Universiteit Leiden
Opleiding Informatica

Deriving Highly Efficient Implementations of Parallel PageRank

Name: Bart van Strien

Date: 25/07/2017

1st supervisor: Harry Wijshoff
2nd supervisor: Kristian Rietveld

MASTER'S THESIS

Leiden Institute of Advanced Computer Science (LIACS)
Leiden University
Niels Bohrweg 1
2333 CA Leiden
The Netherlands

Abstract

In 1999, Page et al. described their algorithm — PageRank — for scoring pages in their web search engine, Google. Already from the start it was clear that due to the large number of pages to be ranked, the PageRank algorithm's efficiency and optimal performance was and is a critical feature. One method of increasing the efficiency of calculations is based on MapReduce, also originally published by Google. In this paper, we use the forelem framework to go from an initial specification of PageRank to a much more performant implementation.

Contents

1 Introduction

The forelem framework was developed to optimise database queries using traditional compiler optimisations [1]. The same framework can also be applied to distributed parallel programs. In this paper we use an extended forelem framework to optimise a Hadoop MapReduce application for determining its effectiveness on non-database applications. Although a single application was chosen, the methods described are largely generic, and thus applicable to many other applications.

Our application of choice is PageRank [2], a graph ranking algorithm originally developed to find the most important web pages in a web graph. Recently, it has seen use in other fields, such as Social Network Analysis. It is a well-understood algorithm, that has undergone a lot of research, and has seen a lot of optimisation attempts. Additionally, it has been used in several benchmarks, amongst which the BigDataBench [3] benchmark. The specific implementation used, using Hadoop MapReduce, is the one from the Pegasus [4] project. It was selected both for its easily understood implementation (see Section 3.2) and its use in the aforementioned benchmark.

We start with an introduction to forelem in Section 2, followed by an introduction to PageRank in Section 3. Section 3.3 presents a simple specification of PageRank in forelem. In Section 3.4 we show this specification yields a valid solution to PageRank, and prove its termination. Using this specification, we then derive multiple implementations in Section 4, and we analyse their performance in Section 5. Section 6 concludes the paper.

1.1 Related work

Various studies have been done to improve PageRank in the past. Much like the parallel implementation by Kang et al. in [4], other parallel implementations exist. In [5], Gleich et al. describe an implementation based on linear systems, that shows faster convergence. Kohlschütter et al. use an implementation based on Gauß-Seidel in [6] to improve convergence speed. Similarly, methods to approximate the PageRank values have been proposed, like in [7]. In contrast to these previous attempts, where the implementations have been fully human-derived, this paper strives to find implementations through a step-by-step mechanical derivation process using simple transformations.

The forelem framework was introduced, applied and extended by Rietveld and Wijshoff in [8], [1] and [9]. In many respects this paper can be viewed as a successor to [9], which describes a forelem implementation of LU-factorisation.

1.2 Definitions

In this paper we write V_G and E_G for the vertices and edges of a graph $G = (V, E)$, respectively. We also define the out-neighbourhood — or successors — of a vertex v as $nbh_G^+(v) = \{u \in V_G | (v, u) \in E_G\}$. We also call the size of the out-neighbourhood the outdegree $deg_G^+(v) = |nbh_G^+(v)|$. We have an

analogous definition of the in-neighbourhood — or predecessors — of a vertex v as $nbh_G^-(v) = \{u \in V_G | (u, v) \in E_G\}$.

In case the target graph is obvious from its context we will forego the subscript, for succinctness. To disambiguate, in this paper "vertices" will be only used to refer to V_G, and "nodes" will only be used to refer to computing nodes.

2 forelem

The forelem framework was originally devised for the optimization of database queries [1]. At the core is the forelem loop which traverses a subset of a collection of tuples (referred to as a tuple reservoir) in an unspecified order. Because the forelem loop only loosely specifies how the traversal is to be performed, a large number of loop transformations is enabled that can potentially improve the performance of the loop. For instance, forelem loops are inherently parallel as no loop order is defined and no loop carried dependencies exist. All tuples in a tuple reservoir have an equal number of fields. These tuples are used to specify operands that reside in shared spaces. Accesses to shared spaces using tuples are written as simple array accesses.

In a forelem loop, each tuple that is part of the reservoir to be traversed is visited exactly once. Another loop construct with the name of whilelem further generalizes this by visiting each tuple in a reservoir an unspecified number of times in an unspecified order. The body of a whilelem loop typically consists of an if-condition, which indicates whether an action is to be performed for a given tuple. A whilelem loop terminates when no tuple exists within the specified reservoir for which the if-condition evaluates to true. Simply put: there is no more work left to do.

Forelem and whilelem loops are written in a pseudocode with a C-inspired syntax. Syntactically, they are fairly simple and an example of their syntax can be seen in Algorithm 1. Note that these loops operate on an abstract data set. During the transformation and code generation process a suitable data structure will be automatically derived, see also the discussion on Materialisation further on in this paper.

Algorithm 1 A syntax example of forelem and whilelem.

```
forelem (t; t ∈ T)
    X[t.i] = f(t)                ▷ Some calculation for tuple "t"
whilelem (t; t ∈ T) {
    if (X[t.i] > 0)
        X[t.i] = f(t)
}
```

3 PageRank

PageRank [2] is an algorithm to rank a set of web pages based on an objective notion of importance. Intuitively the ranking models random surfers, who, after arriving on a web page, follow a random link until they get bored. Once they are bored, they pick a random website and continue from there. Equation 1 shows the definition of the PageRank of a vertex, where the "boredom" is modeled using a constant $0 < d < 1$, also referred to as the damping factor.

$$PR(v) = \frac{1-d}{|V|} + d \sum_{u \in nbh^-(v)} \frac{PR(u)}{deg^+(u)} \tag{1}$$

Usually PageRank is calculated in a generational fashion, where in every generation a vertex "donates" all its PageRank to its successors, and receives it from its predecessors. As an example, a vertex with a PageRank of 0.2, and an outdegree of 5, yields a rank of $\frac{0.2}{5} = 0.04$ to all its successors. Additionally, the damping factor is applied to all incoming rank, and the (graph-)constant value $\frac{1-d}{|V|}$ is added.

The authors [10] suggest a damping factor of approximately 0.85 yields the best results. In the rest of this paper we will assume $d = 0.85$, unless stated otherwise.

3.1 Sink vertices

Since the random surfers get stuck if a page (vertex) has no outgoing links, whenever they reach such a page, they start over at a random other page. To model this, we can either view these vertices as a special case in our calculations, or somehow modify the graph to represent this behaviour. To do the latter, we define a modified graph G', which is constructed from G as follows:

$$\begin{aligned} G &= (V, E) \\ G' &= (V, E') \\ E' &= E \cup \bigcup_{v \in V \wedge deg^+(v)=0} \{(v, u) | \forall u \in V\} \end{aligned}$$

That is, we connect every vertex v for which $deg^+(v) =$ to every other vertex. This is trivially equivalent to randomly selecting another vertex, our desired behaviour. Since web graphs tend to be relatively sparse, applying this technique can contribute significantly to the size of the graph. As we will see in Section 4.1, it may instead be beneficial to model the effect of these vertices differently.

3.2 PageRank in BigDataBench

The PageRank implementation in the BigDataBench benchmark [3] is the one as implemented in the Pegasus [4] project. It is a fairly straight-forward MapReduce version of the original algorithm. Each iteration evaluates the contributions

of each edge, then all those contributions are summed up to produce the next PageRank values. The implementation stops when the change between iterations in smaller than a given error bound ϵ. In [2] empirical evidence is given this algorithm tends to converge in a small number of iterations.

3.3 forelem specification of PageRank

To come to a simple forelem specification of PageRank, we first observe that Equation 1 describes a stop condition. That is, once Equation 1 holds for all $v \in V$, we have found an assignment of values to vertices.

We can assume, without loss of generality, that there are no duplicate edges in E. Thus, iterating over the predecessors of a vertex is equivalent to iterating over a subset of E. Using this, we can then find an alternate, equivalent definition of PageRank which iterates over the edges, rather than the vertices.

First, we define 3 shared spaces:

- PR, the current PageRank values.
- $Dout$, the outdegrees.
- OLD, for a given edge $(u, v) \in E$, the old value of $PR(u)$.

We initialise E as a tuple reservoir that contains a tuple for each edge in the graph. For those vertices u without outgoing edges tuples $< u, v >$ are added for every vertex v in the graph ($v \neq u$). $Dout$ is defined as the outdegree for each vertex, note that the for nodes that had an outdegree 0, the outdegree has become $|V| - 1$. Further we initialise PR to be equal to $\frac{1}{|V|}$ for every vertex and OLD to be equal to 0 for every tuple of the reservoir:

$$PR = \left\{ \left\langle v, \frac{1}{|V|} \right\rangle \middle| \forall v \in V \right\}$$
$$Dout = \left\{ \langle v, deg^+(v) \rangle | \forall v \in V \right\}$$
$$OLD = \{ \langle u, v, 0 \rangle | \forall (u, v) \in E \}$$

Our algorithm then becomes a single whilelem loop, updating the PageRank on an edge-by-edge basis.

Algorithm 2 The initial forelem specification of PageRank.

```
whilelem (e; e ∈ E) {
    if (PR[e.u] ≠ OLD[e]) {
        PR[e.v] += d (PR[e.u] − OLD[e]) 1/Dout[e.u]
        OLD[e] = PR[e.u]
    }
}
```

3.4 Correctness

To prove this algorithm is a correct implementation of PageRank, we first prove that Algorithm 2 terminates. To aid in our proof, we first define the concept of "momentum":

$$\forall (u,v) \in E : m(u,v) = PR(u) - OLD(u,v) \tag{2}$$

$$M(G) = \sum_{(u,v)\in E} \frac{m(u,v)}{deg^+(u)} \tag{3}$$

Now we observe that in our iterations we mutate the momentum twice. In the first line, an edge "donates" its momentum to its target vertex.

$$M_{i+1}(G) = M_i(G) + d \cdot \frac{m_i(u,v)}{deg^+(u)} \cdot deg^+(v)$$

On the second line, the momentum of that edge is removed.

$$m_{i+1}(u,v) = 0$$

Thus our global change is the combination of those two effects.

$$\begin{aligned} M_{i+1}(G) &= M_i(G) + d \cdot \frac{m_i(u,v)}{deg^+(u)} \cdot deg^+(v) - m_i(u,v) \\ &= M_i(G) - (1 - d\frac{deg^+(v)}{deg^+(u)}) \cdot m_i(u,v) \end{aligned}$$

For a given path $p = (v_1, v_2, \ldots, v_n)$, propagating momentum δ on v_1 to v_n leads to a contribution of $\delta \cdot d \cdot \frac{deg^+(v_2)}{deg^+(v_1)} \cdot d \cdot \frac{deg^+(v_3)}{deg^+(v_2)} \cdot \ldots \cdot d \cdot \frac{deg^+(v_n)}{deg^+(v_{n-1})} = \delta \cdot d^{n-1} \frac{deg^+(v_n)}{deg^+(v_1)}$.

If the algorithm loops infinitely — on a finite set of vertices — then there must be at least one cycle that always remains enabled. Any cycle $(v_1, v_2, \ldots, v_n, v_1)$, where $m(v_1, v_2) = \delta$ contributes exactly $\delta \cdot d^n \frac{deg^+(v_1)}{deg^+(v_1)} = \delta \cdot d^n$ back to vertex v_1. Since the momentum in vertex v_1 is ever-decreasing, within a finite number of iterations this vertex will no longer be enabled. Analogously, this holds for all other vertices in the cycle, thus the cycle itself will no longer be enabled within a finite numbers of iterations. Therefore the algorithm cannot loop infinitely and must terminate.

Lastly, we need to prove our algorithm finds a valid approximation of the PageRank. Our algorithm stops only when there are no edges for which $m(u,v) \geq \epsilon$, hence $M(G) < |E| \cdot \epsilon$. Since $M(G)$ represents the change in ranks in a "full" iteration of the graph, and $M(G)$ is strictly decreasing, a small value of $M(G)$ implies the algorithm is close to finding a solution. We also observe $M(G) = 0$ if the assignment of PageRanks is perfect. As $M(G)$ can be arbitrarily small — by choosing an arbitrarily small ϵ — we can thus state we can approximate the PageRank arbitrarily close, if the algorithm terminates. We have proven above that our algorithm always terminates, if $\epsilon > 0$.

4 Transformations and Implementations

In order to find an efficient parallel implementation of this PageRank-equivalent algorithm, several implementations have been derived from Algorithm 2. As a large number of forelem transformations exist, we have applied only those transformations that looked promising. For brevity, this section will only discuss those transformations that were used in our final selection of implementations.

4.1 Tuple Reservoir Reduction

The tuple reservoir reduction transformation reduces the iterated tuple reservoir's size by identifying common subsets (S) in the reservoir which can be compacted when initialising the reservoir and expanded on demand. This reduction is only performed if this compression and expansion can be handled efficiently.

In order to guarantee an efficient implementation, these subsets S of E are only identified if the tuples corresponding to these subsets S can be enumerated in linear (constant) time by a simple enumeration function G_S. In the case of PageRank these subsets S_u originate from these vertices u which originally had an outdegree of 0 and consist of all added tuples $< u, v >$ (where $u \neq v$). Assuming the vertices are numbered 1 to $|V|$, then each subset S_u consists of $\{< u, i > | 1 \leq i \leq |V|\}$ and the enumeration function ends up being a simple *for*-loop from 1 to $|V|$.

Having identified subsets and their enumeration functions then the tuple reservoir can be reduced by deleting all tuples of a subset and replacing them by a simple stub to the corresponding enumeration function. Then at execution time this subset is being generated one at a time and the loop body is replicated for each of the tuples corresponding to this subset. So if we look at PageRank, the transformation results in the following algorithm.

Algorithm 3 Algorithm 2 after applying Tuple Reservoir Reduction

```
whilelem (e; e ∈ E) {
    if (PR[e.u] ≠ OLD[e]) {
        if (e.v = $S)
            forelem (v; v ∈ V \ {e.u})
                PR[v] += d (PR[e.u] − OLD[e]) 1/Dout[e.u]
        else
            PR[e.v] += d (PR[e.u] − OLD[e]) 1/Dout[e.u]
        OLD[e] = PR[e.u]
    }
}
```

Note that in contrast to the initial specification of the tuple reservoir E, for which for every vertex with outdegree 0 additional tuples $< u, v >$ were created for every $v \neq u$, by using tuple reservoir reduction all these tuples were identified

as reducible subsets and therefore deleted. In fact, the initial expansion of the tuple reservoir was needed to obtain a clean and simple representation in the forelem framework — thereby allowing a cleaner convergence proof and facilitating other transformations to be applied to this specification, see below. Instead of the generation of the forelem construct enumerating all the elements of the subset also an arbitrary element of this subset could have been chosen. In this case it is important that the enumeration function can produce an arbitrary element of the subset in constant time.

4.2 Parallel execution through blocking the tuple reservoir

Similar to the loop blocking optimisation in various optimising compilers, forelem can use blocking to parallelise execution. Given a set to iterate over, we split it in parts, then iterate over these separately, ideally in parallel. Any partitioning of E works, as long as $\bigcup_k E_k = E$. Usually a fair partitioning is used, where every E_k has roughly the same size. Some problems can benefit from different schemes, for example to reduce intercommunication.

Since this is the transformation that yields a parallel program, we will typically apply it at the very end, when the program has already been fully restructured. Our first program might already seem obvious at this point, it is simply the application of loop blocking to our initial specification, see Algorithm 4. Note that although this variant is simple, in practice it will be suboptimal, since it requires global synchronisation on the various writes to PR.

Algorithm 4 A parallelised implementation of Algorithm 2. Each processor k has its own E_k.

```
whilelem (e; e ∈ E_k) {
    if (PR[e.u] ≠ OLD[e]) {
        PR[e.v] += d (PR[e.u] − OLD[e]) 1/Dout[e.u]
        OLD[e] = PR[e.u]
    }
}
```

4.3 Orthogonalisation

A second important optimisation is Orthogonalisation, which adds an outer loop to control the order in which tuples are visited. Algorithm 5 shows an example application of this transformation. In this case, it causes the outer loop to iterate over target vertices, and the inner loop over edges that have said vertex as target. If we subsequently apply parallelisation, we get a program specification in which every PR value has exactly one writer. Indeed, all other implementations used in our experiments start with this Orthogonalisation step.

Algorithm 5 Orthogonalisation applied to Algorithm 2.

```
whilelem (v; v ∈ E.v)
    forelem (e; e ∈ E.v[v]) {
        if (PR[e.u] ≠ OLD[e]) {
            PR[e.v] += d (PR[e.u] − OLD[e]) 1/Dout[e.u]
            OLD[e] = PR[e.u]
        }
    }
```

4.4 Encapsulation

When iterating over a subset of the data, Encapsulation replaces this with a larger subset, designed to simplify iteration. In this case, our implementations generally iterate over the set of target vertices. Typically, this set is mostly equivalent to the set of vertices in the graph, and by definition, it is subset of all vertices. Since the set of vertices is already known, encapsulation allows us to iterate over all vertices, rather than selecting all target vertices from the set of edges. If a vertex is iterated that has no corresponding edge, that iteration is a no-op, therefore it does not alter the calculation. From a mathematical standpoint, this is merely a renaming, but algorithmically it might lead to different data structure selection. See Algorithm 6 for an example.

Algorithm 6 Encapsulation applied to Algorithm 5.

```
whilelem (v; v ∈ V)
    forelem (e; e ∈ E.v[v]) {
        if (PR[e.u] ≠ OLD[e]) {
            PR[e.v] += d (PR[e.u] − OLD[e]) 1/Dout[e.u]
            OLD[e] = PR[e.u]
        }
    }
```

4.5 Localisation

Since we are targeting modern computers, cache effects play an important role in the performance of our applications. One way to improve cache utilisation is by increasing data locality, as done by the Localisation transformation. Where shared space data is initially stored separate from tuples, localisation causes shared space data to be stored — or localised — in the tuples directly. Within our example, the values of OLD are stored per-edge, but separate from the edges. We can use the Localisation transformation to interleave the OLD data with the edge data, yielding Algorithm 7.

Algorithm 7 Localisation applied to Algorithm 2.

```
whilelem (e; e ∈ E) {
    if (PR[e.u] ≠ e.old) {
        PR[e.v] += d (PR[e.u] − e.old) 1/Dout[e.u]
        e.old = PR[e.u]
    }
}
```

4.6 Materialisation

So far we have been iterating over tuple reservoirs, without specifying the relevant data structure. Materialisation is the first step in the process of deriving different data structures, including array-of-structs and struct-of-array data structures. In this first step, a particular order is chosen for the tuples in the initially unordered tuple reservoirs. Algorithm 8 uses an array of structures instead, opting to give every edge its own struct.

Algorithm 8 Materialisation applied to Algorithm 2.

```
whilelem (e; e ∈ PE_len) {
    if (PR[B[e].u] ≠ OLD[B[e]]) {
        PR[B[e].v] += d(PR[B[e].u]−OLD[B[e]]) / Dout[B[e].u]
        OLD[B[e]] = PR[B[e].u]
    }
}
```

4.7 Composing transformations

The transformations as described in the previous section can be composed so that their effect is multiplied. Note, that the transformations have forelem code as input and produce forelem code as output, so they are inherently composable. The composition of multiple transformations allowing different orders of application — including re-use of transformations — leads to many different variations of the same program. Indeed, the same transformation may be applied multiple times.

Algorithm 9 shows one of our candidate implementations, obtained by applying orthogonalisation, encapsulation and loop blocking. If an additional localisation step is used, adding OLD to our tuples, Algorithm 10 is obtained.

Algorithm 9 Orthogonalisation, encapsulation and loop blocking, applied to Algorithm 2 in order.

```
forelem (v; v ∈ V_k)
    forelem (e; e ∈ E.v[v]) {
        if (PR[e.u] ≠ OLD[e]) {
            PR[e.v] += (d(PR[e.u]−OLD[e])) / Dout[e.u]
            OLD[e] = PR[e.u]
        }
    }
```

Algorithm 10 Orthogonalisation, encapsulation, localisation and loop blocking applied to Algorithm 2, in order.

```
whilelem (v; v ∈ V_k)
    forelem (e; e ∈ E'.v[v]) {
        if (PR[e.u] ≠ e.old) {
            PR[e.v] += (d(PR[e.u]−e.old)) / Dout[e.u]
            e.old = PR[e.u]
        }
    }
```

An alternative derivation might use early materialisation to produce an implementation that more closely matches an array-of-structs rather than a struct-of-arrays approach, as seen in Algorithm 11. Finally, note that in addition to (multiple) materialisation steps, orthogonalisation, encapsulation, localisation and loop blocking also tuple reservoir reduction as described in Section 4.1 can be applied to all resulting algorithms.

Algorithm 11 Multiple materialisation and localisations steps applied to Algorithm 2.

```
whilelem (v; v ∈ PV_len[k])
    forelem (u; u ∈ B[v].len) {
        if (PR[B[u].u] ≠ B[v].old[u]) {
            PR[v] += (d(PR[B[u].u]−B[v].old[u])) / B[u].dout
            B[v].old[u] = PR[B[u].u]
        }
    }
```

5 Experiments

To determine the performance of the derived PageRank implementations we used the implementation in and datasets of BigDataBench (see Section 3.2)

as a reference. Due to its inclusion in an established Big Data benchmark, this implementation should provide a good baseline performance. As we want to minimise the effect of fixed overhead, all measurements are done on large datasets. Since we use a data generator from the BigDataBench benchmark, we can use arbitrarily large inputs. Given the work in [2] and runtime restrictions, experiments have been performed with datasets between 2^{20} vertices and 2^{28} vertices, with approximately 4.6 million to 2 billion edges respectively.

In our final selection of tested implementations, Implementation 1 corresponds to Algorithm 9, Implementation 2 corresponds to Algorithm 10, Implementation 3 corresponds to Algorithm 11 and Implementation 4 corresponds to Algorithm 4. Additionally tuple reservoir reduction was applied to all these implementations. The original Hadoop benchmark code is referred to as Hadoop.

Our experiments ran on (up to) 16 DAS-4 [11] nodes at Leiden University, both for our MPI-based programs and the Hadoop-based benchmark code. Each node has 2 CPU sockets, with in each a 4-core CPU with Hyper-Threading, yielding 8 physical cores and 16 virtual cores. To determine the performance characteristics of our implementations we used a number of different configurations. Those configurations varied between 1–16 nodes and 1–8 threads per node. For datasets of with 2^{27} vertices or more, only configurations with 16 nodes were evaluated. The Hadoop benchmark was only run in one configuration, on all 16 nodes with 16 reducers, the recommended settings by the BigDataBench benchmark.

5.1 Results

In Figures 1 and 2 the runtimes on the 16×4 and 16×8 configurations are depicted. These configurations were chosen because they resulted in the best runtimes for most implementations. Figure 1 shows that all implementations have roughly the same runtime, with the exception of Implementation 4. This shows that the transformations that were applied on the other implementations to derive more efficient data structures are very beneficial.

Also on Figure 2 Implementation 1 and Implementation 2 are closely tied. However, contrary to what is seen in Figure 1, Implementation 3 outperforms the other implementations in various cases. This is because Implementation 3 more effectively uses the extra cores because its threads have a smaller memory footprint. The other implementations reach peak performance on the 16×4 configuration.

In addition to raw performance we are particularly interested in the scalability of the implementations that were generated from our forelem specification of PageRank. For every number of parallel threads except 128, Figure 3 shows the runtime of all of the generated implementations for the dataset with 2^{26} vertices. In case multiple configurations yielded the same number of threads, for example 2×8 (2 nodes with 8 threads each), 4×4 and 8×2, the configuration with the fewest nodes was chosen, in this case 2×8. As can be seen from Figure 3, the reduction in execution time is linear with respect to the number of threads, i.e. for Implementation 2 the execution time going 16 to 32 threads resulted in a

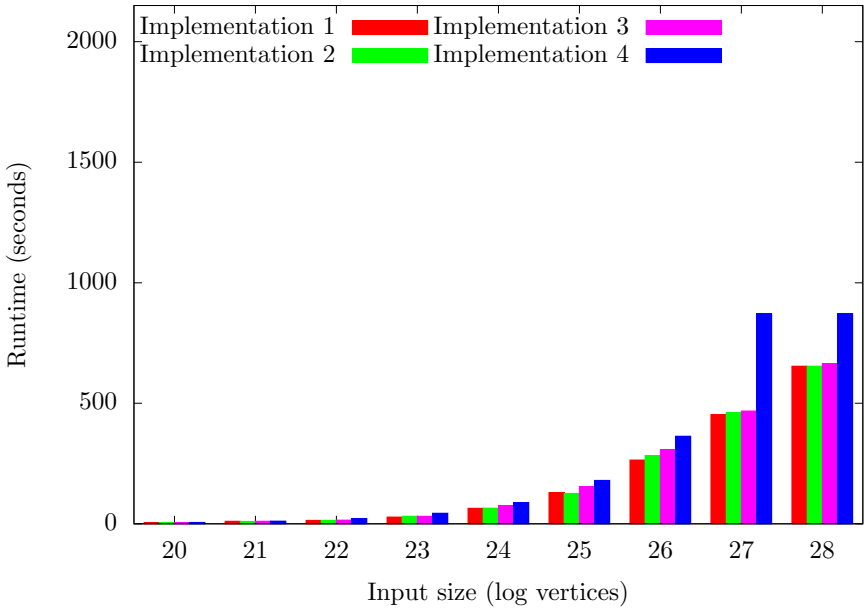

Figure 1: The runtime of the various implementations on the 16×4 configuration. Hadoop was left out to improve legibility.

20% reduction and going from 32 to 64 threads a reduction of 25%, see Table 1 for detailed information on the execution times. Like the previous figures, this figure shows that Implementation 1 and Implementation 2 match each other's performance very closely.

To compare the performance of our derived forelem-based implementations with the original Hadoop implementation from BigDataBench, we show the speedup of the various implementations compared Hadoop in Figure 4. As Hadoop was unrestricted in the number of cores it could use on the 16 nodes, we show the speedup achieved on the 16×8 configuration. The figure clearly shows that all forelem-based implementations outperform the original benchmark implementation. As expected from the previous results, the performance of Implementations 1, 2 and 3 are closely tied, whereas Implementation 4 performs the worst. The minimum speedup achieved by Implementations 1, 2 and 3, at 2^{27} vertices, is approximately a factor 60. Interestingly, for even larger datasets the speedup of these three implementations starts to increase again. It is likely that this is caused by the fact that for larger datasets, the I/O performed by Hadoop to write intermediate results to disk is becoming a larger and larger bottleneck.

The decrease in speedup that is seen from an input size of 2^{20} to 2^{23} for all implementations demands an explanation. We have observed that for smaller datasets Hadoop is not able to fully exploit the available parallelism in the

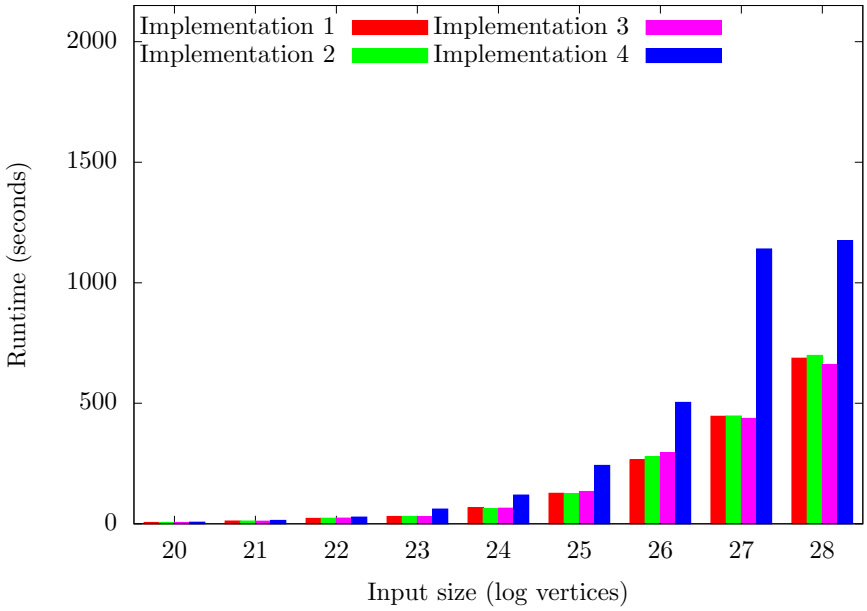

Figure 2: The runtime of the various implementations on the 16×8 configuration.

system. For example, even though 16 nodes are available, Hadoop only performs the mapping task on 7–9 nodes. The forelem-based implementations always use all 16 nodes, even for smaller datasets, resulting in significantly faster runtimes. As the size of the input grows, Hadoop is able to make more efficient use of all available 16 nodes, catching up slightly.

Table 1 contains a summary of all results. From 2^{22} vertices onwards, most of the best solutions use 8 threads per node, and all of them use 16 nodes. In all cases, the 16×8 configuration either performs best, or is very close to the best-performing configuration. As Figure 3 hints at, at high thread counts the difference becomes small, and other factors can come into play. One possible cause for the outliers is partitioning differences: not every partitioning of vertices requires the same amount of communication.

The worst configurations are slightly more regular, in all cases using a single thread per node. Since intra-node communication is faster than inter-node communication, this is in line with our expectations. Unlike the best configurations, there is no point where 1×1 becomes consistently the worst configuration possible. Once again, this could be caused by partitioning differences resulting in large amounts of inter-node communication, negating the increased available computing power.

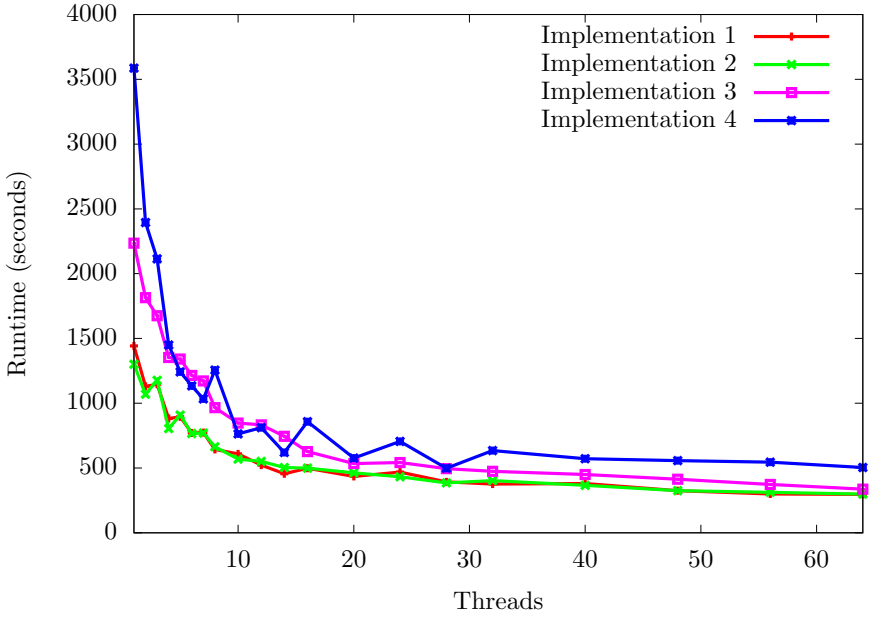

Figure 3: The runtime of each implementation for a given number of threads.

6 Conclusion

A specification of PageRank in the forelem framework has been derived and proven to be correct. Using the forelem framework 4 parallel implementations have been derived that significantly outperform an existing implementation of PageRank, that is part of a published benchmark. We showed the derived implementations to be approximately 60 times faster on the largest dataset surveyed (reducing hours to minutes of computation), and the implementations proved to be scalable.

In fact, the forelem methodology enforces the program specification to be distilled to its essence, i.e. special cases cannot easily be specified, nor complex case statements. As a consequence the programmer has to rethink the problem statement in such a way that the computation can indeed be expressed in its pure form. Having such a clean representation allows a very flexible application of various program transformations as has been demonstrated in this paper. This might very well be the underlying reason for the effectiveness of the forelem approach in deriving such performant implementations.

Of particular interest is the Tuple Reservoir Reduction transformation. In order to obtain a clean forelem representation, an ideal, albeit computationally expensive solution was needed to eliminate sink vertices. Thereupon this representation allowed the Tuple Reservoir Reduction transformation to be applied resulting in an efficient implementation. This shows that forelem has the poten-

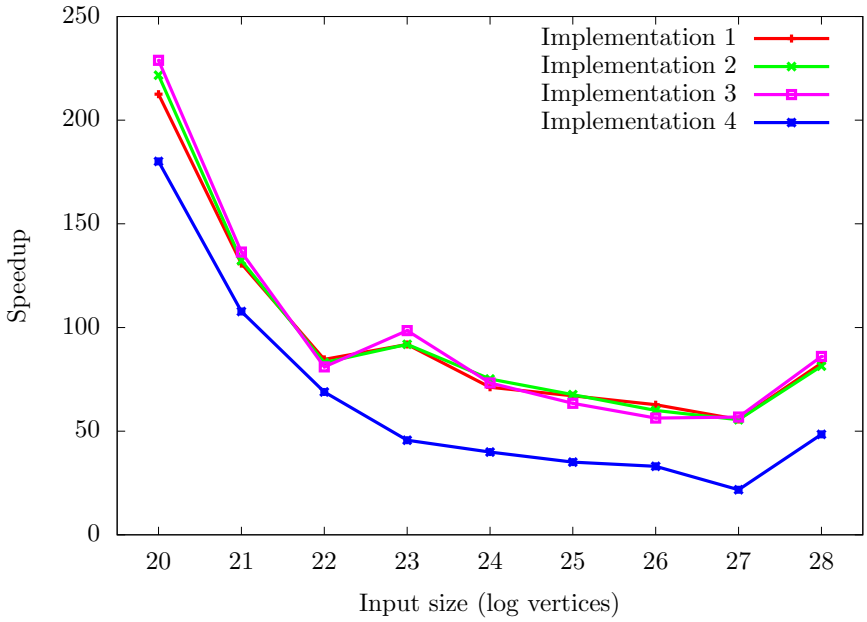

Figure 4: Speedup of the various implementations compared to Hadoop on the 16×8 configuration.

tial to turn simple, almost mathematical definitions of algorithms into programs that are competitive with human-derived implementations.

The methodology proposed by forelem and the corresponding language show their applicability to large MapReduce-like applications. The implementations were largely mechanically derived from a pre-defined set of "simple" transformations. In the future such transformations could be machine-assisted, or even fully automated.

References

[1] K. F. D. Rietveld and H. A. G. Wijshoff, "Reducing layered database applications to their essence through vertical integration," *ACM Trans. Database Syst.*, vol. 40, no. 3, p. 18, 2015. [Online]. Available: http://doi.acm.org/10.1145/2818180

[2] L. Page, S. Brin, R. Motwani, and T. Winograd, "The pagerank citation ranking: Bringing order to the web." Stanford InfoLab, Technical Report 1999–66, November 1999, previous number = SIDL-WP-1999-0120. [Online]. Available: http://ilpubs.stanford.edu:8090/422/

[3] L. Wang, J. Zhan, C. Luo, Y. Zhu, Q. Yang, Y. He, W. Gao, Z. Jia, Y. Shi, S. Zhang, C. Zheng, G. Lu, K. Zhan, X. Li, and B. Qiu, "Bigdatabench: A big data benchmark suite from internet services," in *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*, Feb 2014, pp. 488–499.

[4] U. Kang, C. E. Tsourakakis, and C. Faloutsos, "Pegasus: A peta-scale graph mining system implementation and observations," in *2009 Ninth IEEE International Conference on Data Mining*. IEEE, 2009, pp. 229–238.

[5] D. Gleich, L. Zhukov, and P. Berkhin, "Fast parallel pagerank: A linear system approach," *Yahoo! Research Technical Report YRL-2004-038, available via http://research.yahoo.com/publication/YRL-2004-038.pdf*, vol. 13, p. 22, 2004.

[6] C. Kohlschütter, P.-A. Chirita, and W. Nejdl, "Efficient parallel computation of pagerank," in *European Conference on Information Retrieval*. Springer, 2006, pp. 241–252.

[7] A. Z. Broder, R. Lempel, F. Maghoul, and J. Pedersen, "Efficient pagerank approximation via graph aggregation," *Information Retrieval*, vol. 9, no. 2, pp. 123–138, 2006.

[8] K. F. D. Rietveld and H. A. G. Wijshoff, "Towards a new tuple-based programming paradigm for expressing and optimizing irregular parallel computations," in *Computing Frontiers Conference, CF'14, Cagliari, Italy — May 20 – 22, 2014*, 2014, pp. 16:1–16:10. [Online]. Available: http://doi.acm.org/10.1145/2597917.2597923

[9] ——, "Optimizing sparse matrix computations through compiler-assisted programming," in *Proceedings of the ACM International Conference on Computing Frontiers, CF'16, Como, Italy, May 16–19, 2016*, 2016, pp. 100–109. [Online]. Available: http://doi.acm.org/10.1145/2903150.2903157

[10] S. Brin and L. Page, "The anatomy of a large-scale hypertextual web search engine," *Computer Networks and ISDN Systems*, vol. 30, no. 1, pp. 107 – 117, 1998. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S016975529800110X

[11] H. Bal, D. Epema, C. de Laat, R. van Nieuwpoort, J. Romein, F. Seinstra, C. Snoek, and H. Wijshoff, "A medium-scale distributed system for computer science research: Infrastructure for the long term," *Computer*, vol. 49, no. 5, pp. 54–63, May 2016.

Size	Implementation	Best			Worst		
		Nodes	Threads per node	Time (s)	Nodes	Threads per node	Time (s)
2^{20}	Implementation 1	8	2	3.8	2	1	7.9
	Implementation 2	5	4	3.9	1	1	8.4
	Implementation 3	16	2	4.5	1	1	10.5
	Implementation 4	16	2	5.4	2	1	19.3
	Hadoop	16	-	1277.6	16	-	1277.6
2^{21}	Implementation 1	8	4	7.6	3	1	18.5
	Implementation 2	8	4	7.8	1	1	22.4
	Implementation 3	5	8	8.0	1	1	35.2
	Implementation 4	16	2	10.6	2	1	46.7
	Hadoop	16	-	1522.7	16	-	1522.7
2^{22}	Implementation 1	16	4	14.2	1	1	51.9
	Implementation 2	16	4	14.4	2	1	51.1
	Implementation 3	16	4	15.1	1	1	87.4
	Implementation 4	16	4	21.5	1	1	143.5
	Hadoop	16	-	1927.0	16	-	1927.0
2^{23}	Implementation 1	16	4	28.1	1	1	123.0
	Implementation 2	16	8	30.5	2	1	119.1
	Implementation 3	16	8	28.4	2	1	195.8
	Implementation 4	16	4	44.0	1	1	324.9
	Hadoop	16	-	2798.2	16	-	2798.2
2^{24}	Implementation 1	16	4	64.2	1	1	267.2
	Implementation 2	16	8	63.2	1	1	295.5
	Implementation 3	16	8	64.9	1	1	508.5
	Implementation 4	16	4	88.1	1	1	729.3
	Hadoop	16	-	4753.7	16	-	4753.7
2^{25}	Implementation 1	16	8	126.7	1	1	621.6
	Implementation 2	16	8	125.7	1	1	631.9
	Implementation 3	16	8	134.0	2	1	997.0
	Implementation 4	16	4	180.2	1	1	1627.5
	Hadoop	16	-	8502.0	16	-	8502.0
2^{26}	Implementation 1	16	4	264.5	1	1	1442.8
	Implementation 2	16	8	277.8	2	1	1310.5
	Implementation 3	16	8	295.8	1	1	2235.7
	Implementation 4	16	4	363.3	1	1	3585.9
	Hadoop	16	-	16662.5	16	-	16662.5
2^{27}	Implementation 1	16	8	446.4	1	1	1868.8
	Implementation 2	16	8	447.2	1	1	1877.4
	Implementation 3	16	8	437.2	1	1	2954.9
	Implementation 4	16	4	871.0	1	1	9528.6
	Hadoop	16	-	24824.1	16	-	24824.1

Size	Implementation	Best			Worst		
		Nodes	Threads per node	Time (s)	Nodes	Threads per node	Time (s)
2^{28}	Implementation 1	16	4	654.0	2	8	2838.1
	Implementation 2	16	4	653.6	1	1	2301.9
	Implementation 3	16	8	661.2	1	1	3580.0
	Implementation 4	16	4	871.0	1	1	26415.8
	Hadoop	16	-	56867.5	16	-	56867.5

Table 1: Summary of all results.