Efficient Iterative Processing in the SciDB Parallel Array Engine

Emad Soroush[1], Magdalena Balazinska[1], Simon Krughoff[2], and Andrew Connolly[2]
[1]Dept. of Computer Science & Engineering [2] Astronomy Department
University of Washington, Seattle, USA
{soroush,magda}@cs.washington.edu
{krughoff, ajc}@astro.washington.edu

Abstract

Many scientific data-intensive applications perform iterative computations on array data. There exist multiple engines specialized for array processing. These engines efficiently support various types of operations, but none includes native support for iterative processing. In this paper, we develop a model for iterative array computations and a series of optimizations. We evaluate the benefits of an optimized, native support for iterative array processing on the SciDB engine and real workloads from the astronomy domain.

1. INTRODUCTION

Scientific data often takes the form of multidimensional arrays (*e.g.*, 2D images or 3D environment simulations) and many data management systems are being built to support the array model natively [5, 13, 21]. Additionally, to handle today's large-scale datasets, several engines, including SciDB [13], provide support for processing arrays in parallel in a shared-nothing cluster. Several benchmark studies have shown that these specialized array engines outperform both relational engines and MapReduce-type systems on a variety of array workloads [2, 18].

Many data analysis tasks today require iterative processing [6] and most modern big data management and analytics systems (*e.g.*, [7, 19]) support iterative processing as a first-class citizen including a variety of optimizations for these types of computations [1, 7, 10].

The need for efficient iterative computation extends to analysis executed on multi-dimensional scientific arrays. We describe two applications in Section 2. While it is possible to implement iterative array computations by repeatedly invoking array queries from a script, this approach is highly inefficient (as we show in Figure 4). Instead, a large-scale array management systems such as SciDB should support iterative computations as first-class citizens in the same way other modern data management systems do for relational or graph data.

Contributions: In this paper, we introduce a new model for expressing iterative queries over arrays. We develop a middleware system called ArrayLoop that we implement on top of SciDB to translate queries expressed in this model into queries that can be executed in SciDB. Importantly, ArrayLoop includes three optimizations that trigger rewrites to the iterative queries and ensure their efficient evaluation. The first optimization also includes extensions to the SciDB storage manager. More specifically, the contribution of this paper are as follows:

(1) New model for iterative array processing (Sections 3 and 4): Iterating over arrays is different from iterating over relations. In the case of arrays, the iteration starts with an array and updates the cell values of that array. It does not generate new tuples as in a relational query. Additionally, these update operations typically operate on neighborhoods of cells. These two properties are the foundation of our new model for iterative array processing. Our model enables the declarative specification of iterative array computations, their automated optimization, and their efficient execution.

(2)Incremental iterative processing (Section 5): In many iterative applications, the result of the computation changes only partly from one iteration to the next. As such, implementations that recompute the entire result every time are known to be inefficient. The optimization, called *incremental iterative processing* [6], involves processing only the part of the data that changes across iterations. While the idea of incremental iterations has previously been developed for relational systems, its implementation in an array engine is very different: For an array engine, the optimization can be pushed *all the way to the storage manager* with significant performance benefits. We develop and evaluate such storage-manager-based approach to incremental array processing. Our iterative array model enables the automatic generation of incremental computations from the user's declarative specification of the overall iterative query.

(3) Overlap iterative processing (Section 6): In iterative array applications, including, for example, cluster finding and source detection, operations in the body of the loop update the value of the array cells by using the values of other neighboring array cells. These neighborhoods are often bounded in size. These applications can effectively be processed in parallel if the system partitions an array but also replicates a small amount of overlap cells. In the case of iterative processing, the key challenge lies in keeping these overlap cells up to date. This optimization is specific to queries over arrays and does not apply to relational engines. Our key contribution here lies in new mechanisms for man-

SSDBM '15, June 29 - July 01, 2015, La Jolla, CA, USA

 ISBN 978-1-4503-3709-0/15/06...$15.00
DOI: http://dx.doi.org/10.1145/2791347.2791362

aging the efficient reshuffling of the overlap data across iterations.

(4)Multi-resolution iterative processing (Section 7): Finally, in many applications, the raw data lives in a continuous space (3D universe, 2D ocean, N-D space of continuous variables) and arrays capture discretized approximations of the real data. Different data resolutions are thus possible and scientifically meaningful to analyze. In many applications, it is often efficient to first process the low-resolution versions the data and use the result to speed-up the processing of finer-resolution versions. Our final optimization leverages this approach to speed up iterative computations.

(5) Implementation and evaluation We implement the iterative model and all three optimizations as extensions to the open-source SciDB engine and we demonstrate their effectiveness on experiments with 1 TB of publically-available synthetic LSST images [12]. Experiments show that *Incremental iterative processing* can boost performance by a factor of 4-6X compared to a non-incremental iterative computation. *Iterative overlap processing* together with *mini-iteration processing* can improve performance by 31% compare to SciDB's current implementation of overlap processing. Finally, the *multi-resolution optimization* can cut runtimes in half if an application can leverage this technique. Interestingly, these three optimizations are complementary and their benefits can be compounded.

2. MOTIVATING APPLICATIONS

We start by presenting two array-oriented, iterative applications. We use these applications as examples throughout the paper and also in the evaluation.

Example 2.1. Sigma-clipping and co-addition in LSST images (SigmaClip): When analyzing telescope images, such as those from the LSST survey [8], some sources (a "source" can be a galaxy, a star, etc.) are too faint to be detected in one image but can be detected by stacking multiple images from the same location on the sky. Before the co-addition is applied, astronomers often run a "sigma-clipping" noise-reduction algorithm. Sigma-clipping consists in grouping all pixels by their (x,y) coordinates. For each location, the algorithm computes the mean and standard deviation of the flux. It then sets to null all cell values that lie k standard deviations away from the mean. The algorithm iterates by re-computing the mean and standard deviation. The cleaning process terminates once no new cell values are filtered out. Throughout the paper, we refer to this application as **SigmaClip**. □

Example 2.2. Iterative source detection algorithm (SourceDetect): Once telescope images have been cleaned and co-added, the next step is to extract the actual sources from the images. A simple source detection algorithm is to initialize each non-empty cell with a unique label and consider it as a different object. At each iteration, each cell resets its label to the minimum label value across its neighbors. Two cells are neighbors if they are adjacent. This procedure continues until the algorithm converges. We refer to this application as **SourceDetect**. □

3. ITERATIVE ARRAY MODEL

In SciDB, users operate on arrays by issuing declarative queries using either the Array Query Language (AQL) or the

Algorithm 1 `SigmaClip` application

```
1. function sigma-clipping(A,k)                          ▷ Naïve sigma-clip
2.   Input: Iterative Array A <float d>[x,y,t]
3.   Input: k a constant parameter.
4.   while (some pixels A[x, y, t] are filtered) do
5.     T[x, y] = select avg(d) as μ, stdv(d) as σ from A group by x, y
6.     S[x, y, t] = select * from T join A on T.x = A.x and T.y=A.y
7.     A[x, y, t] = select d from S where μ − k × σ ≤ d ≤ μ + k × σ
8.   end while
9. end function
```

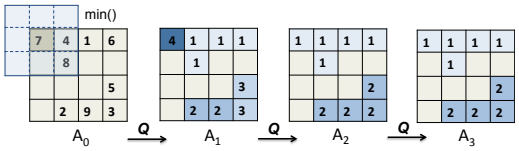


Figure 1: Iterative array A and its state at each iteration for the `SourceDetect` application. $\{Q^{f^\pi,\delta^\pi}_{cells(A)} : \forall c_{i,j} \in cells(A)\ i \in I_1\ \&\ j \in I_2\ \}$ **where** $I_1 = I_2 = \{1,2,3,4\}$ **are the sets of dimension values,** f^π **applies** $min()$ **aggregate on each group of cells,** δ^π **simply stores the aggregated value in each cell** $c_{i,j}$**, and** $\pi : (x,y) \rightarrow [x \pm 1][y \pm 1]$**. At each iteration, a sliding window scans through all the cells.**

Array Functional Language (AFL). The `select` statements in Algorithm 1, which corresponds to the `SigmaClip` application, are examples AQL queries. AQL and AFL queries are translated into query plans in the form of trees of array operators. Each operator O takes one or more arrays as input and outputs an array: $O : A \rightarrow A$ or $O : A \times A \rightarrow A$.

To enable the automatic optimization of iterative computations, such as the one in Algorithm 1, we extend the basic array model with constructs that capture in greater detail how iterative applications process arrays.

In an iterative computation, the goal is to start with an initial array A and transform it through a series of operations in an iterative fashion until a termination condition is satisfied. The iterative computation on A typically involves other arrays, including arrays that capture various intermediate results (*e.g.*, arrays containing the average and standard deviation for each (x,y) location in the `SigmaClip` application) and arrays with constant values (*e.g.*, a connectivity matrix in a graph application). We refer to the array being modified during the iteration as the *iterative array*. Figure 1 shows a (4×4) iterative array that represents a tiny telescope image in the `SourceDetect` application. In the initial state, A_0, each pixel with a flux value above a threshold is assigned a unique value. As the iterative computation progresses, adjacent pixels are re-labeled as they are found to belong to the same source. In the final state A_3, each set of pixels with the same label corresponds to one detected source.

Iterative applications typically define a termination condition that examines the cell-values of the iterative array: In Figure 1, the termination condition T is the count of differences between A_i and A_{i+1}. Our ArrayLoop system represents T as an AQL function.

An iterative array computation takes an iterative array, A, and applies to it a computation Q until convergence, where Q is a sequence of valid AQL or AFL queries. At each step, Q can either update the entire array or only some subset of the array. We capture the distinction with the notion of

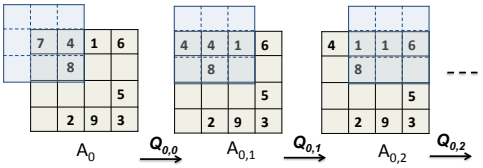


Figure 2: Iterative array A and its state after three minor steps, each of the form: $Q_{i,j} = Q_{c_{i,j}}^{f^\pi,\delta^\pi}$ where $c_{i,j}$ is the cell at $A[i][j]$, f^π applies $min()$ aggregate, δ simply stores the aggregate result as the new value in cell $c_{i,j}$, and $\pi : (x, y) \rightarrow [x \pm 1][y \pm 1]$

major and *minor* iteration steps. A state transition, $A_i \xrightarrow{Q} A_{i+1}$, is a *major step* if the function Q operates on all the cells in A at the same time. Otherwise it is a *minor step*. We are interested in modeling computations where each major step can be decomposed into a set of minor steps that can be evaluated in parallel. Figure 1 shows an iterative array computation with only major steps involved, while Figure 2 presents the same application but executed with minor steps.

We further observe from the example applications in Section 2 that the functions Q often follow a similar pattern. First, the value of each cell in iterative array A_{i+1} that is updated by Q only depends on values in *nearby cells* in array A_i. We capture this spatial constraint with a function π that specifies the mapping from output cells back onto input cells:

Definition 3.1. π is an *assignment* function defined as $\pi : cells(A) \rightarrow \mathcal{P}(cells(A))$, where $cells(A)$ is the set of all the cells in array A and $\mathcal{P}()$ is the powerset function.

Figure 3 illustrates two examples of assignment functions. Our ArrayLoop system supports two types of assignment functions: *windowed* functions such as those illustrated in Figure 3 and *attribute* assignment function. The latter occur in applications such as K-means clustering where all the cells with the same label are grouped together.

Definition 3.2. f^π is an aggregate function defined as $f^\pi : cells(A) \rightarrow \tau$. f^π groups the cells of the array A according to assignment function π, with one group of cells per cell in the array A. It then computes the aggregate functions separately for each group. The aggregate result is stored in tuple τ.

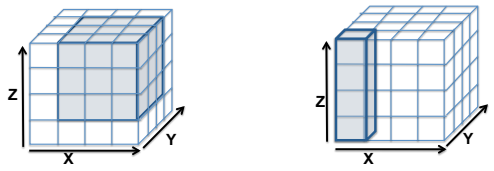
Finally, Q updates the output array with the computed aggregate values:

Definition 3.3. $\delta^\pi : (cells(A), f^\pi) \rightarrow cells(A)$ is a *cell-update* function. It updates each cell of the array A with the corresponding tuple τ computed by f^π and the current value of the cell itself.

These three pieces together define the iterative array computation $Q_C^{f^\pi,\delta^\pi}$ as follows:

Definition 3.4. *An iterative array computation* $Q_C^{f^\pi,\delta^\pi}$ on the subset of cells C where $C \in \mathcal{P}(cells(A))$ generates subset of cells $C' \in \mathcal{P}(cells(A))$ such that $\forall c \in C$ and $c' \in C'$ $c' = \delta^\pi(c, f^\pi(c))$ where c and c' are two corresponding cells in those subsets.

In the example from Figures 1 and 2, which illustrate the `SourceDetect` application, the goal is to detect all the clusters in the array A, where each cell $p_1 = (x_1, y_1)$ in a cluster



(a) `SourceDetect` (b) `SigmaClip`

Figure 3: Two examples of window assignment functions: (a) $\pi_1 : (x, y, z) \rightarrow [x \pm 1][y \pm 1][z \pm 1]$, the associated window is highlighted for the cell at $(2, 1, 2)$. (b) $\pi_2 : (x, y, z) \rightarrow [x][y]$, the associated window is highlighted for all the cells at (x, y, z) with $z = 0$.

has at least one neighbor $p_2 = (x_2, y_2)$ in the same cluster such that $|x_1 - x_2| \leq 1$ and $|y_1 - y_2| \leq 1$, if it is not a single-cell cluster. In this application, π is the 3X3 window around a cell. We slide the window over the array cells in major order. At each *minor* step, at each cell $c_{i,j}$ at the center of the window, we apply an iterative array computation $Q_{i,j} = Q_{c_{i,j}}^{f^\pi,\delta^\pi}$ where f^π applies a $min()$ aggregate over the 3x3 window, π, and δ^π is a cell-update function that simply stores the result of the $min()$ aggregate into cell $c_{i,j}$. Figure 2 illustrates three steps of this computation. Notice that the output of the iterative array computation $Q_{0,0}$ becomes the input for $Q_{0,1}$ and so on. Another strategy is to have many windows grouped and applied together. In other words, instead of applying the iterative array computation per cell, we apply $Q_C^{f^\pi,\delta^\pi}$ on a group of cells $C \in \mathcal{P}(cells(A))$ in one *major* step. Figure 1 shows the iterative array computation for the latter strategy. The former strategy has less expensive steps than the latter strategy, but it requires more steps to converge.

In our model, we encapsulate all the elements of the model in a *FixPoint* operator:

$$FixPoint(A, \pi, f, \delta, T, \epsilon) \quad (1)$$

With our model, the user specifies the logic of the iterative algorithm without worrying about the way it is going to be executed. Our model can be implemented and executed on top of various array execution engines. In the rest of the paper, we describe how the queries specified in our model are rewritten and efficiently run in the SciDB array engine. The execution strategy in SciDB uses only major steps. Mini-step iterations, i.e. asynchronous execution, is left for future work.

4. ITERATIVE ARRAY PROCESSING

We extend SciDB-Py [15] with a python $FixPoint()$ operator following the model from Section 3. We also develop an optimizer module that we name ArrayLoop. The user encapsulates its iterative algorithm in the $FixPoint()$ operator. The ArrayLoop optimizer sits on top of SciDB. ArrayLoop rewrites a $FixPoint(A, \pi, f, \delta, T, \epsilon)$ operator into the AQL queries in Listing 1 that it wraps with an internal while loop.

`is_window` helper function in Listing 1 clarifies whether the window assignment function translates to a window aggregate or a group-by aggregate. ArrayLoop translates a window assignment function to a group-by aggregate if mapping is from a set of input dimensions to one of its subsets. If mapping is from a set of dimensions to the same set of

Listing 1 Pseudocode for rewriting FixPoint operator

```
Input:
  FixPoint(A,pi,f,delta,T,epsilon)
Output:
  While (T(A,A_prev)  < epsilon)
   // Termination function T is also AQL function.
   // Compute the new aggregates from the current iterative array.
   If (is_window(pi))
      G = SELECT f FROM A WINDOW PARTITIONED BY pi
   else
      G = SELECT f FROM A GROUP BY pi
   // Combine the new aggregate with the old value of the cell.
   S = SELECT * FROM G JOIN A ON <matching dimensions>
   A_new = SELECT delta(S) FROM S
   A_prev = A
   A = A_new
```

Algorithm 2 ArrayLoop version of the `SigmaClip` application followed by image co-addition

1. **function ArrayLoop-sigma-clipping**(A,k) ▷ SigmaClip algorithm with FixPoint operator provided by the user.
2. Input: Iterative Array A <float d>$[x,y,t]$,
3. Input: k a constant parameter.
4. $\pi : [x][y][z] \rightarrow [x][y]$.
5. δ : "$A.d \geq \mu - k \times \sigma$ and $A.d \leq \mu + k \times \sigma ? A : null$"
6. $f : \{avg()\ as\ \mu, stdv()\ as\ \sigma\}$
7. $FixPoint(A, \pi, f, \delta, count(), 0)$
8. **end function**

9. **function ArrayLoop-incr-sigma-clipping**(A,k) ▷ ArrayLoop incremental rewriting of the SigmaClip.
10. Input: Iterative Array A <float d>$[x,y,t]$,
11. Input: k: a constant parameter.
12. Local: Iterative Array C <int c,float s,float s^2>$[x,y]$,
13. Local: Array S <float σ,float μ>$[x,y]$,
14. $\Delta A^- \leftarrow A$
15. **while** (ΔA^- is not empty) **do**
16. $T[x, y] \leftarrow$ select count(d) as c, sum(d) as s, sum(d^2) as s^2 from ΔA^- group by x, y
17. **if** (first iteration) **then**
18. $C \leftarrow T[x, y]$
19. **else**
20. merge(C,T,$C.c - T.c$)
21. merge(C,T,$C.s - T.s$)
22. merge(C,T,$C.s^2 - T.s^2$)
23. **end if**
24. $S[x, y] \leftarrow$ select $\frac{T.s}{T.c}$ AS μ, $\sqrt[2]{\frac{T.s^2}{T.c} - (\frac{T.s}{T.c})^2}$ AS σ FROM $\Delta^+ C$
25. merge($A, S, S.\mu - k \times S.\sigma \leq A.d \leq S.\mu + k \times S.\sigma ? A : null$)
26. **end while**
27. **end function**
    co-addition phase:
28. $R[x, y] \leftarrow$ select sum($A.d$) as $coadd$ from A group by x, y

dimensions with additional offsets per dimension, then ArrayLoop translates it to window-aggregate.

In addition, ArrayLoop also implements a set of query rewriting tasks in order to leverage a series of optimizations that we develop: *incremental iterative processing*, *overlap iterative processing*, and *multi-resolution iterative processing*.

ArrayLoop acts as a pre-processing module before executing the iterative query in SciDB. Currently the majority of the ArrayLoop implementation is outside the core SciDB engine. ArrayLoop relies on SciDB for features such as distributed query processing, fault-tolerance, data distribution, and load balancing. In the following sections, we describe each of the three optimizations in more detail.

5. INCREMENTAL ITERATIONS

Incremental iterative processing [6] is a well-known optimization, *e.g.* in semi-naive datalog evaluation, and has been shown to significantly improve performance in relational and graph systems. ArrayLoop leverages the iterative computation model from Section 3 to automatically apply this optimization when the semantics of the applications permit it. The `SigmaClip` application described in Section 2.1 (and shown in Algorithm 1) is an example application that can benefit from incremental iterative processing.

In ArrayLoop, we show how the incremental processing optimization can be applied to arrays. As shown in Algorithm 2, the user provides a `FixPoint` operator in `ArrayLoop-sigma-clipping` function. ArrayLoop *automatically expands and rewrites the operation into an incremental implementation* as shown in the `ArrayLoop-incr-sigma-clipping` function.

Given the `FixPoint` operator, ArrayLoop performs two tasks: (1) it automatically rewrites aggregate functions, if possible, into incremental ones and (2) it efficiently computes the last state of the iterative array using the updated cells at each iteration. The automatic rewrite is enabled by the precise model for iterative computations in the form of the three functions π, f, and δ. Given this precise specification of the loop body, ArrayLoop rewrites the computation using a set of rules that specify how to replace aggregates with their incremental counter-parts when possible. To efficiently compute incremental state updates, we introduce a special *merge* operator. We omit the description of the operator and the optimization that pushes incremental computation into the storage manager due to space constraints but refer the reader to the full version of this paper for details [17].

In our example, the rewrite proceeds as follows. Increments between iterations translate into updates to array cells and can thus be captured with two auxiliary arrays: a positive delta array, ΔA+, which records the new values of updated cells and a negative delta array, ΔA-, which keeps track of the old values of updated cells. If the aggregate function f is incremental, ArrayLoop replaces the initial aggregation with one over one or both of these delta arrays. For example, for `ArrayLoop-incr-sigma-clipping`, only negative delta arrays are generated at each iteration(there is no ΔA^+). So the rewrite produces a group-by aggregate only on ΔA^- (line 16). Next, ArrayLoop merges the partial aggregate values with the aggregate results from the previous iteration (lines 20 through 22). The aggregate rewrite rules define that merge logic for all the aggregate functions. In this example, ArrayLoop will generate one merge statement per aggregate function computed earlier. Finally, on Line 24, ArrayLoop does the final computation to generate the final aggregate values for this iteration. Note that this final phase in the aggregate computation is always done on positive delta arrays (ΔC^+), which generates the same result as computing on negative delta array ΔC^- followed by a subtract merge plus computing on positive delta array ΔC^+ followed by an addition merge. Line 25 leverages the δ function to generate the ΔA^- of the next iteration.

6. ITERATIVE OVERLAP PROCESSING

To process a query over a large-scale array in parallel, SciDB (and other engines) break arrays into sub-arrays called chunks, distribute chunks to different compute nodes (each node receives multiple chunks), and process chunks in parallel at these nodes. Frequently, the value of each output array cell is based on a neighborhood of input array cells. To efficiently process such operations, some have suggested to extract, for each array chunk, an overlap area ϵ from neighboring chunks, store the overlap together with the original chunk [13], and provide both the core data and overlap data to the operator during processing [4]. This technique is called *overlap processing*.

6.1 Efficient Overlap Processing

The `SourceDetect` application described in Section 2.2 is an example application that can benefit form overlap processing. To efficiently update overlap array cells during the iterative computation, our approach is to leverage SciDB's bulk data-shuffling operators as follows: SciDB's operator framework implements a `bool requiresRepart()` function that helps the optimizer to decide whether the input array requires repartitioning before the operator actually executes. We extend the SciDB operator interface such that ArrayLoop can dynamically set the returned value of the operator's `requiresRepart()` function. To update overlap data, ArrayLoop sets the `requiresRepart()` return value to true. ArrayLoop has the flexiblity to set the value to true either at each iteration or every few iterations as we discuss further below. In case an operator in SciDB is guided by ArrayLoop to request repartitioning, the SciDB optimizer injects the `Scatter/Gather` [14] operators to shuffle the data in the input iterative array before the operator executes.

6.2 Mini-Iteration Processing

We observe that a large subset of iterative applications have the property that overlap cells can be updated only every few iterations. These are applications, for example, that try to find structures in the array data, *e.g.* `SourceDetect` application. These applications can find structures locally and eventually need to exchange information to stitch these local structures into larger ones. For those applications, ArrayLoop can add the following additional optimization: ArrayLoop runs the algorithm for multiple iterations without updating the replicas of overlap cells. The application iterates over chunks locally and independently of other chunks. Every few iterations, ArrayLoop triggers the update of overlap cells, and continues with another set of local iterations. The key idea behind this approach is to avoid data movement across array chunks unless a large enough amount of change justifies the cost. We call each series of local iterations without overlap cell synchronization *mini iterations*.

7. MULTI-RESOLUTION OPTIMIZATION

As discussed earlier, many algorithms search for structure in array data. One example is the extraction of celestial objects from telescope images, snow cover regions from satellite images, or clusters from an N-D dataset. In these algorithms, it is often efficient to first identify the outlines of the structures on a low-resolution array, and then refine the details on high-resolution arrays. We call this array-specific optimization *multi-resolution* optimization.

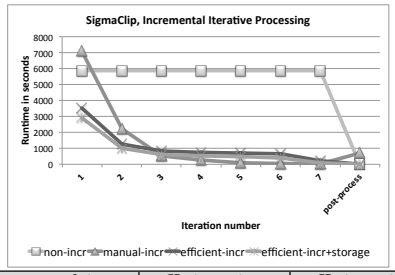
To initiate the *multi-resolution* optimization, ArrayLoop initially generates a series of versions, $A^i, A^{i+1}, \ldots, A^j$, of the original iterative array A. Each version has a different resolution. A^i is the original array. It has the highest resolution. A^j is the lowest-resolution array. The coarser-grained, pixelated versions are generated by applying a sequence of `grid` followed by `filter` operations represented together as $grid_p()$, where p is the predicate of the `filter` operator. The size and the aggregate function in the `grid` operator are application-specific and are specified by the user. The `SourceDetect` application has a grid-size of (2×2) and an aggregate function `count` with a filter predicate that only passes grid blocks without empty cells (in this scenario all the grid blocks with count=4). This ensures that cells that are identified to be in the same cluster in a coarsened version of the array, remain together in finer grained versions of the array as well. In other words, the output of the iterative algorithm on the pixelated version array A^j should be a *valid* intermediate step for A^{j-1}. ArrayLoop runs the iterative function Q on the sequence of pixelated arrays in order of increasing resolution. The output of the iterative algorithm after convergence at pixelated version A^i is transformed into a finer-resolution version using an `xgrid` operator (inverse of a grid operator). It is then merged with A^{i-1}, the next immediate finer-grained version of the iterative array. We represent both operations as $xgrid_m()$. By carefully merging the approximate results with the input array at the next finer-grained level, ArrayLoop skips a significant amount of computation.

8. EVALUATION

In this section, we demonstrate the effectiveness of ArrayLoop's native iterative processing capabilities including the three optimizations on experiments with 1TB of `LSST` images [12]. The images take the form of one large 3D array (2D images accumulated over time) with almost 44 billion non-empty cells. We run the `SigmaClip` application on the large 3D array and `SourceDetect` on the co-added 2D version of the whole dataset. The experiments are executed on a 20-machine cluster. (Intel(R) Xeon(R) CPU E5-2430L @ 2.00GHz) with 64GB of memory and Ubuntu 13.04 as the operating system.

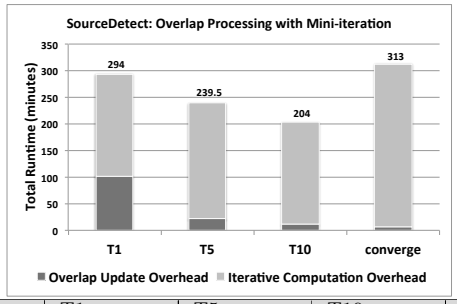
Incremental Iterative Processing: We first demonstrate the effectiveness of our approach to bringing incremental processing to the iterative array model in the context of the `SigmaClip` application. Figure 4 shows the total runtime of the algorithm with different execution strategies. As shown, the `non-incremental` "sigma-clipping" algorithm performs almost four times worse than any other approach. The `manual-incr` approach is a manually-written incremental version of the "sigma-clipping" algorithm. This approach keeps track of all the points that are still candidates to be removed at the next iteration and discards the rest. `efficient-incr` and `efficient-incr+storage` are the two strategies used by ArrayLoop (`ArrayLoop-incr-sigma-clipping` function from Section 5). `efficient-incr` represents ArrayLoop's query rewrite for incremental state management that also leverages our `merge` operator. `efficient-incr+storage` further includes the storage manager extensions. Figure 4 shows the total runtime in each case. ArrayLoop's efficient versions of the algorithm are competitive with the manually written variant. They even outperform the manual version in this application. All the incremental approaches beat the non-incremental one by a factor of $4 - 6X$. Interestingly, our approach to push some incremental computations to the storage manager improves `efficient-incr` by an extra 25%.

Overlap Iterative Processing: Figure 5 shows the effectiveness of the overlap processing and mini-iterations optimization in the context of the `SourceDetect` application, which requires overlap processing. `T1` refers to the policy where ArrayLoop shuffles overlap data at each iteration, or no `mini-Iteration` processing. As expected this approach incurs considerable data shuffling overhead, although it converges faster in the `SourceDetect` application (Figure 5). At the other extreme, we configure ArrayLoop to only shuffle overlap data after local convergence occurs in all the chunks. Interestingly, this approach performs worse than $T1$. Al-



non-incr	manual-incr	efficient-incr	efficient-incr+storage
40957	10975	8007	6096

Figure 4: Runtime of the `SigmaClip` application with and without incremental processing. Constant $k = 3$ in all the algorithms.



	T1	T5	T10	converge
Mini#	51	57	60	94
Major#	51	11	6	3

Figure 5: `SourceDetect` application: Iterative overlap processing with mini-iteration optimization. The table shows the number of major and mini iterations. Major# is the number of times that overlap data is reshuffled and Mini# is the total number of iterations.

though this approach does a minimum number of data shuffling, it suffers from the long tail of mini-iterations (Figure 5: 94 mini-iterations). $T5$ and $T10$ are two other approaches, where ArrayLoop shuffles data with some constant interval. We find that $T10$, which shuffles data every ten iterations, is a good choice in this application. The optimal interval is likely to be application-specific and tuning the value is beyond the scope of this paper.

8.1 Multi-Resolution Optimization

We omit the evaluation result for multi-resolution optimization due to space constraints but refer the reader to the full version of this paper for details [17].

9. RELATED WORK

Several systems have been developed that support iterative big data analytics [1, 6, 7, 16, 20]. Twister [3] and HaLoop [1] extend MapReduce to add a looping construct and preserve state across iterations. In contrast, we focus on iterative processing in a parallel array engine.

PrIter [20] is a distributed framework for fast iterative computation on top of MapReduce. ArrayLoop also supports a form of prioritized processing through multi-resolution optimization.

REX [11] is a parallel shared-nothing query processing platform implemented in Java with a focus on supporting incremental iterative computations in which changes, in the form of deltas, are propagated from iteration to iteration. Similar to REX, ArrayLoop supports incremental iterative processing. However REX lacks other optimization techniques that we provide and is not specialized for arrays.

A handful of systems exist that support iterative computation with focus on graph algorithms including Pregel [9] and GraphLab [7]. Similar to our work, GraphLab has a notion of *ghost* nodes. However, the granularity of computation is per node, while ArrayLoop supports overlap iterative processing per chunk. Our system also supports prioritization through the novel multi-resolution iterative processing.

10. CONCLUSION

We developed a model for iterative processing in a parallel array engine and presented three optimizations to improve the performance of these types of computations in the context of that model and type of engine.

Acknowledgments: This work is supported in part by NSF grant IIS-1110370 and the Intel Science and Technology Center for Big Data.

11. REFERENCES

[1] Y. Bu et al. HaLoop: Efficient iterative data processing on large clusters. *PVLDB*, 3(1), 2010.
[2] Cudre-Mauroux et. al. SS-DB: A Standard Science DBMS Benchmark. http://www-conf.slac.stanford.edu/xldb10/docs/ssdb_benchmark.pdf, 2010.
[3] J. Ekanayake et al. Twister: a runtime for iterative MapReduce. In *HPDC*, pages 810–818, 2010.
[4] E.Soroush, M.Balazinska, and D.Wang. ArrayStore: A storage manager for complex parallel array processing. In *SIGMOD*, pages 253–264, June 2011.
[5] Baumann et. al. The multidimensional database system RasDaMan. In *SIGMOD*, pages 575–577, 1998.
[6] S. Ewen et al. Spinning fast iterative data flows. In *VLDB*, pages 1268–1279, 2012.
[7] Y. Low et al. Distributed GraphLab: a framework for machine learning and data mining in the cloud. In *VLDB*, pages 716–727, 2012.
[8] Large Synoptic Survey Telescope. http://www.lsst.org/.
[9] G. Malewicz et al. Pregel: a system for large-scale graph processing. In *SIGMOD*, 2010.
[10] F. McSherry, D. G. Murray, R. Isaacs, and M. Isard. Differential dataflow. In *CIDR.*, 2013.
[11] S.R. Mihaylov et al. REX: Recursive, delta-based data-centric computation. In *VLDB*, 2012.
[12] UW-CAT. http://myria.cs.washington.edu/repository/uw-cat.html.
[13] J. Rogers et al. Overview of SciDB: Large scale array storage, processing and analysis. In *SIGMOD*, 2010.
[14] SciDB Guide. http://scidb.org/HTMLmanual/13.3/scidb_ug/.
[15] Scidb-py. http://jakevdp.github.io/SciDB-py/tutorial.html.
[16] Shaw et. al. . Optimizing large-scale semi-naive Datalog evaluation in Hadoop. In *In Datalog 2.0*, 2012.
[17] E. Soroush et al. Efficient iterative processing in the SciDB parallel array engine. Technical Report UW-CSE-15-06-01, University of Washington, 2015.
[18] Taft et. al. Genbase: A complex analytics genomics benchmark. In *SIGMOD*, pages 177–188, 2014.
[19] M. Zaharia et al. Spark: cluster computing with working sets. In *HotCloud'10*, 2010.
[20] Y. Zhang et al. PrIter: a distributed framework for prioritized iterative computations. In *VLDB*, 2011.
[21] Zhang et. al. RIOT: I/O-efficient numerical computing without SQL. In *Proc. of the Fourth CIDR Conf.*, 2009.