# Iterative Parallel Data Processing with Stratosphere: An Inside Look

Stephan Ewen* Sebastian Schelter* Kostas Tzoumas*
Daniel Warneke† Volker Markl*

*Technische Universität Berlin, Germany †International Computer Science Institute, Berkeley, USA
firstname.lastname@tu-berlin.de warneke@icsi.berkeley.edu

## ABSTRACT

Iterative algorithms occur in many domains of data analysis, such as machine learning or graph analysis. With increasing interest to run those algorithms on very large data sets, we see a need for new techniques to execute iterations in a massively parallel fashion. In prior work [3] we have shown how to extend and use a parallel data flow system to efficiently run iterative algorithms in a shared-nothing environment. Our approach supports the *incremental processing* nature of many of those algorithms.

In this demonstration proposal we illustrate the process of *implementing*, *compiling*, *optimizing*, and *executing* iterative algorithms on Stratosphere [7] using examples from graph analysis and machine learning. For the first step, we show the algorithm's code and a visualization of the produced data flow programs. The second step shows the optimizer's execution plan choices, while the last phase monitors the execution of the program, visualizing the state of the operators and additional metrics, such as per-iteration runtime and number of updates.

To show that the data flow abstraction supports easy creation of custom programming APIs, we also present programs written against a lightweight *Pregel API* [6] that is layered on top of our system with a small programming effort.

## Categories and Subject Descriptors

H.2.4 [**Database Management**]: Systems—*Parallel Databases*

## Keywords

iterative algorithms, graph processing, machine learning, query execution, parallel databases, query optimization

## 1. INTRODUCTION

The term *Big Data* has been coined for the current trend of ever-increasing volumes of collected data and the challenge to extract meaningful information from it. This trend has spawned many research projects, aiming at designing systems that support such complex big data analysis tasks. The grand challenge is to come up with a system that is flexible enough to support the heterogeneous structure of the data, expressive enough for the complex requirements of deep analysis algorithms, and efficient and scalable such that it can handle the large and growing volumes of data.

Over the past years, a variety of new data analytics systems have been proposed by industry and academia, trying to address these challenges. The landscape of these systems is quite diverse: It stretches from systems that offer fixed parameterizable pipelines, like MapReduce [2], to systems that start with a relational database core and extend it to support non-relational data and operations [4]. In-between these extremes are systems that allow for complex processing pipelines but have first-class support for non relational data types and flexible user-defined operations, like Stratosphere [1]. The last class of systems are domain specialized systems that support a subset of use-cases and are highly optimized for those [6].

Many of the tasks that are executed on such systems implement machine learning and graph analysis algorithms. Such algorithms play a key role in data mining and predictive analytics, which in turn are crucial in order to realize the promise of Big Data. It is only natural that the efficient execution of such algorithms on large data volumes has been a very hot topic of research recently. To run iterative algorithms over big data volumes, extensions to existing systems as well as specialized domain specific approaches have been proposed, the latter typically outperforming the first.

In prior work, we suggested and discussed methods to use a flexible data flow system to efficiently execute iterative algorithms [3]. Our approach yielded a system that runs general non-iterative analysis programs, and is at the same time very efficient in running iterative algorithms. The techniques used are centered around extensions to the data flow compiler to make it loop-aware, and the usage of a *workset iteration* abstraction that captures the semantics of *incremental iterations* to achieve similar performance with specialized systems while retaining the dataflow abstraction. Incremental iterations represent a large class of fast iterative algorithms that exploit sparse computational dependencies in the data to selectively recompute parts of the model in each step, rather then computing a completely new version.

In this demonstration, we illustrate the application of these techniques using the Stratosphere system [7]. With a set of example programs, we go through the different steps of creating such a program, compiling and optimizing it, and running it in the parallel runtime engine. For each step, we visualize several aspects, including the optimizer's plan choices and diverse characteristics of the execution.


*SIGMOD'13,* June 22–27, 2013, New York, New York, USA.

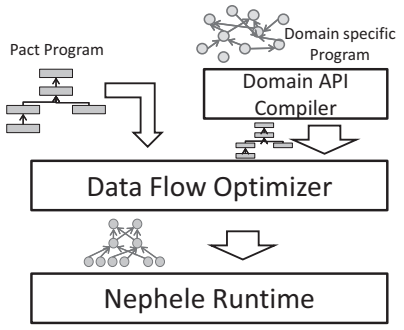


**Figure 1: The coarse system architecture**

In the remainder of the paper, we will first review the techniques used to run iterative algorithms in Stratosphere in Section 2. Afterwards, we describe the demonstration setup in Section 3 and conclude in Section 4.

# 2. STRATOSPHERE AND ITERATIVE ALGORITHMS

This section gives a brief background on Stratosphere and the techniques used to execute iterative algorithms. More details can be found in [3] and [1].

## 2.1 Stratosphere Architecture

The Stratosphere query processor is built from multiple layered components. Figure 1 shows the components relevant to this demo. The bottom of the stack is the parallel runtime engine *Nephele*. Nephele receives a parallel data flow description consisting of vertices and edges. Vertices contain sequential code that consumes and produces streams of records. Edges define how the data is streamed from one vertex to another. Processing is parallelized horizontally when multiple instances of the sequential code work on different partitions of the data stream (data parallelism), and vertically when the code of different operators executes in different vertices (pipeline parallelism).

Stratosphere offers multiple programming interfaces at different levels of abstraction. In this work, we build on top of the *PACT* Programming Model. PACT (= Parallelization Contract) programs are at a similar abstraction level as MapReduce: programs are written by implementing functions that the system evaluates on subsets (partitions) of the data. The functions are inside special second-order functions that define how these partitions are formed. In addition to *Map* and *Reduce*, PACT offers also binary second-order functions (*Match*, *CoGroup*, *Cross*), and allows free assembly of functions in directed acyclic graphs. PACT is hence a second-order functional data flow programming API with an abstraction somewhere between MapReduce (schema free, UDF centric) and relational algebra (complex expressions).

The transformation from PACT programs to the actual parallel data flow is done by an optimizer, similar as a relational algebra expression is transformed to a query execution plan. The optimizer picks the physical execution strategies for the individual functions with the objective of minimizing the overall execution cost. Some second-order functions can be realized with different physical strategies, and properties of the data can in several cases be reused across multiple functions. The latter is for example the case, when a partitioning is preserved by a function and meets the requirement of the successor function. The optimizer has a lightweight way of analyzing the user code to determine whether such properties are preserved [5].

## 2.2 Iterations

Iterations repeat a sequence of computations (the step function), over a data set called the partial- or intermediate solution, which is appropriately initialized. In parallel setups, multiple instances of the step function are evaluated in parallel on different partitions of the intermediate solution. In many settings, one evaluation of the step function on all parallel instances forms a *superstep*, which is also the granularity of synchronization [8].

In Stratosphere we distinguish between two different classes of iterative algorithms: *bulk iterations* and *incremental iterations*. Bulk iterations are the simple form of iterations: In each superstep, a bulk iteration evaluates the step function consuming the entire input (the result of the previous superstep, or the initial data set), and recomputes the next version of the intermediate solution.

In contrast, incremental iterations selectively modify elements of their intermediate solution and evolve the solution rather than fully recompute it. Where applicable, this leads to more efficient algorithms because not every element in the intermediate solution changes in each superstep. The *sparse computational dependencies* present in many problems and data sets allow a superstep to focus on the "hot" parts of the intermediate solution and leave the "cold" parts untouched. Frequently, the majority of the intermediate solution cools down comparatively fast and the later supersteps operate only on a small subset of the data. Note that the intermediate solution is implicitly forwarded to the next superstep, not requiring the algorithm to recreate it.

PACT programs implement iterative algorithms by defining a step function and embedding it into a special iteration operator. This operator invokes the step function repeatedly on the next version of the intermediate solution until a certain termination condition is reached. To support incremental iterations, PACT offers a *workset iteration* abstraction. The intermediate solution is here split into two data sets, the *solution set* and the *working set*. The solution set contains the current state of the intermediate solution while the working set contains the data that drives the computation, like candidates or auxiliary elements. A superstep consumes the entire working set and recomputes it but joins only with a relevant subset of the solution set and computes a delta of new or changed solution set elements. The delta is merged into the solution set at the end of the superstep. That way, large portions of the solution set (the "cold" part) may remain untouched in a superstep. The workset iteration terminates when a superstep computes an empty working set.

To illustrate this, consider the following example algorithm that computes the connected components of an undirected graph. The algorithm ultimately assigns component IDs ($cid$s) to the vertices, such that all vertices in the same connected component have the the same $cid$. Initially, all vertices get a $cid$ equal to their unique vertex ID ($vid$). In each step, a vertex propagates its current $cid$ as a candidate $cid$ to its neighbors. Each neighbor in turn accepts the candidate ID if it is lower then their own current component ID. The algorithm converges when no vertex takes a new $cid$ in a superstep. The incremental nature of the algorithm lies in the fact that a vertex only needs to propagate its current $cid$, if it received a new one in the previous superstep. If it did not receive a new $cid$, the vertex has no new information to contribute and is hence "cold" in the superstep.

Listing 1 shows the code of the algorithm as a workset iteration, written against the Scala version of Stratosphere's PACT API. The algorithm models the solution set as a set of pairs *(vid, cid)*, representing the assignment of component IDs to vertices. The vertex ID uniquely identifies an entry in the solution set. The workset con-

**Algorithm 1:** Connected Components as a PACT Program

```
1  val vertices = ... // vertices: (vid, vid)
2  val edges = ... // define edges: (source, target)
3  // define the function that is iteratively evaluated
4  def incr = (s:Stream[(Int, Int)], ws:Stream[(Int, Int)]) ⇒ {
5    // join the workset (changed vertices) with edges
6    val all = ws join edges on {_._1} isEqualTo {_._1}
7                using {(w, e) ⇒ e._2 → w._2}
8    // find the minimal candidate component id per vertex
9    val min = all reduceBy { _._1 }
10               using { cs ⇒ cs minBy { _._2 } }
11   // join candidates with solution set
12   val delta = min join s on {_._1} isEqualTo {_._1}
13               using {(n, s) ⇒ (n, s) match {
14                 case ((v, cNew), (_, cOld))
15                         if cNew < cOld ⇒ Some((v, cNew))
16                 case _                 ⇒ None
17               }
18             }
19   // return delta and new workset (identical here)
20   (delta, delta)
21 }
22 // evaluate the function iteratively and assign the result
23 output ← incr iterate (s0=vertices keyBy {_._1}, ws0=vertices)
```

tains all vertices that received a new $cid$ in the previous superstep, together with that new $cid$.

Lines 1 and 2 define data sources for the vertices and the edges, the latter assumed to be $(vid_1, vid_2)$ pairs. Line 4 starts the definition of the step function. The function takes the current solution- and workset as parameters ($s$ and $ws$ respectively). In lines 6-7, the function joins the workset (changed vertices) with the edges. The "join" keyword indicates that a function is to be executed through a "Match" second-order function[1]. The expression following "using" is the actual user function that is evaluated over the joined elements. The result of this operation is a set of $(vid, cid)$ tuples, where $vid$ is the target vertex from the edge and $cid$ is the component ID from the changed vertex. Lines 9-10 perform a $min$ aggregation per $vid$ to find the smallest candidate component ID for each vertex. Lines 12-18 create the delta set through a join between the minimal candidate $cid$s and the current solution set. The join checks whether the candidate ID is smaller than the current ID and returns an updated entry with the new $cid$ in that case. In the other cases, nothing is returned, marking the vertex as "cool" in the next superstep. Line 20 returns the delta set that is used to update the solution set and the new workset. Note that they are equal here, which is a special case in this algorithm; in the general case, the workset is different from the delta set. Line 23 defines the result or the iterative evaluation of the above defined function as the final result. Both the initial workset and the initial solution set are set to the vertex data set with $(vid, vid)$ tuples, indicating that all vertices are considered changed during the first superstep.

When compiling a program that includes an iteration, Stratosphere uses several optimizations to execute the iterations efficiently. The compiler performs an analysis of the program's data flow and identifies loop invariant parts. The result of these loop invariant parts are cached to prevent unnecessary re-computations. For workset iterations, the solution set is partitioned and stored in an index to facilitate fast access and updates.

[1]"match" itself is a reserved keyword in Scala.

For many domain specific problems, one can devise simpler APIs and programming paradigms. Those APIs are less general, but simplify the expression of certain algorithms. A popular example is the *vertex-centric computing* model that is implemented for example in Google's Pregel system [6].

Through its flexibility, the PACT programming model forms a good basis for implementing domain specific languages or APIs on top of it. Such higher level APIs are typically implemented in a very lightweight manner by defining a fixed data flow of second-order functions. The domain specific API creates or parameterizes the actual UDFs that are applied in the PACT program. By mapping the domain specific programs to a PACT data flow, the higher level APIs automatically benefit from the optimizer and the scalable and robust parallel runtime, which simplifies their implementation significantly.

## 3. DEMONSTRATION SCENARIO

In this demonstration, we visualize the phases a iterative analysis programs using Stratosphere [7]: their *specification*, *compilation and optimization*, and their massively parallel *execution*. The example programs we use for the demonstration solve well-known graph analysis and machine learning problems, such as computing PageRank, finding shortest paths, finding densely connected subcomponents, optimizing with gradient descent, and matrix factorization. The demo attendant may select from these algorithms and select among several parameters, like input data set, or error thresholds. The Stratosphere system runs on a cluster at TU Berlin, and the visualization is forwarded via network. We provide test data sets on which the example programs run quickly to allow interactive demonstrations.

The demo programs are **specified** using the PACT API. They contain the definitions of the user-defined functions, as well as their composition into a data flow, similar to the example in Algorithm 1. The example algorithms cover bulk iterations and incremental iterations, and for some algorithms we provide both versions. The data flow of an algorithm is diplayed by the demonstration client when selecting an example algorithm to run.

To demonstate the ability to create domain specific APIs on top of the PACT data flow abstraction, we provide an implementation of a *vertex-centric computing* API that is similar to Pregel [6]. Where applicable, we provide an additional version of the example algorithms in that API. Because internally, the programs are mapped to PACT data flows, the remaining demo steps are the same as for the other programs.

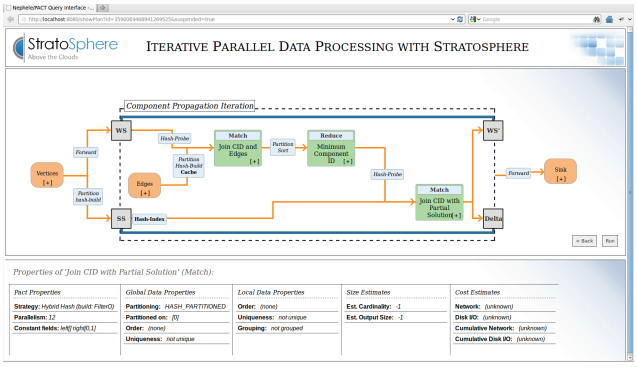


**Figure 2: Screenshot of the optimizer-created execution plan.**

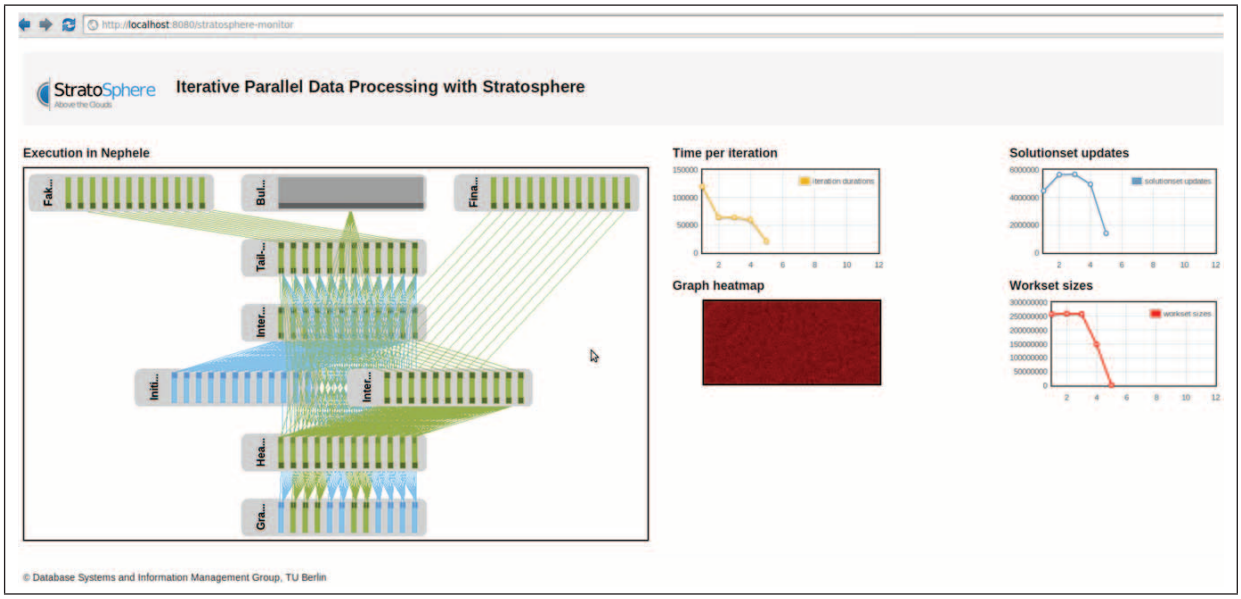


**Figure 3: The execution monitor**

Once a program is submitted to the system, it is **optimized** and transformed to a parallel data flow graph. As described in Section 2.1, the optimizer picks and places physical operators and creates the execution plan. Figure 2 shows a screenshot of the demo client displaying a plan. The plan visualization draws the data flow and labels vertices and edges with the optimizer's choice of execution strategies. Each vertex corresponds to a PACT, containing the user-defined function and a strategy to apply it to the data, according to the second-order function. Edges represent the result of one PACT being forwarded to the next. They are labeled with the data shipping strategy they execute (e. g. in-memory pipe or network hash partitioning) and local operations they apply (such as sorting). In addition, the plan shows which data structure is used to store and update the solution set. A click on a vertex shows inferred properties of the data, such as partitioning and order properties, as well as basic cost estimates for disk and network.

Once the plan is examined, it can be forwarded to the runtime for execution.

We visualize the program's **parallel execution** using an extended version of Stratosphere's runtime monitor, as shown in the screenshot in Figure 3. The left hand side shows the parallel data flow with its vertices and edges. Vertices inside a group execute the same operations on different partitions of the data. The color of a vertex indicates its status: executing, end-of-superstep, or finished.

The right hand side of the execution monitor demonstrates several graphs that show multiple performance metrics, like the timer per superstep. In the case of bulk iterations, additional graphs may display statistics about the job, such as the total error difference in each superstep. For incremental iterations, the graphs show the number of elements in the work set and the number of changes in the solution set as an indicator of the work done per superstep. When applicable, a heatmap is shown, visualizing which parts of solution set are currently active.

# 4. CONCLUSION

We demonstrate the techniques used in Stratosphere to efficiently execute iterative algorithms on large data sets. Stratosphere is a very flexible data flow system with dedicated support for iterative algorithms. A distinguishing aspect is its abstraction for workset iterations, which allow it to efficiently run algorithms that only incrementally evolve the computed solution.

With a set of example programs, we illustrate how these programs are written, how the system optimizes them, and how they are finally executed in the distributed parallel runtime.

## Acknowledgments

This research is funded by the German Research Foundation under grant "FOR 1036: Stratosphere - Information Management on the Cloud" and the European Union (EU) grant no. 257859 (project "ROBUST"). D. Warneke is supported through a scholarship of the German Academic Exchange Service (DAAD).

# 5. REFERENCES

[1] D. Battré, S. Ewen, F. Hueske, O. Kao, V. Markl, and D. Warneke. Nephele/PACTs: A Programming Model and Execution Framework for Web-Scale Analytical Processing. In *Symposium on Cloud Computing*, 2010.

[2] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *OSDI*, pages 137–150, 2004.

[3] S. Ewen, K. Tzoumas, M. Kaufmann, and V. Markl. Spinning fast iterative data flows. *PVLDB*, 5(11):1268–1279, 2012.

[4] E. Friedman, P. Pawlowski, and J. Cieslewicz. SQL/MapReduce: a practical approach to self-describing, polymorphic, and parallelizable user-defined functions. *Proc. VLDB Endow.*, 2(2):1402–1413, 2009.

[5] F. Hueske, M. Peters, M. Sax, A. Rheinländer, R. Bergmann, A. Krettek, and K. Tzoumas. Opening the black boxes in data flow optimization. *PVLDB*, 5(11):1256–1267, 2012.

[6] G. Malewicz, M. H. Austern, A. J. C. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: a system for large-scale graph processing. In *SIGMOD Conference*, 2010.

[7] Stratosphere Project. `http://stratosphere.eu/`, 2013.

[8] L. G. Valiant. A bridging model for parallel computation. *Commun. ACM*, 33(8):103–111, 1990.