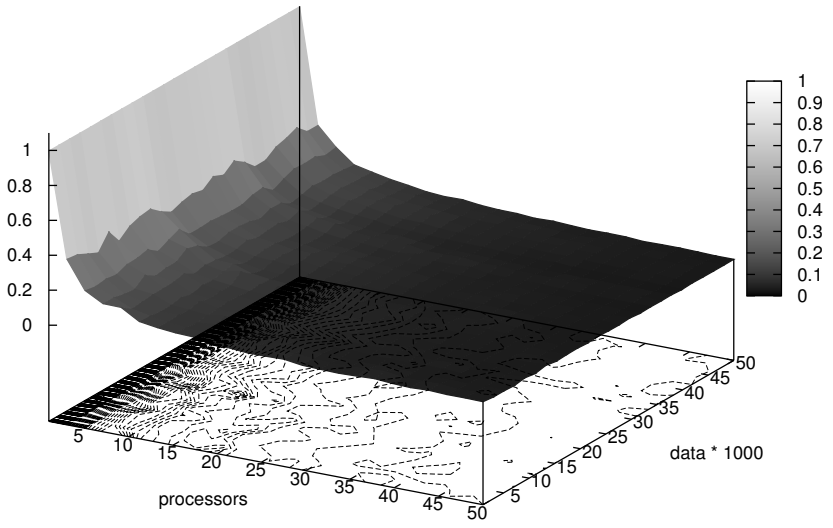

Fig. 6. Efficiency of O(log(D)) PLA

4.4 Simple distributed algorithms

The previous paragraphs assumed that a routing function exists for a given operation. If the operation does not contain the key on which the data is partitioned, it cannot be routed. Also, if the algorithm requires all data to be visited, there is not a single partition to which the operation can be routed.

In these cases a map-reduce kind of strategy can be used [Dean and Ghemawat, 2008]. The initial server i decides to spawn a partition local algorithm to all other servers and itself. It then waits for all partition results to return and *reduces* these partial results to the actual result. We have already calculated the scalability of partition local algorithms. For these distributed algorithms the only additional factor is the reduce function. Looking at the overhead function for these distributed algorithms, we find:

$$
\begin{aligned}
&T_O \\
&= N(c_{route} + c_{comm} - \tfrac{c_{comm}}{N} + \tfrac{Ncosts(w,|p|)}{N} + costs(reduce(N)) - costs(w,|D|) \\
&= N(c_{route} + c_{comm} - \tfrac{c_{comm}}{N} + costs(w,|p|) + costs(reduce(N)) - costs(w,|D|)
\end{aligned}
$$

The only scalable algorithm class we have considered are linear PLA's. If the PLA used in the distributed algorithm has less complexity, the PLA does not scale so the distributed version will not either. The overhead function for linear PLA's differs only in the additional $costs(reduce(N))$ parameter for distributed algorithms. As N is considered much smaller than $|p|$ and is asymptotically constant towards D, it is unlikely that it has any scalability effects. Again, the simulator confirms this in Fig. 9.

If we look at the simulator results for lower order PLA's, we find the same non-scalable results for distributed PLA's of $O(1)$ (Fig. **??**) and $O(log(D))$ (Fig. 8).

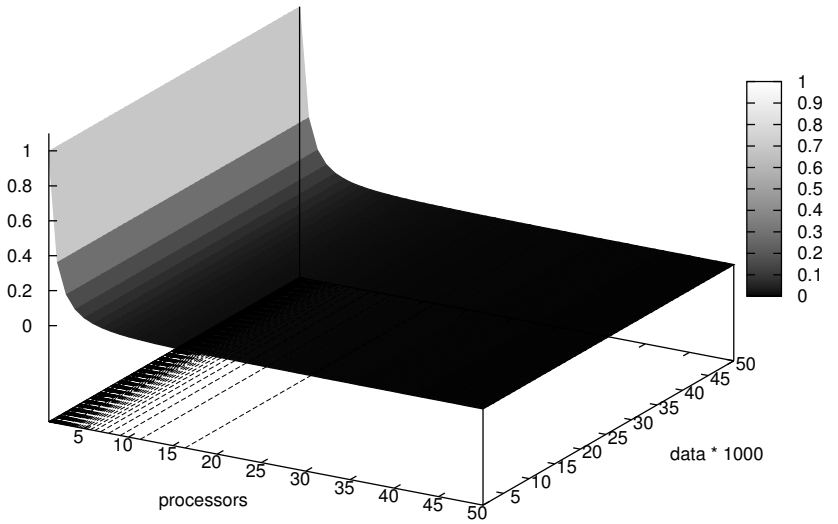

Fig. 7. Efficiency of distributed O(1) PLA

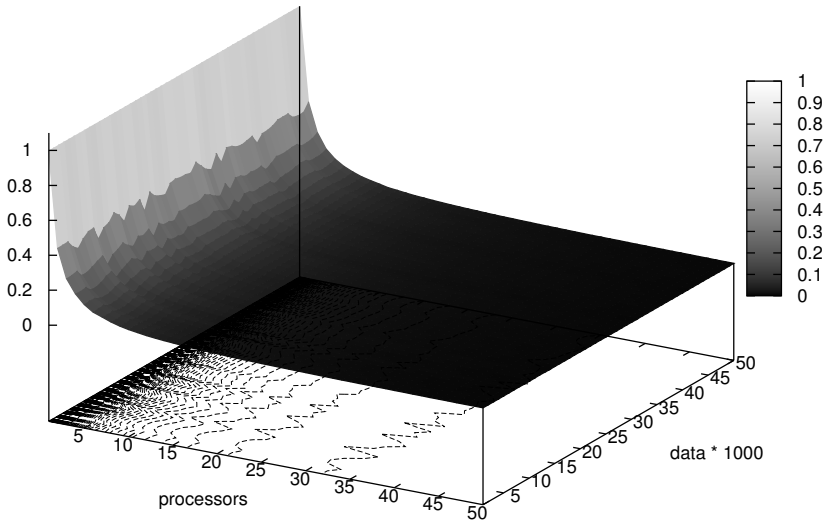

Fig. 8. Efficiency of distributed O(log(D)) PLA

4.5 Summary

With our model we have calculated for different complexity classes of partition local algorithms their respective isoefficiencies. These results were validated by our simulator. We did not find isoefficiency differences between operations that can be directly routed and operations that need to be distributed. We assume that this is due to the fact that our model (and our simulation) do not consider concurrent operations.

5 Future work

As concluded in the previous section we expect a difference in scalability for routable and non-routable operations when concurrent operations are considered, e.g. in the case of multiple concurrent website visitors. In Sect. 2 we argued that in designing a data model and the partitioning of a data model a lot of trade offs have to be considered. When considering a more complex relational data model, analyzing the possible partitioning options and their scalable properties can be difficult. To apply *parti-*

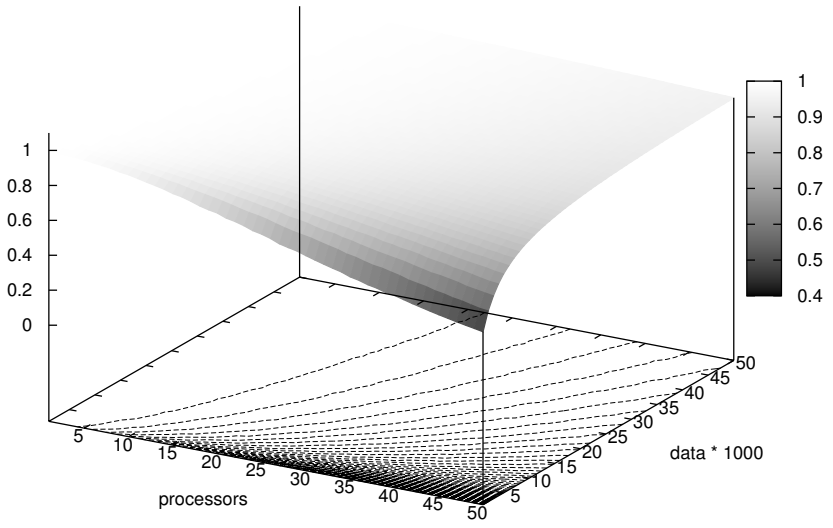

Fig. 9. Efficiency of distributed O(D) PLA

tioning aware data modeling to larger real-life practical examples more classes of algorithms need to be investigated such as more non-local algorithms.

In principle, the presented model can be applied not only to study isoefficiency of distributed web servers but also to cell processors for smaller scale applications. Since sequential programming is no longer sufficient to fully utilize a cpu, the number of cores per cpu increases and the need for parallel computing is growing. Applying our results on cell processors may be an interesting subject for future research. Going even further, by partitioning to really small datasets might make a partition fit into faster caches more local to a cpu which could lead to super-linear speed up in some cases. While the model accurately gives an upper bound on isoefficiency the effects of caching could be added to the model and the simulator, making it possible to analyze these scenarios as well.

For websites, *latency* is an important metric. Normally, a fixed number of requests is being handled at a single time. The lower the latency, the more requests per second the system can handle, thereby also in-

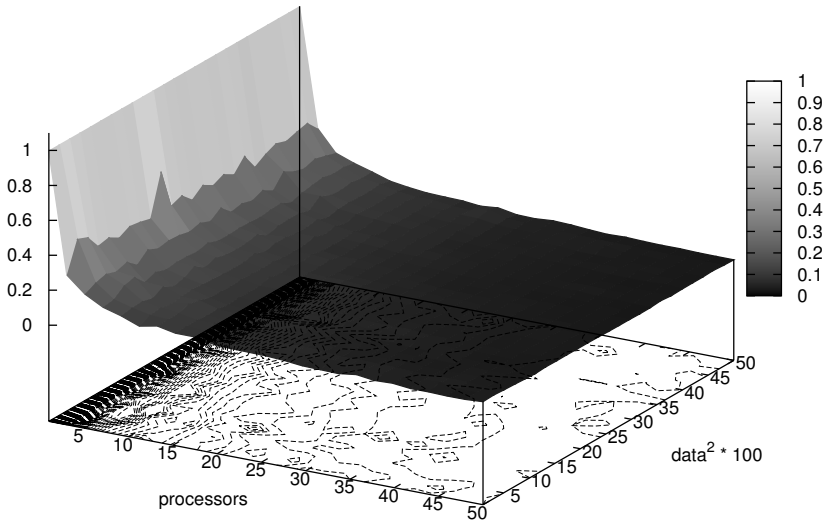

Fig. 10. Efficiency of O(log(D)) PLA with $data^2$ scale

creasing the bandwidth in high-load situations. Also, the strong relation between latency and user frustration (see e.g. [Kohavi and Longbotham, 2007]) adds to the importance of optimizing for latency. Making more requests partition-local should help here, e.g. with algorithm classes which have a few steps and need to gather information from a few places. Here, partition locality would really improve latency although probably not necessarily scalability. Another subject to investigate is the effects of *decrease in concurrency* per partition. Since the number of concurrent threads per processor is approximately reduced by n for each partition-local algorithm, there should be less locking and thus improved performance.

In the current model it is assumed that each partition has its own processor. This means that adding more processors in a running system would also require *repartitioning*, i.e. increasing the number of partitions which on its turn could require heavy data reshuffling. A simple solution could be to over-partition the cluster, e.g. have more partitions

per processor. In this case a full partition could be reallocated to the new processor.

Non-linear or *remote routing* systems use p2p and trees to assign blocks of tuples to a partition, e.g. Hazelcast and Terracotta. Other systems have centralized directories, e.g. Apaches' Hadoop has name-nodes. In this article it is assumed that a system is fully loaded and data is already partitioned. More complex routing might introduce worse than $O(n)$ routing and have different scalability aspects.

6 Related work

A lot of research is done on *SPMD and MIMD models*. There are common platforms for distributed systems and a lot of research goes into optimizing algorithms for these models. For example, *Partitioned Global Address Space (PGAS)* languages, such as UPC [The UPC Consortium, 2005], Titanium [Hilfinger et al., 2005] and Crays' Chapel focus on giving a developer tools to explicitly partition data constructs to be private to a thread. They focus more on algorithmic optimization while we focus on higher level data schema design.

Graph partitioning research (e.g. [Goehring and Saad, 1995], [Hendrickson, 1998], [Pinar and Hendrickson, 2001], [Leland, 1994], [Moulitsas and Karypis, 2008]) looks at a given data set as a graph and then tries to optimize partitioning that graph. Most algorithms focus on minimizing communication between partitions for a specific application. As noted in [Hendrickson, 2000] and [Hendrickson and Kolda, 2000] this has some drawbacks. In our research we try avoiding looking at individual tuples but only use the relations as defined in a relation schema.

More specifically for web sites, a lot of research is done on scaling web clusters. E.g. [Garcia et al., 2008] concludes that web cluster scaling is near-linear until the database becomes the bottleneck. Research is done to make the database more scalable, e.g. Ganymed [Plattner and Alonso, 2004] allows for a more or less transparently scalable database layer. This particular research only allows for a single write (master) database but multiple read (slave) databases. For most use cases, scaling read operations will be sufficient. However, we are more interested in the generic case also allowing multiple writing instances where we have to deal with the 'P' from CAP. Other research is focusing on generic distributed hash tables and similar NoSQL data systems, e.g. [Schintke

et al., 2010] which uses optimized Paxos commits to allow scalable distribution of data. Since Paxos commits require a lot of communication and are not guaranteed to finish, an engineer might decide to use this only for some part of the schema. Deciding which part of the schema is viable to use inter-partition transactions is the ultimate goal of our research. Some research is done focusing more on scaling hardware or scaling on an application server level then on application design, e.g. [Veal and Foong, 2007]. [Gunther, 2001] discusses how to measure an existing application and predict growth path. As the article itself states, by changing the program the predictions were already invalidated. In our opinion this makes a case to focus on the design phase and try to derive more generic, higher level predictions up front.

The idea of routing a request to a best fitting partition is not new. For example, [Cherkasova and Karlsson, 2001] uses this technique to find the best fitting web server for static content. By distributing the content over several server and keeping some content available on all servers they have found that super-linear scaling can be achieved.

7 Conclusions

We have developed a first step towards a model for reasoning about the performance of access of data which is partitioned over a network. We have applied the model successfully for partition local algorithms of different complexity. Using a simulator we have confirmed these results.

References

Ludmila Cherkasova and Magnus Karlsson. Scalable web server cluster design with workload-aware request distribution strategy ward. In *Proceedings of the Third International Workshop on Advanced Issues of E-Commerce and Web-Based Information Systems (WECWIS '01)*, WECWIS '01, pages 212–, 2001.

Jeffrey Dean and Sanjay Ghemawat. MapReduce: simplified data processing on large clusters. *Commun. ACM*, 51:107–113, January 2008.

Daniel F. Garcia, Rodrigo Garcia, Joaquín Entrialgo, Javier Garcia, and Manuel Garcia. Experimental evaluation of horizontal and vertical

scalability of cluster-based application servers for transactional workloads. In *Proceedings of the 8th conference on Applied informatics and communications*, pages 29–34, 2008. ISBN 978-960-6766-94-7.

Seth Gilbert and Nancy Lynch. Brewer's conjecture and the feasibility of consistent available partition-tolerant web services. In *In ACM SIGACT News*, page 2002, 2002.

T. Goehring and Y. Saad. Heuristic algorithms for automatic graph partitioning, 1995. Technical Report umsi-94-29, Department of Computer Science, University of Minnesota.

Ananth Y. Grama, Anshul Gupta, and Vipin Kumar. Isoefficiency: Measuring the scalability of parallel algorithms and architectures. *IEEE Concurrency*, 1:12–21, 1993.

Neil Gunther. Performance and scalability models for a hypergrowth e-commerce web site. In Reiner Dumke, Claus Rautenstrauch, Andre Scholz, and Andreas Schmietendorf, editors, *Performance Engineering*, volume 2047 of *Lecture Notes in Computer Science*, pages 267–282. Springer Berlin / Heidelberg, 2001. ISBN 978-3-540-42145-0.

Bruce Hendrickson. Graph partitioning and parallel solvers: Has the emperor no clothes? (extended abstract). In *Proceedings of the 5th International Symposium on Solving Irregularly Structured Problems in Parallel*, pages 218–225, London, UK, 1998. Springer-Verlag.

Bruce Hendrickson. Load balancing fictions, falsehoods and fallacies. *Applied Mathematical Modelling*, 25(2):99 – 108, 2000.

Bruce Hendrickson and Tamara G. Kolda. Graph partitioning models for parallel computing. *Parallel Computing*, 26:1519–1534, November 2000.

P. Hilfinger, D. Bonachea, K. Datta, D. Gay, S. Graham, B. Liblit, G. Pike, J. Su, and K. Yelick. Titanium Language Reference Manual, 2005. Tech. Report EECS-2005-135.

Ron Kohavi and Roger Longbotham. Online experiments: Lessons learned. *Computer*, 40:103–105, 2007.

B. Leland, R. Hendrickson. An empirical study of static load balancing algorithms. In *Scalable High-Performance Computing Conference*, pages 682–685. IEEE, 1994.

Erik Meijer and Gavin Bierman. A co-relational model of data for large shared data banks. *Queue*, 9:30:30–30:48, March 2011. ISSN 1542-7730.

Irene Moulitsas and George Karypis. Architecture aware partitioning algorithms. In *Proceedings of the 8th International Conference on Algorithms and Architectures for Parallel Processing*, ICA3PP '08, pages 42–53, Berlin, Heidelberg, 2008. Springer-Verlag.

Ali Pinar and Bruce Hendrickson. Partitioning for complex objectives. *Parallel and Distributed Processing Symposium, International*, 3:1232–1237, 2001.

Christian Plattner and Gustavo Alonso. Ganymed: Scalable replication for transactional web applications. In Hans-Arno Jacobsen, editor, *Middleware 2004*, volume 3231 of *Lecture Notes in Computer Science*, pages 155–174. Springer Berlin / Heidelberg, 2004. ISBN 978-3-540-23428-9.

Florian Schintke, Alexander Reinefeld, Seif Haridi, and Thorsten Schütt. Enhanced paxos commit for transactions on dhts. In *Proceedings of the 2010 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing*, CCGRID '10, pages 448–454, 2010. ISBN 978-0-7695-4039-9.

The UPC Consortium. Unified Paralell C (UPC) Language Specification (V 1.2), 2005. High Performance Computing Laboratory, The George Washington University.

Bryan Veal and Annie Foong. Performance scalability of a multi-core web server. In *Proceedings of the 3rd ACM/IEEE Symposium on Architecture for networking and communications systems*, ANCS '07, pages 57–66, 2007. ISBN 978-1-59593-945-6.