Tim Kiefer, Peter Benjamin Volk, Wolfgang Lehner

Pairwise Element Computation with MapReduce

Erstveröffentlichung in / First published in:

HPDC '10: The 19th International Symposium on High Performance Distributed Computing. Chicago Illinois, 2010. S. 826-833. ACM. ISBN 978-1-60558-942-8

DOI: https://doi.org/10.1145/1851476.1851595

Final edited form was published in "HPDC '10: The 19th International Symposium on High Performance Distributed Computing. Chicago Illinois, 2010". S. 826-833. ISBN 978-1-60558-942-8
https://doi.org/10.1145/1851476.1851595

Pairwise Element Computation with MapReduce

Tim Kiefer, Peter Benjamin Volk, Wolfgang Lehner
Dresden University of Technology
Database Technology Group
Dresden, Germany
{tim.kiefer, peter_benjamin.volk, wolfgang.lehner}@tu-dresden.de

ABSTRACT

In this paper, we present a parallel method to evaluate functions on pairs of elements. It is a challenge to partition the Cartesian product of a set with itself in order to parallelize the function evaluation on all pairs. Our solution uses (a) replication of set elements to allow for partitioning and (b) aggregation of the results gathered for different copies of an element. Based on an execution model with nodes that execute tasks on local data without online communication, we present a generic algorithm and show how it can be implemented with MapReduce. Three different distribution schemes that define the partitioning of the Cartesian product are introduced, compared, and evaluated. Any one of the distribution schemes can be used to derive and implement a specific algorithm for parallel pairwise element computation.

Categories and Subject Descriptors

D.1 [**Programming Techniques**]: General; F.2 [**Analysis of Algorithms and Problem Complexity**]: General

General Terms

Algorithms

Keywords

MapReduce, parallel pairwise computation, Cartesian product partitioning, distribution scheme

1. INTRODUCTION

To compute a function on all pairs of elements in a given set, as it is shown in Figure 1 for the function $\mathrm{comp}(-\mathrm{ute})$, is a familiar task. It is a basic building block in many algorithms throughout a wide range of applications. For example, clustering algorithms like DBSCAN [8] group elements based on their similarity. A distance computed between any two elements is used to decide whether a pair of elements is similar. Likewise, cross-document co-referencing of websites or documents tries to determine whether two mentions of entities refer to the same person [9]. Complex operations on pairs of documents are required to compute a complete cross-reference in a set of documents. In bioinformatics, comparing the mutual information of all pairs of genes from gene expression micro-arrays is a necessary first step for reconstructing gene regulatory networks [5, 11]. At last, the computation of the covariance matrix of a matrix A requires to compute $A \times A^T$. This multiplication is a pairwise inner product on all rows of A. The covariance matrix is computed, e.g., for principal component analysis [12, 13].

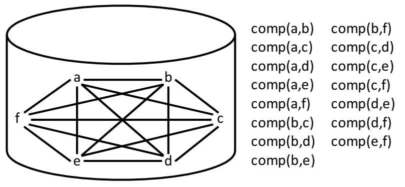


Figure 1: Pairwise element computation

The challenges of pairwise element computation rise with the complexity of the computed function or the size of the dataset, be it its cardinality or its physical size. In all cases, execution on a single machine is prohibited due to computation or memory limitations. Consequently, algorithms that leverage parallel infrastructures are needed for pairwise element computation. It is not obvious how this task, where each element needs to be processed with all other elements, can be parallelized. One way to implement such an algorithm is to use the MapReduce (MR) programming model. Together with a corresponding framework implementation, MR was introduced in 2004 [6]. The ease of programming parallel tasks offered by MR (as compared to other parallel programming models, e.g., message-based models built on MPI) and the fact that cloud infrastructures, e.g., provided by Amazon AWS [1] as Elastic Compute Cloud (EC2) or Elastic MapReduce, are available for any single user or business make the use of MR appealing.

The contribution of our work comprises two parts. First, an abstract algorithm is presented based on the underlying execution model. The algorithm outlines how pairwise element computation can be performed in parallel. We will show how all steps that need to be performed can be implemented with two consecutive MR jobs. Second, we present, discuss, and evaluate the broadcast, block, and design distribution scheme, which can be used to specify the partitioning of the Cartesian product of elements.

Throughout the paper, we assume that an appropriate function is evaluated on all pairs of elements of one set. Although it is possible to generalize some of the approaches such that elements of one set can be paired with elements of another set, we will not consider this case. We also assume that any pairwise evaluation is symmetric, i.e., $\mathrm{comp}(s_i, s_j)$ returns the same result as $\mathrm{comp}(s_j, s_i)$. Only marginal modifications of the proposed methods are needed to allow non-symmetric evaluations as well. We implemented and tested all methods based on the Hadoop MR framework, at this time available in version 0.20.1 [2].

The remainder of this paper is organized as follows. First, we discuss related work in Section 2. Then, in Section 3, we introduce the execution model that we base our algorithms on. In Section 4 our parallel solution for pairwise element computation is shown together with a possible implementation using MR. In Section 5, we present and compare the distribution schemes that define the partitioning of the Cartesian product. An evaluation of these methods based on existing cloud infrastructures follows in Section 6. We conclude our work in Section 7.

2. RELATED WORK

Elsayed et al. propose a method to compute pairwise document similarity using MR [7]. Their method first builds a reverse index from terms to documents. As a result, the set for pairwise comparison shrinks to the set of documents that contain a certain term (which usually is smaller than the initial set of documents). It is then possible to evaluate the Cartesian product of this set locally in just one mapper (per term) and to aggregate the result over multiple terms. In the given application it is possible to reduce the problem's complexity. In contrast, our work concentrates on applications where the quadratic complexity of the pairwise comparison cannot be reduced.

Some I/O optimization techniques have been proposed to work with datasets that do not fit in memory. For example, block matrix multiplication [4] allows to process datasets beyond the size of the main memory. These techniques can be used to ease the challenge of large datasets. Our work aims at datasets that exceed the limit of reasonable local processing, either by size or by computational costs.

We are not aware of any further algorithms for parallel pairwise element computation or related work dealing with the same.

3. EXECUTION MODEL

To provide premises for the execution environment and cornerstones for our algorithm, we first introduce the execution model that we base our work on. We assume a number of nodes that are connected by a (possibly slow) network. All nodes can execute tasks in parallel whereas each task processes local data. There is no online communication possible between different nodes and no shared memory can be accessed quickly by different tasks. Stored data can be transmitted via the network from one node to another node.

Furthermore, we make the following assumptions on the data that are processed. The input dataset is stored as files, distributed on the participating nodes. Random access to single elements may not be possible, as we understand our algorithm as a building block in a row of, e.g., MR jobs. Consequently, the preceding job may have written the dataset to files, where each file contains multiple records. With the same reasoning, we write the results of the computation in a distributed fashion spread among all nodes. Based on the step or application following our algorithm, the results may be aggregated. We will describe this optional aggregation in the next section.

We assume any single element to have a unique identifier. Hence, the result of a pairwise computation of elements s_1 and s_2 can be stored as the identifier of element s_2 together with the function result as shown in Figure 2. It follows that, although the results of all computations are stored together with the respective elements, the storage requirements are usually less than quadratic in the size of the input dataset. Consider the following example for clarification. Assume a dataset of 10,000 elements, 500KB each. This results in a dataset size of 5GB. Furthermore, assume an element's identifier to be 8B and the result of the computed function also to be 8B in size. It follows that after the computation, each element is about 650KB; 500KB for the element data itself and $9,999 * 16\text{B} \approx 150\text{KB}$ for the evaluation results with all other elements. The resulting dataset is about 6.5GB (instead of 50TB that would result from quadratic expansion). Some applications (like DBSCAN) may also allow to prune some results because function evaluations are only interesting if they fulfill certain requirements, e.g., a distance to be less than a threshold.

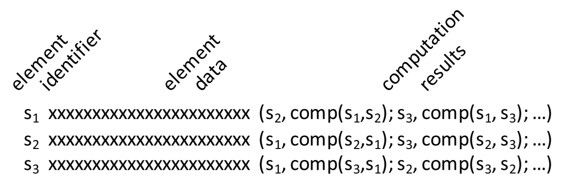

element identifier
element data
computation results
s_1 xxxxxxxxxxxxxxxxxxxxxxx (s_2, comp(s_1,s_2); s_3, comp(s_1, s_3); ...)
s_2 xxxxxxxxxxxxxxxxxxxxxxx (s_1, comp(s_2,s_1); s_3, comp(s_2, s_3); ...)
s_3 xxxxxxxxxxxxxxxxxxxxxxx (s_1, comp(s_3,s_1); s_2, comp(s_3, s_2); ...)

Figure 2: Element storage organization

4. ABSTRACT SOLUTION

The challenge of parallel pairwise element computation is how the Cartesian product of the dataset with itself can be partitioned. Based on the execution model introduced before, an algorithm comprises the following steps:

Step 1: Build subsets of the dataset and ship them to the nodes. This step defines the set of elements that a certain node works with (working set). Independent tasks (parallel computation) can only be achieved if elements can be replicated and are therefore allowed to be in different subsets. (If an element can only be in one subset, then all other elements need to be in the same subset and all computations that involve this element need to be done by one node. As soon as a second node is supposed to work at the same time, it needs at least two elements (or copies of elements) that are in the first subset already.)

Step 2: Perform pairwise element computation on all subsets in parallel. Here, each node evaluates some (or all) pairs of elements in its subset.

Step 3: Aggregate results of the comparisons per element. Based on the application, it may be necessary to collect the various copies of each element and to aggregate the results of the pairwise computations.

For a concrete solution, it needs to be decided how working sets are built and which pairs of elements of the given working set are evaluated.

Implementing the Solution with MR

The algorithm for pairwise element computation can be implemented with two MR jobs. Algorithm 1 shows the functions for distribution and pairwise computation while Algorithm 2 shows the op-

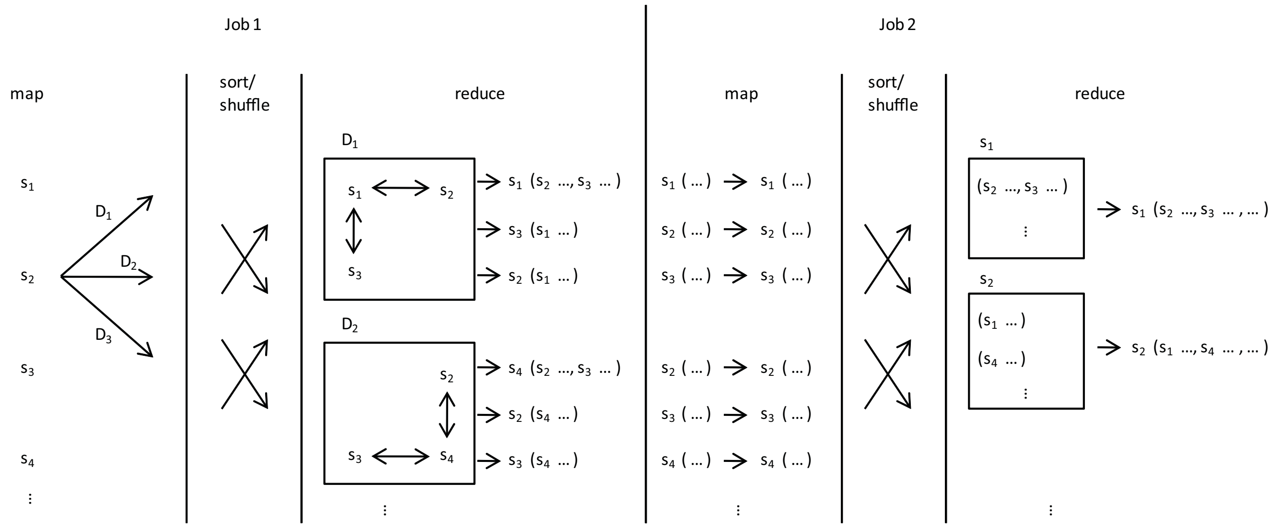


Figure 3: Flow of elements through MR jobs

tional aggregation of the results. Additionally, Figure 3 visualizes the flow of elements through both MR jobs.

The map function of the first job fulfills the first of our three steps, i.e., subsets of the dataset are built and distributed. The **getSubsets** function determines a number of working sets based on an element's identifier. In the example in Figure 3, calling getSubsets on s_2 returns D_1, D_2, and D_3. For each of these subsets, a key/value pair is emitted with the subset(-identifier) as key and the element itself as value. The sort/shuffle phase of this first MR job ensures that all elements of a certain subset are processed by one reducer. In the example, one reducer processes subset D_1 containing elements s_1, s_2, and s_3. Based on the subset(-identifier) and the elements, a list of pairs is determined in the **getPairs** function. All returned pairs of elements are evaluated and the results are stored together with the involved element. In Figure 3, in subset D_1, pairs (s_1, s_2) and (s_1, s_3) are evaluated. Although both elements, s_2 and s_3, are in subset D_1, the evaluation of this pair takes place in a different subset. The output of the reduce phase contains each element (including all copies of the element) together with the results of all pairwise computations performed in the various subsets.

The second MR job can be used to aggregate the results of various copies of an element. Since elements have been emitted with their identifiers as keys by the first job, nothing needs to be done in the map function of the second job. The grouping of elements with the same identifier happens in the sort/shuffle phase, which ensures that all copies of a certain element are processed by one reducer. Figure 3 shows how all copies of the elements s_1 are processed by one reducer. The reduce function aggregates the partial results from all copies in a way that is defined by the application (realized in the **aggregateResults** function). The result is just one element per id together with the results of all pairwise computations with all other elements.

To deduce a specific algorithm, the functions getSubsets and getPairs need to be defined. We will propose different methods in the next section. Also, depending on the application, the function aggregateResults needs to be specified.

Algorithm 1 Distribution and Pairwise Comparison

```
begin function map(id(element), element)
    $[subset] \leftarrow$ getSubsets($id(element)$)
    for all $D \in [subset]$ do
        emit($D, element$)
    end for
end function map

begin function reduce(D, [element])
    $[(i, j)] =$ getPairs($D, [element]$)
    for all $(i, j) \in [(i, j)]$ do
        $r \leftarrow$ evaluate($element_i, element_j$)
        addResult($element_i, (element_j, r)$)
        addResult($element_j, (element_i, r)$)
    end for
    for all $s \in [element]$ do
        emit($id(s), s$)
    end for
end function reduce
```

5. DISTRIBUTION SCHEMES

Formally, the problem of building subsets (step 1) and determining pairs within these subsets (step 2) can be described as follows.

Problem: Let $S = \{s_1, s_2, \ldots, s_v\}$ be a set of elements and let $\mathcal{D} = \{D_1, \ldots, D_b\}$ be a collection of distinct subsets of S, called working sets. Furthermore, let $\mathcal{P} = \{P_1, \ldots, P_b\}$ be a collection of relations (pairs), one for each working set in $\mathcal{D}$. For each relation in $\mathcal{P}$, we have $P_l \subseteq (D_l \times D_l)$ with $D_l \in \mathcal{D}$.

How can $\mathcal{D}$ and $\mathcal{P}$ be constructed, such that (a) work that is done in parallel is well balanced and (b) each pair of elements is evaluated *exactly* once among all nodes?

Both intuitive wishes lead to the formal demands that (a) all working sets are similar in size and (b) for any two elements s_i and s_j $(i > j)$ in S, there is exactly one working set D_l with pair relation P_l to fulfill: $s_i \in D_l$, $s_j \in D_l$, and $(s_i, s_j) \in P_l$.

Algorithm 2 Aggregation

```
begin function map(id(element), element)
  {do nothing}
end function map

begin function reduce(id(element), [element])
  $newElement \leftarrow$ aggregateResults($[element]$)
  emit($id(element), newElement$)
end function reduce
```

The problem can trivially be solved with $b = 1$, $D_1 = S$ and $P_1 = \{(s_i, s_j) \mid s_i, s_j \in S, i > j\}$. However, the intention is to parallelize the evaluation of all pairs of elements. Hence, we are looking for solutions where b is equal to or greater than the available number of nodes n.

A non-trivial solution for a set S that contains $v = 7$ elements is shown in Figure 4. You can see the system $\mathcal{D}$ of $b = 7$ subsets; each containing $k = 3$ elements. Together with the relations $\mathcal{P}$, it is ensured that all pairs are evaluated exactly once. The work is split into 7 independent tasks.

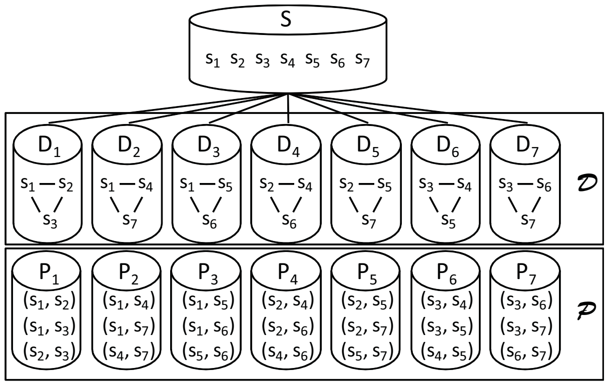

S
s1 s2 s3 s4 s5 s6 s7
D1 D2 D3 D4 D5 D6 D7
s1—s2 s3
s1—s4 s7
s1—s5 s6
s2—s4 s6
s2—s5 s7
s3—s4 s5
s3—s6 s7
𝒟
P1 P2 P3 P4 P5 P6 P7
(s1, s2) (s1, s3) (s2, s3)
(s1, s4) (s1, s7) (s4, s7)
(s1, s5) (s1, s6) (s5, s6)
(s2, s4) (s2, s6) (s4, s6)
(s2, s5) (s2, s7) (s5, s7)
(s3, s4) (s3, s5) (s4, s5)
(s3, s6) (s3, s7) (s6, s7)
𝒫

Figure 4: Example solution with systems $\mathcal{D}$ and $\mathcal{P}$

In the following subsections, we will introduce the broadcast, block, and design distribution schemes that provide constructions for $\mathcal{D}$ and $\mathcal{P}$. Afterwards, a comparison of all schemes will be given in the last subsection.

5.1 Broadcast Approach

The broadcast approach is based on the assumption that the dataset size is moderate but the function to evaluate is expensive. Then, the whole dataset can be distributed to all nodes. Hence, in the first step, each element is replicated as often as there are nodes and each subset contains all elements of the dataset; $D_1 = \ldots = D_b = S$. Depending on the application preceding our algorithm, a dataset of moderate size may already be stored redundantly on all nodes. Such a precondition can be leveraged.

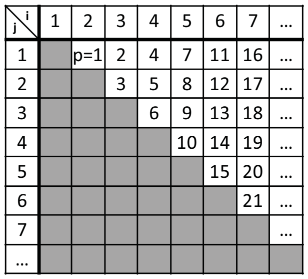
To ensure that each pair of elements is evaluated only once, the construction of $\mathcal{P}$ is crucial. All pairs are enumerated as shown in Figure 5. The upper right triangle of the matrix (dataset times dataset) is labeled, starting at $p = 1$. Depending on the number of nodes, the first one evaluates all pairs from 1 to $h = \lceil v(v-1)/2n \rceil$, where v is the cardinality of the dataset and n is the number of nodes. The second node computes the following h pairs, and so on. This ensures that all pairs are evaluated and that each node has to do approximately the same amount of work.

To derive the indexes i and j based on the label p, the following equation needs to be resolved:

$$p(i, j) = \frac{(i-1)(i-2)}{2} + j.$$

The relation P_l can then be constructed as follows:

$$P_l = \{(i(p), j(p)) \mid (l-1)h + 1 \le p \le \min(lh, v)\}.$$



j \ i	1	2	3	4	5	6	7	...
1	▨	p=1	2	4	7	11	16	...
2	▨	▨	3	5	8	12	17	...
3	▨	▨	▨	6	9	13	18	...
4	▨	▨	▨	▨	10	14	19	...
5	▨	▨	▨	▨	▨	15	20	...
6	▨	▨	▨	▨	▨	▨	21	...
7	▨	▨	▨	▨	▨	▨	▨	...
...	▨	▨	▨	▨	▨	▨	▨	▨

Figure 5: Enumeration of the distance matrix

Because the construction of $\mathcal{D}$ in the broadcast approach is simple, the MR implementation can be reduced to one job. A feature of the Hadoop MR implementation, the distributed cache, is used to broadcast the dataset to all nodes. The evaluation of pairs can then be done in the map function of the new MR job. The reduce function aggregates results as shown in Algorithm 2.

5.2 Block Approach

The block approach uses an optimized enumeration of all pairs to derive a distribution scheme. The idea is to form rectangular blocks in the upper right triangle of the matrix of pairs (dataset times dataset). All pairs in one block are evaluated by one node.

Because each node evaluates pairs in a contiguous part of the matrix, the node needs only a subset of all elements.

Each working set in $\mathcal{D}$ corresponds to one block. For example the p-th block equals $D_p = R_p \cup C_p$. In the example shown in Figure 6, the second block ($p = 2$) comprises elements $R_2 = \{s_h \mid 1 \le h \le 5\}$, i.e., rows 1 to 5, and elements $C_2 = \{s_h \mid 6 \le h \le 10\}$ contributed by columns 6 through 10.

To determine D_p, the position $(I(p), J(p))$ of the block is derived from resolving

$$p(I, J) = \frac{I(I-1)}{2} + J.$$

Then, C_p and R_p are given by

$$C_p = \{s_h \mid (I(p)-1)e + 1 \le h \le min(I(p)e, v)\} \text{ and}$$
$$R_p = \{s_h \mid (J(p)-1)e + 1 \le h \le min(J(p)e, v)\},$$

where e is the edge length of one such block ($e = 5$ in the example in Figure 6).

You can see that each element of R_p needs to be evaluated with each element of C_p. However, if a block lies on the main diagonal of the matrix ($I(p) = J(p)$), then only about half of the pairs need to be evaluated. It follows that

$$\begin{aligned} P_p &= \{(s_i, s_j) \mid s_i \in C_p, s_j \in R_p, i > j\} \\ &= \{(s_i, s_j) \mid s_i, s_j \in D_p, i > j\}. \end{aligned}$$

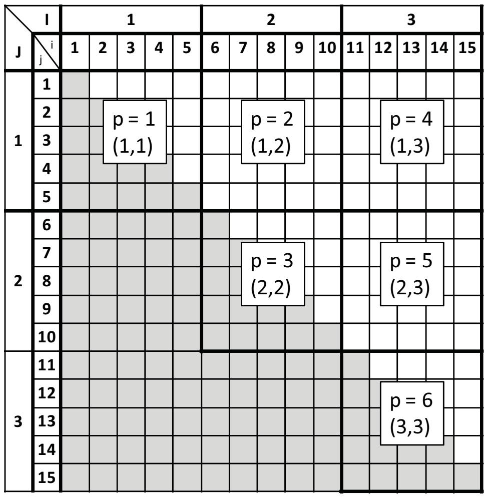


Figure 6: Enumeration of the blocks

The characteristics of the block scheme can be quantified. With h being the blocking factor and each block being of size $e \times e$ with $e = \lceil v/h \rceil$ ($e = 5$ and $h = 3$ in the example in Figure 6), it follows that each node works with $2e$ elements and performs at most e^2 evaluations. Each element is used in h different blocks. The same holds for blocks on the main diagonal of the matrix if always two such diagonal blocks are processed together.

5.3 Design Approach

The third approach uses techniques common in Algebra, more precisely the vast research area of combinatorial designs [3], to derive a distribution scheme. A $(v, k, 1)$-design is defined as follows.

DEFINITION 1 ($(v, k, 1)$-DESIGN). *Let $S = \{s_1, s_2, \ldots, s_v\}$. A collection $\mathcal{D}$ of distinct subsets of S is called a $(v, k, 1)$-design if $2 \leq k < v$, and*

1. *each set in $\mathcal{D}$ contains exactly k elements*
2. *each 2-element subset of S is contained in exactly **one** of the sets in $\mathcal{D}$ (this accounts for the 1 in $(v, k, 1)$-design)*

The sets of $\mathcal{D}$ are called blocks ($|\mathcal{D}| = b$).

In contrast to our problem, a $(v, k, 1)$-design requires each block in $\mathcal{D}$ to contain *exactly* k elements. Our less strict demand is to have blocks of *similar* size.

Figure 7 shows a simple $(7, 3, 1)$-design (vertexes are elements of S; lines that connect multiple vertexes represent subsets). The design in Figure 7 actually shows the same system of subsets as is shown in Figure 4.

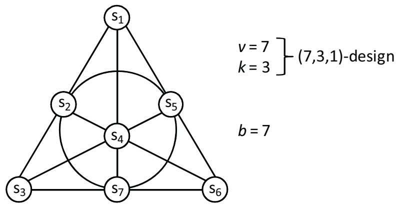


Figure 7: Example showing a $(7, 3, 1)$-design

There are proven necessary conditions for the existence of designs. These conditions show that combinations of v and k can be found such that no $(v, k, 1)$-design exists. Moreover, it is in most cases not easily possible to decide whether or not a design with a given set of parameters exists. There are, however, results for special combinations of parameters where (a) it has been proved that a design exists and (b) the design can be constructed easily. The following definition and theorem build the basis for the existence of the required design [3].

DEFINITION 2 (PROJECTIVE PLANE). *An $(m^2+m+1, m+1, 1)$-design is called a (finite) projective plane of order m.*

THEOREM 1. *From [3, Theorem 2.32, page 17]: For every prime power q there exists a projective plane of order q, that is to say, a $(q^2 + q + 1, q + 1, 1)$-design.*

For projective planes, q needs to be a prime power and a given $v = q^2 + q + 1$ determines $k = q + 1$. In general, it is not the case that the number of elements can be written as $\hat{q} = q^2 + q + 1$ where q is a prime power. However, this can be eased because the problem at hand is less strict than the formal algebraic problem. A weaker solution of a *design-like* collection of distinct subsets can be constructed where each subset may contain fewer than k elements. The basis for this solution is the projective plane of the smallest prime q such that $\hat{q} \geq v$.

An algorithm to construct projective planes was proposed by Lee et al. [10]. The construction of $\mathcal{D}$ can be derived from Theorem 2.

THEOREM 2. *Let S be the dataset of v elements and q be a prime power ($v = q^2 + q + 1$). Then, $\mathcal{D}$ defined as follows forms a $(q^2 + q + 1, q + 1, 1)$-design.*

1. *For $i = 1$:*
   $D_i = \{s_j \mid 1 \leq j \leq q + 1\}$
2. *For $1 < i \leq q + 1$:*
   $D_i = s_1 \cup \{s_j \mid q(i - 1) + 2 \leq j \leq qi + 1\}$
3. *For $q + 1 < i \leq q^2 + q + 1$:*
   Let $h = \left\lfloor \frac{i-2}{q} \right\rfloor - 1$ and $l = (i - 2) \mod q$
   $D_i = s_{h+2} \cup \{s_{q(m+1)+((l-hm)(\mathrm{mod}\, q)+2)} \mid 0 \leq m \leq q - 1\}$

The proof follows immediately from the proof of Theorem 4 in [10].

If $v < \hat{q}$, then the elements $s_{v+1}, \ldots, s_{\hat{q}}$ do not exist. From the construction rules in Theorem 2 it follows that the number of elements in the majority of working sets from D_{q+2} to $D_{\hat{q}}$ still contain about the same number of elements (with a difference of at most 1). The working sets D_1 to D_{q+1} contain either $q + 1$ elements or just one element (and can therefore be dropped). Because we assume v to be large, it follows that the main characteristics of this method are dominated by the majority of working sets that contain about the same number of elements ($\approx \sqrt{v}$). If, e.g., $v = 10,000$, then $q = 101$; hence, the first $q + 1 = 102$ working sets are dominated by the following $10,201$ ($q + 2, \ldots, \hat{q}$) working sets.

The pair relation $\mathcal{P}$ for the design approach is the full relation of all elements in each working set with duplicates eliminated that are introduced by the symmetry of the evaluation. Hence, for all $1 \leq l \leq b$:

$$P_l = \{(s_i, s_j) \mid s_i, s_j \in D_l, i > j\}.$$

5.4 Comparison of Distribution Schemes

The main characteristics of each distribution scheme depend on multiple parameters, some of which are fixed by the application or the environment. The flexible parameters can be used to influence certain properties of a method. Before we will name the characteristics that we use to compare the methods, we describe their parameters.

All distribution schemes have the following three parameters in common:

v	number of elements in the dataset;
n	number of nodes;
p	number of tasks.

The first two parameters are assumed to be fixed. Whether or not the number of tasks can be influenced depends on the method. The number of tasks sets the possible degree of parallelism; a task is the smallest unit of work that can be executed by a node.

For the block approach, a fourth parameter is used. It can be chosen arbitrarily but needs to be a natural number.

h	blocking factor.

To compare the three distribution schemes, the following metrics are used.

Number of Tasks: The number of tasks is determined by the number of working sets that the dataset can be split into.

Communication Costs are based on the amount of data that need to be transmitted via the network. We assume that most of the input data can be read locally by the nodes and that the output is written locally, too. Hence, network costs are dominated by the costs to communicate intermediate data.

Replication Factor: The replication factor is the number of working sets that a certain element belongs to. Consequently, it tells how many copies of an element are made and distributed. The replication factor is used to calculate the size of intermediate data that is materialized in the system.

Working Set Size is the number of elements that a node processes per task. Because we want the working set to be kept in memory, its size may hit a limitation introduced by the amount of available main memory.

Evaluations per Task describes the number of function evaluations performed by a node in a single task. Ideally, this should be the total number of evaluations divided by the number of tasks. Otherwise, the different tasks are unequally expensive.

Table 1 summarizes all methods with respect to the introduced metrics and provides a first hint whether a characteristic is advantageous (checkmark), neutral (tilde), or disadvantageous (cross). The communication costs for the block approach, e.g., are $2vh$ because all v records are replicated h times and sent once for the computation and once for the optional aggregation (see Algorithms 1 and 2). All metrics for the design approach use $\sqrt{v}$ as an approximation for the number of elements per block. In a projective plane, i.e., when $v = \hat{q} = q^2 + q + 1$, each block contains $q + 1$ elements which, is approximately $\sqrt{v}$. In a design-like structure, i.e., when $v < \hat{q}$, the approximation also covers blocks that are slightly smaller because there are fewer than $\hat{q}$ elements available.

6. EVALUATION

Evaluating the number of tasks, i.e., the possible degree of parallelism, shows that the broadcast approach is most flexible with respect to this metric. The number of tasks can be any number, e.g., the number of nodes. The block approach allows to adjust the number of tasks although it is less flexible (p must be of the form $(h(h+1))/2$). The design approach does not allow to influence the number of tasks. However, because it is the same as the number of elements, no scalability issues occur. It can be assumed that there will always be many more tasks than nodes ($p \geq v > n$) so that no node should ever be idle.

Table 1 shows that the communication costs scale with p for the broadcast approach and with $\sqrt{p}$ for the other two approaches. Since $\sqrt{v} > n$ is likely, the communication costs for the design approach will usually be equal to the upper bound of $2vn$ (sending to all nodes).

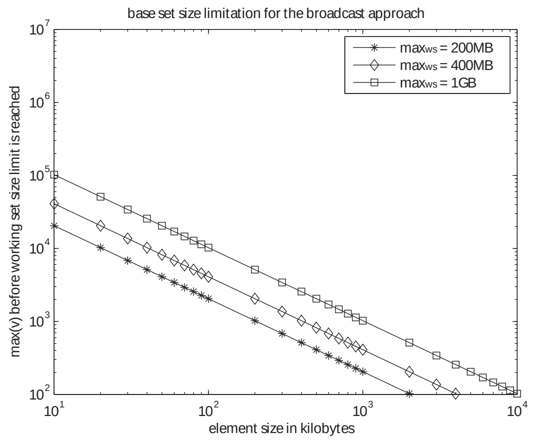
The row *Evaluations per Task* in Table 1 underlines that all approaches are well-balanced. The work is spread evenly among all nodes.

More crucial metrics (with respect to feasibility) are the replication factor and the working set size. This observation is based on two limits introduced by the execution environment: main memory and storage for intermediate results. First, the available main memory per node (denoted by max_{ws}) is limited; it can be as little as 200MB. An analysis of the available Hadoop infrastructures (Amazon AWS, Google/IBM academic cloud) showed that, although modern machines usually provide significantly more than 200MB of main memory, the amount available for each node is small. This is due to the facts that (a) multiple virtual machines share one physical machine and therefore its main memory and (b) each virtual machine is configured to host multiple nodes (instances of mappers and reducer) at the same time. The second limitation is posed by the available (and reasonable) storage for intermediate results (denoted by max_{is}). The storage limitation directly limits the replication factor.

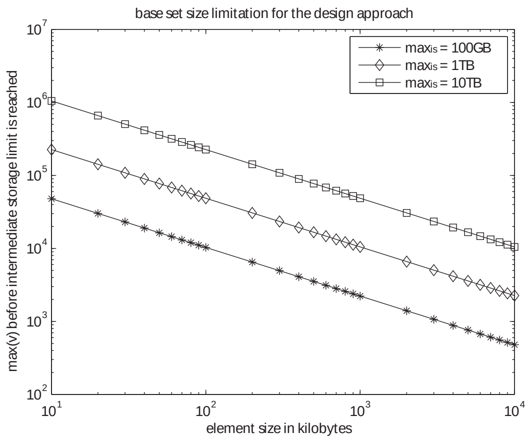
Final edited form was published in "HPDC '10: The 19th International Symposium on High Performance Distributed Computing. Chicago Illinois, 2010". S. 826-833. ISBN 978-1-60558-942-8
https://doi.org/10.1145/1851476.1851595

	Broadcast Approach	Block Approach	Design Approach
Number of Tasks (p)	✔ arbitrary	$\sim$ $\frac{h(h+1)}{2}$	✘ $q^2+q+1 \geq v$, q prime
Communication Costs	✘ $2vp$	✔ $2vh$	✘ $\approx 2v\sqrt{v}$ (max $2vn$)
Replication Factor	✔ p	✔ h	✘ $\approx \sqrt{v}$
Working Set Size	✘ v	✔ $2\left\lceil\frac{v}{h}\right\rceil$	✔ $\approx \sqrt{v}$
Evaluations per Task	✔ $\frac{v(v-1)}{2p}$	✔ $\left\lceil\frac{v}{h}\right\rceil^2$	✔ $\approx \frac{v-1}{2}$

Table 1: Comparison of distribution schemes



a



b

Figure 8: Limits on (a) working set size for broadcast approach and (b) intermediate storage for design approach

The different distribution schemes tend to violate different limitations. The broadcast approach, where the working set equals the whole dataset, tends to exceed max_{ws}. The design approach on the other hand, having small working sets, generates huge amounts of intermediate data caused by the replication factor being as high as $\sqrt{v}$. Figure 8(a) shows the maximum number of elements (v) — based on the size of each element in kilobytes — that can be processed before a working set hits max_{ws} (note the logarithmic axes in the charts). Similarly, Figure 8(b) shows the highest possible number of elements for the design approach before the intermediate storage requirements for materialized data exceed max_{is}.

The block approach is unique as it offers the blocking factor h to influence both working set size and replication factor. Depending on the requirements of a certain application and the limitations introduced by the environment, h can be chosen appropriately in a certain range. For both limitations we have:

$$\frac{2vs}{h} \leq max_{ws} \quad \text{and} \quad vsh \leq max_{is}.$$

It follows that h can only be chosen in a certain range:

$$\frac{2vs}{max_{ws}} \leq h \leq \frac{max_{is}}{vs}.$$

This implies a necessary condition on the size of the dataset (vs) for the existence of a valid h:

$$vs \leq \sqrt{\frac{max_{ws}max_{is}}{2}}.$$

Figure 9(a) shows the valid range for h for different values of max_{ws} and max_{is}. Rising lines are lower bounds for h imposed by max_{ws} and falling lines are upper bounds introduced by max_{is}. Consequently, there is no valid h for dataset sizes greater than the one where both limitations have an intersection. The valid ranges for h — assuming $max_{ws} = 200$MB and $max_{is} = 1$TB — are shaded in gray in the chart in Figure 9(a). Having, e.g., a dataset of size 4GB, it follows that h can be chosen arbitrarily between 39 and 263.

Figure 9(b) shows the comparison of all three approaches for fixed values of $max_{ws} = 200$MB and $max_{is} = 1$TB with respect to the maximum dataset cardinality for a given element size. The chart underlines that the broadcast approach is only reasonable for smaller datasets. It also shows that the design and block approach have a cross-over point and that for large elements ($>$ 1MB) the design approach allows a few more elements in the dataset than the block approach does.

We have implemented all three methods based on the Hadoop MR framework (version 0.20.1). Using these implementations, we conducted experiments on both, AWS EC2 and the Google/IBM aca-

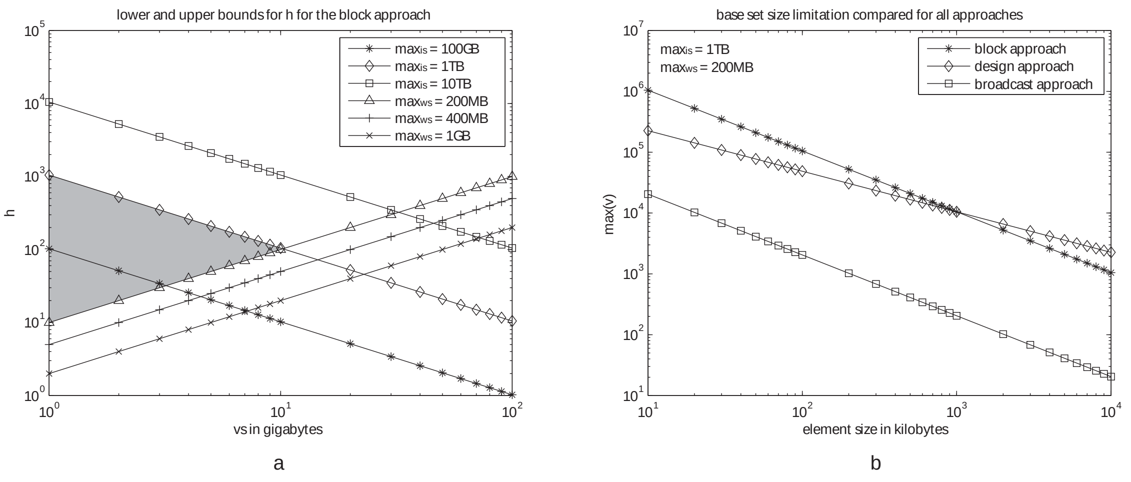


Figure 9: (a) Limits on h for block approach and (b) comparison of approaches

demic cloud. The results for replication factor and working set sizes showed to be close to our theoretic evaluations. However, we observed that the working set size limit was hit a little earlier than expected. This can be explained by the fact that, next to the elements themselves, other variables and data need to be kept in memory.

7. SUMMARY

In this paper, we showed that computing complex functions on pairs of elements is an important building block for a number of algorithms. We proposed a generic algorithm to perform pairwise element evaluation in parallel. Additionally, we showed that the MR programming model can be used to implement this algorithm.

We presented and compared three different distribution schemes that can be used to derive concrete algorithms. The evaluation of all methods with respect to limitations introduced by available systems showed that all approaches can be used to partition the Cartesian product of elements and therefore distribute the computational work to evaluate all pairs. However, all approaches hit limitations on the size of the dataset.

The limitations on the dataset size can be eased by introducing hierarchical solutions based on the existing distribution schemes. For the block approach, e.g., it is possible to build coarse-grained blocks and to process them sequentially. Each of these first level blocks is processed in parallel by building fine-grained second level blocks as shown before. Each block is aggregated before the next one is processed. This method eases both limits: the one on the working set size and the other one on the intermediate storage. For the design approach, it is similarly possible to process and aggregate subsets of all blocks sequentially, which reduces the requirements for intermediate storage. While the idea and implementation of these enhancements are clear, more thorough investigations of their characteristics are subject to future work.

Acknowledgments

This work was supported by IBM and Google via their academic cloud initiative.

The authors wish to thank Rainer Gemulla (IBM Research) and Dirk Habich (TU Dresden) for their creative and constructive input on this work.

8. REFERENCES

[1] http://aws.amazon.com/, 2009.
[2] http://hadoop.apache.org/, 2010.
[3] I. Anderson. *Combinatorial designs: Construction methods*. Ellis Horwood Chichester, West Sussex England, 1990.
[4] M. Bader and C. Mayer. Cache oblivious matrix operations using Peano curves. *LECTURE NOTES IN COMPUTER SCIENCE*, 4699:521, 2007.
[5] D. Chang, N. Jones, D. Li, M. Ouyang, and R. Ragade. Compute pairwise Euclidean distances of data points with GPUs. In *Proceedings of BIOCOMP'08*, pages 278–283, 2008.
[6] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *Proceedings of OSDI'04*, 2004.
[7] T. Elsayed, J. Lin, and D. Oard. Pairwise document similarity in large collections with MapReduce. In *Proceedings of ACL-08: HLT*, pages 265–268, 2008.
[8] M. Ester, H. Kriegel, J. Sander, and X. Xu. A density-based algorithm for discovering clusters in large spatial databases with noise. In *Proceedings of KDD-96*, pages 226–231, 1996.
[9] C. Gooi and J. Allan. Cross-document coreference on a large scale corpus. In *Proceedings of HLT/NAACL-04*, 2004.
[10] J. Lee, S. Kang, and H. Choi. A Fast Construction Algorithm for the Incidence Matrices of a Class of Symmetric Balanced Incomplete Block Designs. *LECTURE NOTES IN COMPUTER SCIENCE*, pages 11–19, 2004.
[11] P. Qiu, A. Gentles, and S. Plevritis. Fast calculation of pairwise mutual information for gene regulatory network reconstruction. *Computer Methods and Programs in Biomedicine*, 94(2):177–180, 2009.
[12] J. Shlens. A tutorial on principal component analysis. *Systems Neurobiology Laboratory, University of California at San Diego*, 2005.
[13] L. Smith. A tutorial on principal components analysis. *Cornell University, USA*, 51:52, 2002.