Implementing Parallel Google Map-Reduce in Eden

Jost Berthold, Mischa Dieterle, and Rita Loogen

Philipps-Universität Marburg, Fachbereich Mathematik und Informatik
Hans Meerwein Straße, D-35032 Marburg, Germany
{berthold,dieterle,loogen}@informatik.uni-marburg.de

Abstract. Recent publications have emphasised map-reduce as a general programming model (labelled Google map-reduce), and described existing high-performance implementations for large data sets. We present two parallel implementations for this Google map-reduce skeleton, one following earlier work, and one optimised version, in the parallel Haskell extension Eden. Eden's specific features, like lazy stream processing, dynamic reply channels, and nondeterministic stream merging, support the efficient implementation of the complex coordination structure of this skeleton. We compare the two implementations of the Google map-reduce skeleton in usage and performance, and deliver runtime analyses for example applications. Although very flexible, the Google map-reduce skeleton is often too general, and typical examples reveal a better runtime behaviour using alternative skeletons.

1 Introduction

To supply conceptual understanding and abstractions of parallel programming, the notion of *algorithmic skeletons* has been coined by Murray Cole in 1989 [1]. An algorithmic skeleton abstractly describes the (parallelisable) structure of an algorithm, but separates specification of the concrete work to do as a parameter function. Skeletons are meant to offer ready-made efficient implementations for common algorithmic patterns, the specification of which remains sequential. Thus, an algorithmic skeleton contains inherent parallelisation potential in the algorithm, but this remains hidden in its implementation. A broader research community has quickly adopted and developed the idea further [2]. Skeletons are a well-established research subject in the scientific community, yet until today they have only little impact on mainstream software engineering, in comparison with other models, like MPI [3] collective operations and *design patterns* [4].

To a certain extent, MPI collective operations and algorithmic skeletons follow the same philosophy: to specify common patterns found in many applications, and to provide optimised implementations that remain hidden in libraries. However, while skeletons describe a whole, potentially complex, algorithm, collective operations only predefine and optimise common standard tasks which are often needed in implementing more complex algorithms. The design pattern paradigm

H. Sips, D. Epema, and H.-X. Lin (Eds.): Euro-Par 2009, LNCS 5704, pp. 990–1002, 2009.

has considerable potential in identifying inherent parallelism in common applications, capturing complex algorithmic structures, and providing conceptual insight in parallelisation techniques and problem decomposition. However, it often merely targets concepts and leaves implementation to established low-level libraries and languages (see e.g. textbook [5] for a typical example). Design patterns thus cannot provide the programming comfort and abstraction level of algorithmic skeletons. Moreover, collective operations and design patterns, by their very nature, are explicit about parallelism already in their specification, whereas skeletons completely hide parallelism issues.

Even more remarkable is the fact that applications from industry have meanwhile achieved the mainstream breakthrough for the skeleton idea (even though it is never called like this). In 2004, we saw the first publication which abstractly described large-scale *map-and-reduce* data processing at Google [6,7]. It was proposed as a "programming model" for large dataset processing, but in fact precisely realises the skeleton idea. A publication by Ralf Lämmel [8] points out shortcomings of the skeleton's formal specification, provides sequential Haskell implementations, and briefly discusses parallelism. Given the great acceptance that the programming model has found, and its close relation to skeleton programming, this paper investigates possible parallel implementations starting from Lämmel's Haskell code, and discusses their respective advantages and drawbacks.

From the perspective of functional languages, skeletons are specialised higher-order functions with a parallel implementation. Essentially, the skeleton idea applies a functional paradigm for coordination, independent of the underlying *computation language*. While skeleton libraries for imperative language, e.g. [9,10], typically offer a fixed, established set of skeletons, parallel functional languages are able to express new skeletons, or to easily create them by composition [11,2]. Some functional languages parallelise by pre-defined data parallel operations and skeletons, like NESL [12], OCamlP3l [13], or PMLS [11]. These fixed skeleton implementations are highly optimised and allow composition, but not the definition of new problem-specific skeletons or operations. More explicit functional coordination languages are appropriate tools not only to apply skeletons, but also for their *implementation*, allowing formal analysis and conceptual modeling. Coordination structure, programming model and algorithm structure can be cleanly separated by functional languages, profiting from their abstract, mathematically oriented nature. In our work, we use the general-purpose parallel Haskell dialect Eden [14] as an implementation language for the skeletons.

The paper is structured as follows: Section 2 explains the classical map-and-reduce skeleton defined in Eden, thereby introducing features of the language we use for our implementations. Section 3 introduces the Google map-reduce skeleton. Parallel implementation variants are discussed in Section 4. A section with measurements and analyses for some example applications follows. Section 6 considers related work, the final section concludes.

2 Parallel Transformation and Reduction in Eden

Classically, reduction over a list of elements is known as the higher-order function `fold` (from the left or from the right), and is often combined with a preceding transformation of the list elements (in other words, `map`). Denotationally, it is a composition of the higher-order functions `map` and `fold`:

```
mapFoldL :: (a -> b) -> (c -> b -> c) -> c -> [a] -> c
mapFoldL mapF redF n list = foldl redF n (map mapF list)

mapFoldR :: (a -> b) -> (b -> c -> c) -> c -> [a] -> c
mapFoldR mapF redF n list = foldr redF n (map mapF list)
```

In this general form, the folding direction leads to slightly different types of the reduction operator `redF`. Parallel implementations have to unify types `b` and `c` and require associativity to separate sub-reductions. In addition, the parameter `n` should be neutral element of `redF`. Under these conditions, the folding direction is irrelevant, as both versions yield the same result. Parallel implementations may even reorder the input, requiring the reduction operator `redF` to be commutative.

Assuming associativity and commutativity, we can easily define a parallel map-and-reduce skeleton for input streams in the functional Haskell dialect Eden, as shown in Fig.1.

```
parmapFoldL :: (Trans a, Trans b) =>
               Int ->           -- no. of processes
               (a -> b) ->      -- mapped on input
               (b -> b -> b) -> -- reduction (assumed commutative)
               b ->             -- neutral element for reduction
               [a] -> b
parmapFoldL np mapF redF neutral list = foldl' redF neutral subRs
    where sublists    = unshuffle np list
          subFoldProc = process (foldl' redF neutral . (map mapF))
          subRs       = spawn (replicate np subFoldProc) sublists

unshuffle :: Int -> [a] -> [[a]] -- distributes the input stream
unshuffle n list = ...           -- round-robin in np streams
spawn :: [Process a b] -> [a] -> [b] -- instantiates a set of processes
spawn ps inputs = ...                -- with respective inputs
```

Fig. 1. Parallel map-reduce implementation in Eden

The input stream is distributed round-robin into `np` inputs for `np` Eden processes, which are instantiated by the Eden library function `spawn`. `Process` is the type constructor for Eden Process abstractions, which are created by the function `process :: (a -> b) -> Process a b`. Type class `Trans` provides implicitly used data transfer functions. The spawned processes perform the transformation (`map`) and pre-reduce the map results (duplicating the given neutral element) by the *strict*

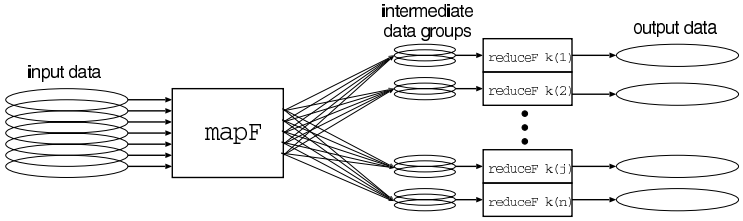


Fig. 2. Computation scheme of Google map-reduce

left-fold `foldl'`. As explained, the fold operator `redF` has got a restricted type in order to combine the subresults in a second stage, and is assumed commutative because input for the pre-reductions is distributed round-robin and streamed.

3 The "Google Map-Reduce" Skeleton

A more general variant of map-and-reduce has been proposed, as a programming model for processing large datasets, by Google personnel Jeff Dean and Sanjay Ghemawat. In January 2008, an update of the original publication (OSDI 2004 [6]) appeared in the ACM communications [7].

The intention to provide a framework which allows one "to express the simple computations [...] but hides the messy details of parallelization, fault-tolerance, data distribution, and load balancing, in a library"[6] is precisely the skeleton idea. However, the word "skeleton" does not figure in any of the two publications! Neither publication claims for the model to be new, its essential merit is that it brought the skeleton approach to industry. The model has found great acceptance as a programming model for parallel data processing (e.g. [15,16]).

The computation scheme of Google map-reduce is depicted in Fig. 2. In a nutshell, a Google map-reduce instance first transforms key/value pairs into (intermediate) other key/value pairs, using a `mapF` function. After this, each collection of intermediate data *with the same key* is reduced to one resulting key/value pair, using a `reduceF` function. In-between the transformation and the reduction, the intermediate data is grouped by keys, so the whole computation has two logical phases. The parallelisation described in the original work [6] imposes additional requirements on the applied parameter functions; which, on the other hand, have more liberal type constraints than what `map` and `foldl` would require. Ralf Lämmel, in his related publication [8], captures them in a formal specification derived from the original examples and description, using Haskell.[1]

```
gOOGLE_MapReduce :: forall k1 k2 v1 v2 v3. Ord k2 => -- for grouping
         (k1 -> v1 -> [(k2,v2)])      -- 'map' function, with keys 1
         ->  (k2 -> [v2] -> Maybe v3) -- 'reduce' function, can use key 2
         ->  Map k1 v1                -- A key to input-value mapping
         ->  Map k2 v3                -- A key to output-value mapping
```

[1] The code is provided online by Lämmel, so we do not reproduce it here, see *http://www.cs.vu.nl/~ralf/MapReduce/*

The 'map' function takes an input pair of type `(k1,v1)` (`k1` could actually be subsumed under v1 by pairing) and may produce a whole *list* of intermediate key-value pairs `[(k2,v2)]` from it, which are then grouped by key (the ordering constraint enables more efficient data structures – an equality constraint `Eq k2` would suffice). Each of the value lists for a key is then processed by the 'reduce' function to yield a final result of type `v3` for this key, or no output (thereby the `Maybe`-type). Unlike the usual `fold`, output does not necessarily have the same type as the intermediate value (but typically `v2` and `v3` are of the same type). So the parameter function of `fold` in the Google publications is not properly a function which could be the argument to a fold (i.e. reduce) operation, nor is it always a reduction in the narrow sense. Additionally, as Lämmel points out, the original paper confuses lists and sets: The input to a skeleton instance is neither a set, nor a list, but a finite mapping from keys to values, where duplicate values are not allowed for the same key. And likewise, the output of the skeleton conceptually does not allow the same key to appear twice.

Examples: A classical combination of `map` and `foldl` can be expressed as a special case of the more general skeleton. The map function here produces singleton lists and assigns a constant intermediate key 0 to every one. The reduction function ignores these keys, and left-folds the intermediate values as usual.

```
mapfold :: (a -> b) -> (b -> b -> b) -> b -> [a] -> b
mapfold mapF redF neutral input = head (map snd (toList gResult))
    where mapF' _   x  = [(0,mapF x)]
          redF' _ list = Just (foldl' redF neutral list)
          gResult      = gOOGLE_MapReduce mapF' redF'
                                  (fromList (zip (repeat 0) input))
```

A more general example, often given in publications on Google map-reduce, is to compute how many times certain words appear in a collection of web pages. The input is a set of pairs: web page URLs and web page content (and the URL is completely ignored). The 'map' part retrieves all words from the content and uses them as intermediate keys, assigning constant 1 as intermediate value to all words. Reduction sums up all these ones to determine how many times a word has been found in the input.

```
wordOccurrence = gOOGLE_MapReduce toMap forReduction
  where toMap        :: URL -> String -> [(String,Int)]
        toMap url content = zip (words content) (repeat 1)
        forReduction :: String -> [Int] -> Maybe Int
        forReduction word counts = Just (sum counts)
```

A range of other, more complex applications is possible, for instance, iteratively clustering large data sets by the k-means method, used as a benchmark in two recent publications [15,16]. We will discuss this benchmark in Section 5.

4 Parallel Google Map-Reduce

Google map-reduce offers different opportunities for parallel execution. First, it is clear that the map function can be applied to all input data independently.

```
-- inner interface
parMapReduce  ::  Ord k2 =>
      Int -> (k2 -> Int)           -- No. of partitions, key partitioning
      ->  (k1 -> v1 -> [(k2,v2)])  -- 'map' function
      ->  (k2 -> [v2] -> Maybe v3) -- 'combiner' function
      ->  (k2 -> [v3] -> Maybe v4) -- 'reduce' function
      ->  [Map k1 v1]              -- Distributed input data
      ->  [Map k2 v4]              -- Distributed output data
parMapReduce parts keycode mAP cOMBINER rEDUCE
 =  map (  -- parallelise in n reducers:  parmap
         reducePerKey rEDUCE       -- 7. Apply 'reduce' to each partition
       . mergeByKey )              -- 6. Merge scattered intermediate data
   . transpose                     -- 5. Transpose scattered partitions
   .  map (  -- parallelise in n=m mappers:  farm n
       map ( reducePerKey cOMBINER -- 4. Apply 'combiner' locally
             . groupByKey )        -- 3. Group local intermediate data
       . partition parts keycode   -- 2. Partition local intermediate data
       . mapPerKey mAP )           -- 1. Apply 'map' locally to each piece
```

Fig. 3. Parallel Google map-reduce skeleton, following Lämmel [8] (we have added the parallelisation annotations in bold face)

Furthermore, since reduction is done for every possible intermediate key, several processors can be used in parallel to reduce the values for different keys. Additionally, the mapper processes in the implementation perform pre-grouping of intermediate pairs by (a hash function of) intermediate keys. Usually, implementations strictly split the whole algorithm in two phases. The productive implementation described in [7] is based on intermediate files in Google's own shared file system GFS. Pre-grouped data is periodically written to disk, and later fetched and merged by the reducer tasks before they start reduction of values with the same key. This makes it possible to reassign jobs in case of machine failures, making the system more robust. Furthermore, at the end of the map phase, remaining map tasks are assigned to several machines simultaneously to compensate load imbalances.

Following Lämmel's specification. To enable parallel execution, Lämmel proposes the version shown in Fig. 3. Interface and functionality of the Google map-reduce skeleton are extended in two places.

First, input to the map function is grouped in bigger "map jobs", which allows to adapt task size to the resources available. For instance, the job size can be chosen appropriately to fit the block size of the file system. For this purpose, the proposed outer interface (not shown) includes a size parameter and an estimation function. The skeleton input is sequentially traversed and partitioned into tasks with estimated size close to (but less than) the desired task size.

Second, *two* additional pre-groupings of equal keys are introduced, one for a pre-reduction in the mapper processes, and one to aggregate input for the reducer processes. The map operation receives a task (a set of input pairs), and

produces a varying number of intermediate output. This output is sorted using intermediate keys, and the map processes pre-group output with the same key, using the added parameter function `cOMBINER`. In many cases, this combiner will be the same function as the one used for reduction, but in the general case, its type differs from the `rEDUCE` function type. To reduce the (potentially unbounded) number of intermediate keys, these intermediate (pre-reduced) results are then partitioned into a fixed number of key groups for the reducer processes, using two additional skeleton parameters. The parameter `parts` indicates how many partitions (and parallel reducer processes) to use, and the function `keycode` maps (or: is *expected* to map; the code in [8] does not check this property) each possible intermediate key to a value between 1 and `parts`. This mimics the behaviour of the productive Google implementation, which saves partitioned data into n intermediate files per mapper.

Our parallel straightforward implementation of the skeleton consists of replacing the *map* calls in the code (see Fig. 3) by appropriate parallel `map` implementations. The number of reducers, n, equals the number of `parts` into which the hash function `keycode` partitions the intermediate keys. A straightforward parallel `map` skeleton with one process per task can be used to create these n reducer processes. An implementation which verbally follows the description should create `m` mapper processes, which process a whole stream of input tasks each. Different Eden skeletons realise this functionality: a `farm` skeleton with static task distribution, or a dynamically balanced `workpool` [14]. However, the interface proposed by Lämmel lacks the `m` parameter, thus our parallelisation simply uses as many mappers as reducer processes, $n = m$.

An optimised implementation. A major drawback of this straightforward version, directly derived from Lämmel's code [8], is its strict partitioning into

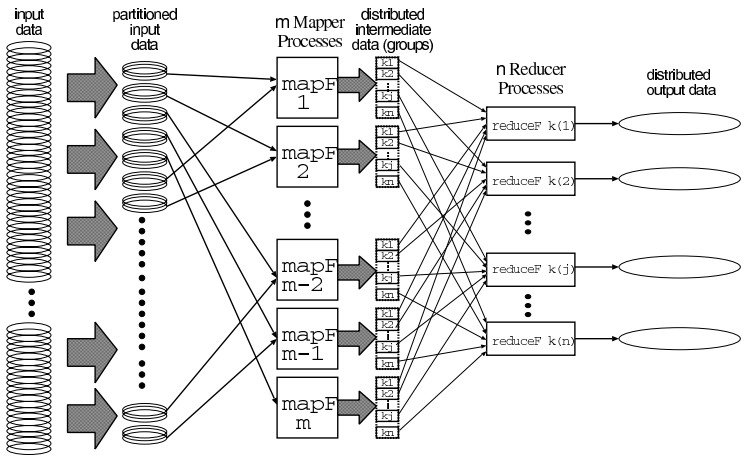


Fig. 4. Parallel Google map-reduce using distributed transpose functionality

the map phase and the reduce phase, and the call to `transpose` in between. In the Eden implementation suggested in Fig 3, all intermediate data produced by the mapper processes is sent back to the caller, to be reordered (by `transpose`) and sent further on to the reducer processes.

Our optimised implementation uses direct stream communication between mappers and reducers, as depicted in Fig. 4. Furthermore, instances of mapper and reducer are gathered in one process, which saves some communication (not shown). In order to *directly* send the respective parts of each mapper's output to the responsible reducer process via channels, a unidirectional $m : n$ communication must be set up. Each process creates a list of m channels and passes them on to the caller. The latter thus receives a whole matrix of channels (one line received from each worker process) and passes them on to the workers column-wise. Intermediate data can now be partitioned as before, and intermediate grouped pairs directly sent to the worker responsible for the respective part.

Google's productive implementation realises this $m : n$ communication by shared files. The whole data subset processed by one mapper is pre-grouped into buckets, each for one reducer process, and written to a distributed mass storage system (GFS), to be fetched by reducer processes later. While this is clearly essential for fault tolerance (in order to restart computations without data being lost in failing machines), we consider accumulating all intermediate data on mass storage a certain disadvantage in performance and infrastructure requirements.

5 Measurements for Example Applications

Example applications of Google map-reduce can be taken from literature [15,16], which, however, tend to apply very simple reduce functions and can be realised using the elementary map-reduce without keys as well. We have chosen two example programs with non-trivial key-based reduction from literature, after comparing performance for a simple map-fold computation. We have tested:

- a simple map-fold computation (sum of Euler totient values),
- the NAS-EP benchmark (using key-based reduction),
- the K-Means implementation (using key-based reduction)

on a Beowulf cluster[2] with up to 32 Intel Pentium 4 SMP processor elements (PEs) running at 3 GHz with 512 MB RAM and a Fast Ethernet interconnection. Trace visualisations show activity profiles of the PEs (y-axis) over time (x-axis) in seconds.

Sum of Euler totient values. The sumEuler program is a straightforward map-fold computation, summing up values of the Euler function $\varphi(k)$. $\varphi(k)$ tells how many $j < k$ are relatively prime to k, and the test program

[2] At Heriot-Watt University Edinburgh.

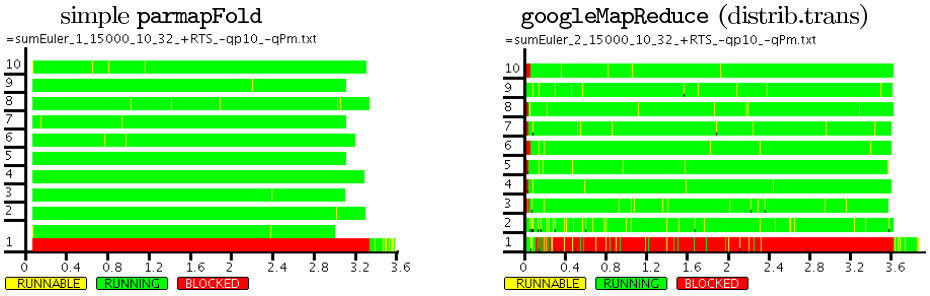


Fig. 5. Sum (reduction) of Euler totient (map) applications, input $15k$

computes it naïvely instead of using prime factorisation. So, the program computes $\sum_{k=1}^{n} \varphi(k) = \sum_{k=1}^{n} |\{j < k \mid gcd(k,j) = 1\}|$, or in Haskell syntax:

```
result = foldl1 (+) (map phi [1..n])
phi k = length (filter (primeTo k) [1..(k-1)])
```

Fig. 5 shows runtime traces for two versions: the straightforward parallel map-fold skeleton and the Google map-reduce version with distributed transposition.

Both programs perform well on our measurement platform. The Google map-reduce implementation suffers from the overhead of distributing the map tasks (which is almost entirely eliminated in the map-fold version), whereas the other version obviously exposes uneven workload due to the static task distribution.

NAS-EP benchmark. NAS-EP (Embarrassingly Parallel) [17] is a transformation problem where two-dimensional statistics are accumulated from a large number of Gaussian pseudo-random numbers, and one of only few problems which profit from the per-key reduction functionality provided by the Google map-reduce skeleton (keys indicating the 10 different square annuli). Fig. 6 shows the results, using worker functions for reduced input data size, and the skeleton version with distributed transposition. Workload distribution is fair, and the system profits from Eden's stream functionality to finish early in the reducer processes.

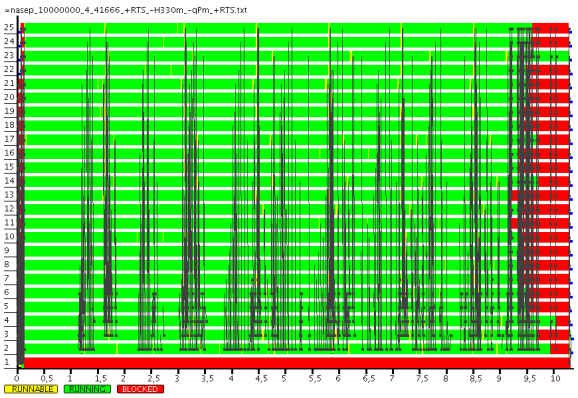


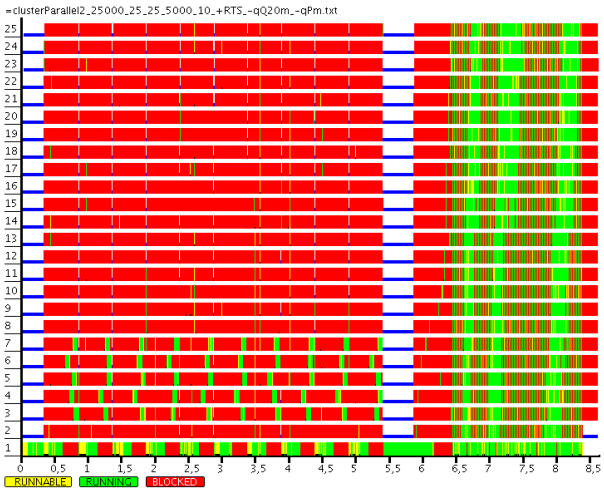
Fig. 6. NAS-EP benchmark, input 10^8

Parallel k-means. The final example, k-means, illustrates that the additional reduction using the `cOMBINE` function (as opposed to simply applying `rEDUCE`

```
mAP ::  Int -> Vector -> [(Int,Vector)]
mAP _ vec = [(1+minIndex (map (distance vec) centroids),vec)]

cOMBINE  :: Int -> [Vector] -> Maybe (Int,Vector)
cOMBINE _ vs = Just (length vs, center vs)

rEDUCE :: Int -> [(Int,Vector)] -> Maybe (Int,Vector)
rEDUCE _ vs = Just (round w,v)
    where vs'  = map (\(k,v) -> (fromIntegral k,v)) vs ::[(Double,Vector)]
          (w,v) = foldl1' combineWgt vs'

combineWgt :: (Double,Vector) -> (Double,Vector) -> (Double,Vector)
combineWgt (k1,v1) (k2,v2) = (k1+k2,zipWith (+) (wgt f v1) (wgt (1-f) v2))
       where {f = 1/(1+(k2/k1)); wgt x = map (*x) }
```

Fig. 7. Parameter functions for the parallel k-means algorithm

twice) is necessary, but might render algorithms more complicated. It also shows that Google map-reduce, however expressive, can turn out a suboptimal choice. The input of the k-means benchmark is a collection of data vectors (and arbitrary irrelevant keys). A set of k cluster centroids is chosen randomly in the first iteration. Parameterising the function to map with these centroids, the map part computes distances from the input vector to all centroids and yields the ID of the nearest centroid as the key, leaving the data as the value. The reduction, for each centroid, computes the mean vector of all data vectors assigned to the respective cluster to yield a set of k new cluster centroids, which is used in the next iteration, until the cluster centroids finally converge to desired precision.



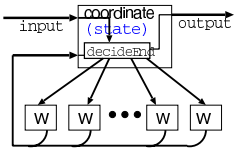


Fig. 8. Iteration skeleton

Fig. 9. k-means algorithm, Google map-reduce followed by iteration version

Fig. 7 shows the parameter functions for implementing k-means with Google map-reduce. Instead of computing sub-centroids by a simple sum, the vector subsets are precombined to a sub-centroid with added weight (`cOMBINE` function), to avoid numeric overflows and imprecision. The final reduction `rEDUCE` then uses the weight, which indicates the (arbitrary) number of vectors assigned to the sub-centroid, and combines centroids by a *weighted* average.

However, k-means is not suitable for the Google map-reduce skeleton on distributed memory machines. The problem is, the algorithm works iteratively in fixed steps, and amounts to setting up a new Google map-reduce instance for each iteration. These globally synchronised iteration steps and the skeleton setup overhead dominate the runtime behaviour, especially because the tasks are data vectors, sent and received again in each step. An iteration skeleton (as depicted in Fig. 8) should be used instead, with the advantage that the data vectors become initialisation data for the skeleton. The execution trace in Fig. 9 shows both versions for 25000 data vectors: first 10 iterations of the Google map-reduce version, then a version using an iteration skeleton (which performs around 90 iterations in shorter time). Communication of the data vectors completely eats up parallel speedup, whereas they are initialisation data in the second version.

6 Related Work

We have already cited the basic references for the skeleton in question throughout its description and in the introduction. Originally published in 2004 [6], the Google employees Dean and Ghemawat have updated their publication recently with ACM [7], providing more recent figures of the data throughput in the productive implementation. The Hadoop project [18] provides an open-source java implementation. A more thorough description of its functionality is provided by Ralf Lämmel [8], first published online in 2006.

Both the original description by the Google authors and Ralf Lämmel discuss inherent parallelism of the Google map-reduce skeleton. While Lämmel presents substantial work for a sound understanding and specification of the skeleton, his parallelisation ideas remain at a high level, at times over-simplified, and he does not discuss any concrete implementation. The original Google work restricts itself to describing and quantifying the existing parallelisation, but gives details about the physical setup, the middleware in use, and error recovery strategies.

Several publications have adopted and highlight Google map-reduce, with different focus. An evaluation in the context of machine-learning applications can be found in [15]. Ranger et al. [16] present an implementation framework for Google map-reduce for multicores, *Phoenix*. They report superlinear speedups (ranges up to 30) on a 24-core machine, achieved by adjusting the data sizes to ideal values for good cache behaviour. Other authors have recognised the advantages of the high-level programming model and propose it for other custom architectures: Cell [19] and FPGAs [20].

7 Conclusions

A comprehensive overview of Google map-reduce and its relation to algorithmic skeletons has been given. We have discussed two implementations for Google map-reduce, one following earlier work, and an optimised Eden version. As our runtime analyses for some example applications show, the skeleton implementation delivers good performance and is easily applicable to a range of problems. Implementations using explicitly parallel functional languages like Eden open the view on computation structure and synchronisation, which largely facilitates skeleton customisation and development.

The popularity of Google map-reduce nicely shows the ease and flexibility of skeletons to a broader audience; and has found good acceptance in mainstream development. On the other hand, the popularity of just one skeleton may sometimes mislead application development, not considering alternative skeletons. It turns out that the full generality of the Google map-reduce skeleton is often not needed, and other skeletons are more appropriate. Nevertheless, we consider Google map-reduce a big step for skeleton programming to finally get adequate attention, as a mathematically sound high-level programming model for novel parallel architectures.

Acknowledgements: We thank the anonymous referees, Phil Trinder, Kevin Hammond and Hans-Wolfgang Loidl, for helpful comments on earlier versions.

References

1. Cole, M.I.: Algorithmic Skeletons: Structured Management of Parallel Computation. Research Monographs in Parallel and Distributed Comp. MIT Press, Cambridge (1989)
2. Rabhi, F.A., Gorlatch, S. (eds.): Patterns and Skeletons for Parallel and Distributed Computing. Springer, Heidelberg (2003)
3. MPI-2: Extensions to the message-passing interface. Technical report, University of Tennessee, Knoxville (July 1997)
4. Gamma, E., Helm, R., Johnson, R.E., Vlissides, J.M.: Design patterns: Abstraction and reuse of object-oriented design. In: Nierstrasz, O. (ed.) ECOOP 1993. LNCS, vol. 707, pp. 406–431. Springer, Heidelberg (1993)
5. Mattson, T.G., Sanders, B.A., Massingill, B.L.: Patterns for Parallel Programming. SPS. Addison-Wesley, Reading (2005)
6. Dean, J., Ghemawat, S.: Mapreduce: Simplified data processing on large clusters. In: OSDI 2004, Sixth Symp. on Operating System Design and Implementation (2004)
7. Dean, J., Ghemawat, S.: Mapreduce: simplified data processing on large clusters. Communications of the ACM 51(1), 107–113 (2008)
8. Lämmel, R.: Google's MapReduce programming model - Revisited. Science of Computer Programming 70(1), 1–30 (2008)
9. Poldner, M., Kuchen, H.: Scalable farms. In: Proceedings of ParCo 2005. NIC Series, vol. 33 (2005)
10. Benoit, A.: ESkel — The Edinburgh Skeleton Library. University of Edinburgh (2007), `http://homepages.inf.ed.ac.uk/abenoit1/eSkel/`

11. Michaelson, G., Scaife, N., Bristow, P., King, P.: Nested Algorithmic Skeletons from Higher Order Functions. Parallel Algorithms and Appl. 16, 181–206 (2001)
12. Blelloch, G.: Programming Parallel Algorithms. Communications of the ACM 39(3), 85–97 (1996)
13. Danelutto, M., DiCosmo, R., Leroy, X., Pelagatti, S.: Parallel functional programming with skeletons: the OCamlP3L experiment. In: ACM workshop on ML and its applications (1998)
14. Loogen, R., Ortega-Mallén, Y., Peña-Marí, R.: Parallel Functional Programming in Eden. Journal of Functional Programming 15(3), 431–475 (2005)
15. Chu, C.T., Kim, S.K., Lin, Y.A., Yu, Y., Bradski, G., Ng, A.Y., Olukotun, K.: Map-reduce for machine learning on multicore. In: Schölkopf, B., Platt, J., Hoffman, T. (eds.) Advances in Neural Information Processing Systems, vol. 19. MIT Press, Cambridge (2007)
16. Ranger, C., Raghuraman, R., Penmetsa, A., Bradski, G., Kozyraki, C.: Evaluating mapreduce for multi-core and multiprocessor systems. In: HPCA 2007: Int. Symp. on High Performance Computer Architecture, pp. 13–24. IEEE Computer Society, Los Alamitos (2007)
17. Bayley, D., Barszcz, E., et al.: NAS Parallel Benchmarks. Technical Report RNR-94-007, NASA Advanced Supercomputing (NAS) Division (1994)
18. Apache Hadoop Project. Web page, `http://hadoop.apache.org`
19. de Kruijf, M., Sankaralingam, K.: MapReduce for the Cell B.E. Architecture. Technical Report TR1625, Computer Sc.Dept., Madison, WI (2007)
20. Yeung, J.H., Tsang, C., Tsoi, K., Kwan, B.S., Cheung, C.C., Chan, A.P., Leong, P.H.: Map-reduce as a programming model for custom computing machines. In: Proc. IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM). IEEE Computer Society, Los Alamitos (2008)