Scaling PageRank to 100 Billion Pages

Stergios Stergiou
utopcell@google.com

ABSTRACT
Distributed graph processing frameworks formulate tasks as sequences of supersteps within which communication is performed asynchronously by sending messages over the graph edges. PageRank’s communication pattern is identical across all its supersteps since each vertex sends messages to all its edges. We exploit this pattern to develop a new communication paradigm that allows us to exchange messages that include only edge payloads, dramatically reducing bandwidth requirements. Experiments on a graph of 38 billion vertices and 3.1 trillion edges yield execution times of 34.4 seconds per iteration, suggesting more than an order of magnitude improvement over the state-of-the-art.

CCS CONCEPTS
• Information systems → Graph-based database models; Computing platforms.

KEYWORDS
graph processing, implicit targets, pagerank

ACM Reference Format:

1 INTRODUCTION
The advent of web search and the popularity of social networks has led to the creation of very large graphs. Facebook reports 2 billion monthly active users [5] and at least 1 trillion connections between them [12], while Google reports indexing hundreds of billions of web pages [6]. Executing PageRank at this scale requires efficient graph processing systems. Several such systems have been proposed in the recent past, including Pegasus [19], Pregel [24], HaLoop [9], Giraph [4], comb. BLAS [10], PowerGraph [14], Giraph++ [32], Naïad [25], GPS [29], Mizan [20], Spark/GraphX [15], Blogel [33], PowerLyra [11], BIDMat/Kylix [35], X-Stream [28], Chaos [27] and GraphChi [21]. Extensive comparisons between state-of-the-art frameworks can be found in [16] and [34].

One of the main tasks of a large-scale graph processing framework is to efficiently implement the following primitive: iterate over the graph’s vertices, generate messages for each of the edges’ targets, and deliver all generated messages to the appropriate targets. To achieve this, the input graph is first split into a number of partitions. Partitioning can either be user-driven, the output of a graph balancing algorithm or random, but typically graph vertices are randomly placed in partitions. A source partition transmits messages towards a target partition in the form of {message target id, message payload} tuples. Target identifiers are necessary so that the receiving partition is able to deliver the message to the appropriate vertex upon reception. However, vertex ids are large hashes whose size can dominate the size of the tuple (see Section 3.1). This is especially true for PageRank, where payloads are typically only 32-bits long. While some graph processing systems map vertex ids to a contiguous range of integers [29], thereby reducing their size, relabeling vertex ids is expensive for web-scale graphs.

Orthogonally, even though random placement generates well-balanced partitions, any partitioning scheme will lead to inlinks imbalance when there exist large in-degree vertices. To address this, graph processing systems introduced partitioning schemes where individual edges (instead of vertices) are randomly partitioned [14]. This approach indeed leads to partitions that are also inlink-balanced, but imposes a significant overhead as the outlinks of any given vertex no longer reside in the same partition. As a consequence, multiple partitions need to maintain state information of a given vertex and additional communication is necessary to keep this information synchronized. Alternatively, source-side aggregation has been used to address the problem, where messages towards the same target are combined before sent. This is effective because there will be a large number of edges on every partition pointing to a large in-degree vertex. However, source-side aggregation requires maintaining a mapping from unique edge targets to aggregated payloads, which is impractical since the number of unique edge targets per partition is usually the same order of magnitude as the number of all edges in the partition.

In this paper, we introduce a new communication paradigm for distributed graph processing systems, specifically optimized for PageRank-like communication patterns, which allows us to: (1) exchange messages between partitions by only transmitting message payloads; and (2) perform partial source-side message aggregation, only for frequent message targets. First, we impose an order on the messages exchanged between two arbitrary partitions within each superstep. This allows us to memorize the order of incoming target ids per {source, target} partitions tuple. Consequently, partitions are able to transmit just message payloads: the corresponding target partitions are expected to pair the incoming payloads with the correct target vertex identifiers upon receiving them (see Figure 1). Second, we observe that after incoming id orders have been obtained, source partitions no longer need to maintain target ids for each edge and therefore, only the target partition id is maintained for each edge. This allows us to accelerate the local iterations over each partition’s edges by transforming vertex identifiers into a more compact form that maintains just enough information for the system to be able to send the corresponding messages (see Section 3.3). Third, we establish a fast, deterministic method for selecting
orders
 buffers
 messages
 edges

Figure 1: Implicit Targeting. Edge target identifiers are labeled as $K_i$. Edge payloads are labeled as $V_i$. Partitions $P_1$ and $P_2$ are sending messages to partitions $P_3$ and $P_4$. (i) each partition iterates over the edges assigned to it and buffers messages per target partition; (ii) buffered messages are sent to their respective target partitions; (iii) packets arrive at target partitions; (iv) incoming messages containing payloads are paired with pre-computed target vertex identifiers (labeled as $L_j$) and delivered to the application. $L_j$ are local vertex identifiers, only known to their partitions, corresponding to the global $K_j$ identifiers.

Figure 2: Source-side Aggregation. Partitions $P_1, P_2$ and $P_4$ contain 5, 3 and 6 edges respectively towards vertex $v$ in partition $P_3$. (a) aggregation is not performed; (b) source-side aggregation is performed with frequency threshold equal to 4. Messages from partitions $P_1$ and $P_4$ are aggregated, while messages from partition $P_2$ are sent directly to $v$ as they are below the frequency threshold.

a small subset of target vertices in each partition for which we perform source-side payload aggregation. Specifically, we deduce a set of target ids per partition whose elements are the target of at least $f_{TH}$ edges in the partition, where $f_{TH}$ is a system-generated local frequency threshold (see Figure 2). This set includes all large in-degree vertices w.h.p., is small compared to the number of unique target ids, and is obtained via a lightweight streaming algorithm that requires memory only proportional to the size of the set.

We implemented our new communication paradigm, aptly named implicit targeting (as target ids are implicit in messages), on Ithanos, a Yahoo! graph processing framework that has previously been used for ML [31] [18] and other [30] distributed workloads. PageRank [26], the classic web graph algorithm that simulates the surfing behavior of a random user [22], has been extensively used for evaluating the performance of graph processing frameworks. We evaluated PageRank on graphs with up to 96.3 billion vertices. Specifically, for a graph of ~38 billion vertices and ~3.1 trillion edges, we obtained execution times of 34.4 seconds per iteration. We also compared our system against Giraph on smaller public graphs and demonstrated a 30X mean speedup.

The rest of the paper is organized as follows. In Section 2, we briefly describe PageRank and the online algorithm used to detect frequent elements. In Section 3 we describe our implicit targeting communication pattern. In Section 4 we analyze the performance of our source-side message aggregation algorithm. In Section 5 we report experimental results.

2 PRELIMINARIES

2.1 PageRank

Let $C$ be the set of pages in a web corpus, $d(v)$ be the number of outlinks of page $v$ and $W$ be a column stochastic matrix that represents the connections between pages such that $W[u, v] = 1/d(v)$ if there exists a link from page $v$ to page $u$ and 0 otherwise. PageRank imposes a total order on $C$ assuming ties are broken arbitrarily. It denotes the probability that a user (called the random surfer) would end up at a particular page if they started from an arbitrary page and either visited one of the outlinks from that page, or with a small probability $\alpha$ jumped to a pre-specified set of pages (named the teleportation set $T$). For pages without any outlinks the user would jump directly to a page in $T$.

For each graph vertex $v$ we maintain the current iteration PageRank ($rank(v)$) and the next iteration PageRank ($next\_rank(v)$) values. $rank(v)$ is initialized to either 1.0 or to a value computed at a previous sync point. In the main computation phase, before processing any edges, all local $next\_rank$ values are copied into $rank$, just before they are initialized to 0. Each process iterates over all edges present in its graph partition. For each edge $(u, v)$ it sends a APR message to target vertex $v$, where $\Delta PR = (1 - \alpha) \cdot rank(u)/d(u)$. It also adds $\alpha \cdot rank(u)$ to a local teleportation variable teleportation. If $u$ has no outlinks, it adds the complete $rank(u)$ to teleportation. Concurrently, upon reception of a message the process adds the
Algorithm 1 FrequentItems< $H$ >

1: function Process(value)
2:     if value $\in$ items.keys() then
3:         items[value] $\leftarrow$ items[value] + 1;
4:     else if items.size() $<$ $H$ then
5:         items[value] = 1;
6:     else
7:         for item $\in$ items.keys() do
8:             items[item] $\leftarrow$ items[item] - 1
9:         if items[item] = 0 then
10:            delete items[item];

corresponding $\Delta PR$ to the next_rank($v$) of the graph vertex $v$ associated with the message. If $v$ is not one of the vertices that the process is maintaining, then it redirects the incoming $\Delta PR$ to teleportation. This can happen if edge $(u, v)$ points to an uncrawled page. At the end of the iteration, the master process collects all local teleportation values from its peers, aggregates them, and distributes the sum to its peers and itself.

2.2 Frequent Items

Following, we present an algorithm that detects all target ids of frequency $f > n/(H + 1)$, of which there can be at most $H$, where $n$ is the number of all target ids in the partition. Let us first describe the algorithm via an example for the base case of $H = 1$ [8]. Given an array $A$ of $n$ elements, we would like to detect the majority element if one exists. That is, we would like to detect an element that appears more than $n/2$ times in the array. The algorithm works in a streaming mode: it does not know $n$ in advance and only examines each element once. It maintains a counter and candidate variable. Initially, counter = 1 and candidate = $A[1]$. It examines each of the array elements $A[2], \ldots, A[n]$. If the current element is the same as the one in candidate, then counter is incremented. Otherwise, counter is decremented. If counter becomes negative, then candidate is set to the current element and counter is reset to 1. In the end, if a majority element exists, it will be present in candidate. Intuitively, this holds because the number of all non-majority elements in $A$ is smaller than the number of times the majority element appears in $A$ and therefore they are not enough to displace it from candidate after all elements have been processed. The algorithm is generalized in a straightforward manner for arbitrary $H$ [13]. It maintains a map (items) between each candidate and its corresponding counter. It has $O(n)$ amortized time complexity and $O(H)$ space complexity. It is depicted on Algorithm 1.

3 IMPLICIT TARGETING

Implicit targeting communication assumes that the target partition understands how to deliver messages to a target vertex without actually receiving the graph vertex identifier along with the message it corresponds to. The underlying idea is the following: During each superstep, a target partition $j$ receives sets of messages from its peers without any specific order of arrival. However, sets of messages arriving from a particular source partition $i$ will be delivered in-order to $j$. We guarantee this by adopting unidirectional message queues for point-to-point communication between $i$ and $j$. So long as the source partition $i$ iterates over its edges in the same order, the subset of the edges corresponding to target partition $j$ will also be processed in the same order and therefore, the corresponding messages will be sent out, and be received by partition $j$ in the same order.

3.1 Vertex Identifiers

For large graphs, we can see that identifiers that are smaller than 128-bits are impractical via a birthday paradox argument: Let $H$ be the number of unique values of a vertex id and $q(n, H) = 1 - p(n, H)$ be the probability that no two values are the same in a set of $n$ values randomly drawn from $[1..H]$ with repetition. Then

$$q(n, H) = 1 - \frac{1}{H} \cdot \frac{1}{H} \cdot \frac{1}{H} \cdots \frac{1}{H} \approx e^{-n/H}$$

Solving for $n$, we obtain the size of graph $n(p, H)$ for which there will be collisions between its vertex identifiers with probability at least $p$:

$$n(p, H) = \sqrt{2H \ln \frac{1}{1 - p}}$$

Assuming 64-bit vertex identifiers and $p = 10^{-5}$, there will be collisions between ids with graphs of as few as 6.1 million vertices. Arguably, such graphs are not the target of a large-scale graph framework. Moreover, for public graphs such as the Common Crawl Graph (see Table 1), the collision probability is 26%. On the other hand, 128-bit ids allow for graphs with 820 billion vertices before collisions for $p = 10^{-15}$.

3.2 Target Learning Phase

In order for each partition to learn the graph’s vertex identifiers, a target-learning phase is executed during graph loading. After a partition has loaded its subset of the graph, it knows the partition out-degrees for each of its peer partitions, i.e. the number of edges that point to graph vertices in each of the other partitions. In the first part of the target-learning phase, partitions also collect partition in-degrees, i.e. the number of edges they should be expecting from each peer partition during each superstep. In the second part of the target-learning phase, each partition scans its subset of the graph and sends out messages that contain the edges’ primary target identifiers. The partition can quickly compute the target for each edge as it knows the exact range of identifiers each partition is responsible for. At the same time, each partition receives sets of messages from its peers. For each message, it obtains the mapping from the primary vertex identifier to the local vertex identifier, which it then places at the end of the incoming order vector for the corresponding source partition. After sending out all messages, each partition sends termination messages to each peer.

3.3 Partition Learning Phase

Being able to send payloads without target ids significantly reduces communication requirements and surfaces a new opportunity for optimization. First, we note that it is no longer necessary for each source partition to know the primary vertex identifiers of its edges’
targets. This information would be necessary for a traditional communication pattern, as the target partition needs to associate the primary id to a local id in order to process each payload appropriately. However, in the implicit targeting communication pattern, it is sufficient for the source partition to know just the target partition id, and for the target partition to know the originating partition for each set of incoming payloads. Therefore, each partition iterates over its edges, deduces their target partitions, and stores only target partition identifiers for each of the edges. This typically reduces the footprint for each partition’s edges eight-fold (since 128-bit identifiers for vertices are replaced with 16-bit identifiers for the partitions they belong to) and speeds up iterating over a partition’s edges. The partition learning phase occurs concurrently with the target learning phase. We note that the time required to establish the implicit targeting communication pattern is always less than the execution time of a single superstep that iterates over all graph edges in a traditional communication pattern, and is therefore very efficient.

4 MESSAGE AGGREGATION

Maintaining in-order vectors per partition introduces another challenge to the system: As each target partition may access its $K - 1$ incoming-message orders non-sequentially, it would be inefficient to store them on secondary storage and are therefore maintained in memory, which is a limited resource. While graph partitions are typically well-balanced, inlinks per partition are not. In fact, we have observed inlink count imbalances of five-fold or more. One way to address this challenge would be to rebalance the graph as a preprocessing step. However, graph partitioning algorithms are expensive to execute on large-scale graphs, so that any benefit is typically not enough to offset the delay they introduce to the overall execution time except for very long-running computations. An alternative to graph rebalancing would be to combine messages with the same target primary identifier on the source side. This approach would indeed address the inlinks imbalance. However, in order to support source-side aggregation, one needs to maintain a structure that is $O(p_u)$, where $p_u$ is the number of unique edge targets in the partition. Hash-based graph partitioning implies that this is $p_u = O(p_s)$, where $p_s$ is the number of edge targets in the partition. Therefore, this approach simply moves the memory bottleneck to a different component of the system.

A natural way to refine this approach would be to selectively perform source-side aggregation only for the very frequent target primary ids (target ids with high inlink counts within the partition). This optimization indeed removes the memory overhead of source-side aggregation. Nevertheless, it introduces another challenge: It is not possible to deterministically compute the frequency of the elements in an array of size $n$ unless $\Omega(n)$ memory is used [23]. Although this would be a one-time operation, it is still inefficient for very large graphs. We observe the following: We do not really need to know the exact frequency of all target ids in each partition. We only need to identify those that cross a given frequency threshold. This greatly simplifies the time and space complexity of the problem and allows us to adopt the algorithm in Section 2.2. In this context, $n$ is equal to $p_s$.

The algorithm is executed while loading the graph partition. Each primary target identifier is examined via Process() (Algorithm 1). It is essentially executed at no cost, as graph loading is heavily I/O bound and the processing of each vertex is pipelined with the loading of the next vertex. We note that the elements in the set $F_I$ of the resulting $H$ target identifiers are not guaranteed to appear more than $n/(H + 1)$ times. Rather, it is only guaranteed that the target identifiers whose frequency is larger than $n/(H + 1)$ will be part of the resulting set of $H$ ids.

Let us now examine the performance of a system that adopts source-side aggregation. Let $v$ be a vertex with in-degree $d_v$, $K$ be the number of partitions and $\mu = d_v/K$. Conditioned on $v$, the sources of all edges $(s, v)$ are evenly distributed across all partitions $K$. Let us consider applying aggregation on a specific partition with frequency $f = \mu$. Then the probability that $v$ is in $F_I$ is $50\%$. As a result, we expect $d_v/2 + K$ messages to be sent across all partitions corresponding to the $(s, v)$ edges. This approach is very inefficient, as it only discards half of the messages towards vertex $v$. Fortunately, we will subsequently show that almost no edge escapes source-side aggregation when $f = \mu/2$.

Lemma 4.1. The probability that a specific partition $P$ has at most $k$ edges towards vertex $v$ with in-degree $d_v$ is:

$$\sum_{i=0}^{k} \binom{\mu K}{i} \frac{1}{K} \left(1 - \frac{1}{K}\right)^{\mu K - i}$$

Proof. Let $X_i$ be the indicator variable that denotes whether the $i$-th edge has been placed in $P$. Then $X_i$ is a Bernoulli RV with success probability $1/K$. Therefore $\sum_i X_i$ follows the binomial distribution characterized by $d_v$ trials and $1/K$ success probability.

We will use Hoeffding’s inequality to bound Eq. (1).

Theorem 4.2 (Hoeffding [17]). Let $X_i$ be $d_v$ IID Bernoulli RVs with success probability $1/K$. Then it holds that:

$$\mathbb{P}\left(\sum_i X_i \leq \mu - \epsilon d_v\right) \leq e^{-2\epsilon^2 d_v}$$

We can now bound the probability that a specific partition does not apply source-side aggregation to the edges it is responsible for that are targeting vertex $v$.

Theorem 4.3. The probability that a specific partition $P$ has at most $\mu/2$ edges towards vertex $v$ is bounded by $e^{-\mu/2K}$.

Proof. Let $\epsilon = 1/2K$. Then from Eq. (2) we obtain:

$$\mathbb{P}\left(\sum_i X_i \leq \mu/2\right) \leq e^{-2\epsilon^2 d_v} = e^{-\mu/2K}.$$
Table 1: Public Datasets

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Vertices</th>
<th>Edges</th>
</tr>
</thead>
<tbody>
<tr>
<td>Common Crawl [1, 3]</td>
<td>3,315,452,339</td>
<td>128,736,952,320</td>
</tr>
<tr>
<td>Twitter [2]</td>
<td>52,579,682</td>
<td>1,963,263,821</td>
</tr>
</tbody>
</table>

Table 2: Web Graph Properties

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>URLs</td>
<td>37.73 Billion</td>
</tr>
<tr>
<td>Total Outlinks</td>
<td>3.076 Trillion</td>
</tr>
<tr>
<td>Graph Storage Footprint</td>
<td>99.2 TB</td>
</tr>
</tbody>
</table>

Proof. There are at most $K$ partitions on which source-side aggregation can be applied, therefore at most $K$ messages will be emitted as a result of such aggregations. On the partitions where source-side aggregation is not applied, at most $\mu/2$ edges will be targeting vertex $v$. By linearity of expectation, at most $\frac{\mu K}{2} e^{-\mu/2K}$ non-aggregated edges will reach vertex $v$. □

By Eq. 3 we see that this simple, coordination-free scheme applies an exponentially small dampening factor to $d_v$. In Section 5, we will see in practice that source-side aggregation almost completely eliminates inlink imbalance even for small values of $H$.

5 EVALUATION

We explore the performance of PageRank on Hronos and compare it with Giraph [12]. Giraph is a production-hardened framework that has been shown to scale to 1 trillion edge graphs [12]. A recent comparison between Pregel+, GPS, Giraph and GraphLab on the WebUK[7] and Twitter[2] datasets for PageRank can be found in [34]. We provide results for publicly available graphs (Table 1) and large proprietary web graphs (Table 2). We collected results on a 3,000-node cluster of the following node configuration: 64GB RAM, 2x Intel Xeon E5-2620, 10Gbps Ethernet.

5.1 Public Datasets

We first provide results for Hronos and Giraph on the public datasets (see Table 3). We observe mean speedups of more than 15X for 64 threads, and more than 30X for 5,000 threads. We observe that Giraph appears to favor fewer workers, each of which using multiple threads, rather than single-thread workers. For each experiment, we show the best Workers×Threads combination that we could find via a parameter sweep. Better results could not be obtained for any of the experiments even when we allowed for up to 40,000 threads. For all Giraph experiments, workers were given the full memory of the nodes.

5.2 Web Graphs

We provide results on a web graph of 37.7 billion vertices and 3.076 trillion edges (see Table 2). Loading and pre-processing of the graph partitions consumes approximately 9’ for the slowest mapper. Service discovery requires negligible time using a single-node ZooKeeper. We first examine how well balanced graph splits
Table 3: *Hronos* Performance for a Single PageRank Iteration on Public Datasets

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Hronos</th>
<th>P=64</th>
<th>P=384</th>
<th>P=5000</th>
</tr>
</thead>
<tbody>
<tr>
<td>Common Crawl 2012</td>
<td>128.19s</td>
<td>7.63s</td>
<td>&lt;0.1s</td>
<td>&lt;0.1s</td>
</tr>
<tr>
<td>WebUK 2006</td>
<td>3.64s</td>
<td>0.53s</td>
<td>9.50s</td>
<td>9.50s</td>
</tr>
<tr>
<td>Twitter</td>
<td>1.06s</td>
<td>0.21s</td>
<td>9.21s</td>
<td>9.21s</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Giraph</th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Common Crawl 2012</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>67.51s</td>
</tr>
<tr>
<td>WebUK 2006</td>
<td>51.6s</td>
<td>9.54s</td>
<td>9.50s</td>
<td>9.50s</td>
</tr>
<tr>
<td>Twitter</td>
<td>17.57s</td>
<td>6.05s</td>
<td>9.21s</td>
<td>9.21s</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Hronos Speedup over Giraph</th>
<th>P=64</th>
<th>P=384</th>
<th>P=5000</th>
</tr>
</thead>
<tbody>
<tr>
<td>Common Crawl 2012</td>
<td>-</td>
<td>-</td>
<td>27.55X</td>
</tr>
<tr>
<td>WebUK 2006</td>
<td>14.17X</td>
<td>18.00X</td>
<td>95.00X+</td>
</tr>
<tr>
<td>Twitter</td>
<td>16.58X</td>
<td>28.81X</td>
<td>92.10X+</td>
</tr>
</tbody>
</table>

Figure 6: Relative execution times for one PageRank iteration for implicit pattern \((H = 10000)\) and explicit pattern (graph in memory) for 1000, 2000, 3000, 4000 and 5000 graph splits.

Figure 7: Relative execution times for a single PageRank iteration for implicit pattern \((H = 10000)\) and explicit pattern (graph in memory) for 5000 graph splits and 20%, 40%, 60%, 80% graph sub-sampling.

We then examine the number of inlinks to a partition when source-side aggregation is used for different values of \(H\). First, we observe that inlink imbalance is indeed a significant problem. Specifically, the largest inlink count is more than five times larger than average (Figure 4). Inlink count dictates the size of the in-order vectors for the implicit pattern and therefore has a direct impact on memory requirements. Source-side aggregation, through the detection of high-frequency inlinks, resolves the imbalance even for relatively small numbers of \(H\). For instance, for \(H = 30000\), the worst-case imbalance is 19.68%. We also measure the performance of the algorithm for various values of \(H\) and observe there is no overhead when using aggregation. In fact, we observe a performance increase in accordance with the reduction in overall messages sent over the network. For instance, when \(H = 30000\), the messages sent over the network are 97% of all messages, while execution time improves to 96% compared to the baseline case (see Figure 5).

Finally, *Hronos* exhibits strong and weak scalability properties: (1) we observe that performance improves almost linearly with the number of graph partitions for both the implicit and explicit pattern (Figure 6) and (2) we sub-sample the web graph in order to inspect performance as the graph size increases while keeping a fixed split count and observe that the system scales linearly with graph size (Figure 7).

6 CONCLUSION

In this work, we introduced a novel communication paradigm for graph processing frameworks that is optimized for PageRank-like workloads. We evaluated its performance on graphs of up to 96 billion vertices and 5.9 trillion edges and demonstrated more than an order of magnitude improvements over the state-of-the-art.

ACKNOWLEDGMENTS

We would like to thank Dipen Rughwani for collecting experimental results on Giraph and Kostas Tsioutsiouliklis for insightful discussions and proof-reading earlier versions of this work.
REFERENCES