Hybrid Merge/Overlap Execution Technique for Parallel Array Processing

Emad Soroush
Dept. of Computer Science and Engineering
University of Washington, Seattle, USA
soroush@cs.washington.edu

Magdalena Balazinska
Dept. of Computer Science and Engineering
University of Washington, Seattle, USA
magda@cs.washington.edu

ABSTRACT

Whether in business or science, multi-dimensional arrays are a common abstraction in data analytics and many systems exist for efficiently processing arrays. As dataset grow in size, it is becoming increasingly important to process these arrays in parallel. In this paper, we discuss different types of array operations and review how they can be processed in parallel using two different existing techniques. The first technique, which we call *merge*, consists in partitioning an array, processing the partitions in parallel, then merging the results to reconcile computations that span partition boundaries. The second technique, which we call *overlap*, consists in partitioning an array into subarrays that overlap by a given number of cells along each dimension. Thanks to this overlap, the array partitions can be processed in parallel without any merge phase. We discuss when each technique can be applied to an array operation. We show that even for a single array operation, a different approach may yield the best performance for different regions of an array. Following this observation, we introduce a new parallel array processing technique that combines the merge and overlap approaches. Our technique enables a parallel array processing system to mix-and-match the merge and overlap techniques within a single operation on an array. Through experiments on real, scientific data, we show that this hybrid approach outperforms the other two techniques.

Categories and Subject Descriptors

H.2.4 [**Information Systems**]: Database Management−−−Systems; H.2.8 [**Information Systems**]: Database Management−−−Database applications

General Terms

Design, Performance

Keywords

Parallel Array Processing, Scientific Databases, Overlap Execution Strategy

AD 2011, March 25, 2011, Uppsala, Sweden.

1. INTRODUCTION

Multidimensional arrays arise as natural data types in many analytics scenarios from traditional OLAP applications [19, 28] to scientific data analysis [10, 27]. As a result, many engines have been developed to efficiently process such arrays [2, 10, 21, 30]. Additionally, as arrays grow in size, it is becoming increasingly important to process arrays efficiently in parallel [2, 21, 27]. In this paper, we propose a new parallel array processing method for existing array data management systems.

Some array operations are trivial to parallelize. These are operations that process array cells independently [27]. For example, a filter operation [21] copies an input array to the output but sets all cells that do not satisfy a predicate to null. To parallelize such an operation, one can simply divide a large array into multiple subarrays also called *chunks* [22] and process these chunks in parallel. Figure 1 illustrates a 2D image divided into four subarrays or chunks. Figure 2 shows how such an array could be processed in parallel (we come back to both figures later).

Other operations, however, are more challenging to execute in parallel. These are operations where each output cell value is based on the value of multiple input cells. These operations include traditional aggregations but also more complex array operations such as smoothing [21], regridding [6], feature extraction [13], and others. For such operations, two parallel processing techniques are commonly used. In the first technique, which we call *merge*, an array is spilt into multiple subarrays, each subarray is partly processed in parallel, and the intermediate results are merged [2, 4, 5, 10, 11, 13, 21]. In the second technique, which we call *overlap*, an array is split into multiple subarrays that overlap by a given number of cells in each dimension. The overlap data enables the system to handle corner-case cells without a second merge phase [21, 23, 24]. We describe these techniques in greater detail in Section 3.

A key challenge is in deciding which of these two techniques provides the best performance for a given array operation. Indeed, as we show in this paper, the best technique depends not only on the operation type but also on the data value distribution in the array.

To address this challenge, we develop a *hybrid merge/overlap* array processing technique. With our approach, an array is partitioned into multiple chunks. Each chunk can then be processed using *either the merge or the overlap technique*, independent of the technique chosen for the other chunks. As part of our approach, we develop (1) a simple, three function API that operator developers must

implement in order for their operation to be processed using our hybrid method, (2) a hybrid parallel array processing technique, and (3) implementation details for supporting this approach. We do not address the problem of which execution technique is selected for a given chunk. We leave the automated selection of the techniques for future work.

Depending on the application, arrays are either *dense* or *sparse*. An array is said to be sparse when most of its cells do not contain any data. Otherwise, the array is dense. An example dense array is a 2D image. For example, astronomers commonly operate on 2D telescope images. They clean, smooth, regrid, combine, and otherwise transform these images in order to extract observed celestial objects such as galaxies and stars [6, 16]. An example of a sparse array is the 3D data from an astronomy simulation [15, 26]. In these simulations, the universe is modeled as a set of particles. These particles are points in a 3D space and are simulated over a series of discrete timesteps. The simulator periodically outputs a 3D array of particles representing the state of the universe at a given simulation timestep. The approach that we propose is applicable to both array types and we use both these examples throughout the paper.

We evaluate our approach on a 3D astronomy simulation dataset [15]. We show that, in this dataset, different subarrays are most efficiently processed by different techniques and that the hybrid execution techniques leads to the fastest overall performance.

2. ARRAY OPERATION TYPES AND PROPERTIES

Different array processing techniques are applicable to different types of array operations. In this section, we identify five types of array operations that we classify along two axes as shown in Table 1.

Along the first axis, we distinguish between three types of array operations: *independent, bounded dependent, and unbounded dependent*:

Independent array operations are those that process each array cell independently of the others. More precisely:

DEFINITION 2.1. *An array operation is said to be independent if the value of each output array cell comes from the value of exactly one input array cell.*

Examples of independent operations include *filter*, *apply*, *slice*, and *subsample* [21]. Filter applies a predicate to the attribute values in the input array. The output array has the same dimensions, size, and content as the input array, except that cells where the predicate is found false are set to empty. Apply transforms the content of an array by applying a calculation to individual array cell values. Slice and subsample are structural operators but they still process each array cell independently of others. A slice operation projects an array along a particular index value in a single dimension. Subsample extracts a subarray from the input array.

Many array operations, however, are not that simple. Frequently, in order to compute the value of one array cell in the result array, an operation needs the values of multiple input array cells. Such neighborhoods of cells are sometimes called *stencils* (*e.g.*, [3]). We call these array operations *dependent.*

DEFINITION 2.2. *An array operation is dependent if multiple input array cells contribute to the value of each output array cell.*

Smoothing [21] and regridding [6] are typical such operations. A smoothing operation computes for each cell the weighted average of cell values in the neighborhood of that cell. Closer cells typically get higher weight. A regrid operation additionally compacts/expands cells, producing an array with fewer/more cells than the input array. In the rest of this paper, regrid examples refer to compacting regrid.

As a concrete example, given a telescope image in the form of a 2D array, astronomers may want to reduce its resolution such that cells collapse 10:3 [6]. To achieve this goal, a regrid operation needs to take each group of 10×10 cells in the input array to compute the value of each cell in a 3×3 group in the output array.

Depending on the type of operation, the number of cells needed to compute the value of an output cell can either be bounded, or unbounded. More formally, we introduce the following definition:

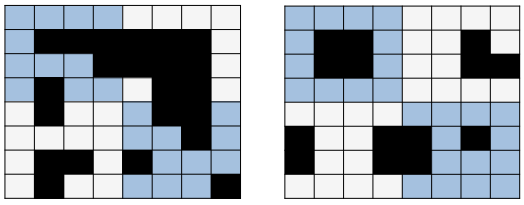
DEFINITION 2.3. *A dependent array operation is bounded if the input cells needed to compute the value of an output cell are within a pre-defined distance δ_i of the target cell along each dimension i of the input array. That is, to compute the value of cell at index $[x][y]$ in the output array, only cells in the input array that fall in the range: $[x-\delta_x, x+\delta_x][y-\delta_y, y+\delta_y]$ must be examined. Otherwise, the dependent operation is unbounded.*

In the case of smoothing or regridding of a dense array, the number of input cells that contribute to the value of each output cell is fixed and thus bounded. In the telescope image example, a group of 10×10 cells contributes to each output cell. For other operations, the number of cells may not be fixed but may still be bounded. For example, consider an operation that extracts clusters from telescope images, where a cluster is a connected set of bright pixels and corresponds to a celestial object such as a galaxy or a star. For each cluster, the operation computes and outputs the centroid. The input to this operation is a 2D array representing an image. The output is a 2D array with mostly null cells except for cells that correspond to the centroid of each cluster. To compute each cluster centroid in the output array, the system needs the values of all pixels contributing to the cluster. The number of pixels will change for each cluster. It can be bounded if clusters are known to have a bounded size or it can be unbounded if clusters are unbounded. Figure 1 illustrates these two cases.

When multiple cells contribute to the value of an output cell, these values can be aggregated using different functions. We distinguish two types of functions: *algebraic* and *holistic* and two corresponding types of array operations that we also call algebraic and holistic. This distinction forms the second axis of our classification and is based on the OLAP definitions [12]

DEFINITION 2.4. *A dependent array operation is algebraic if the value of each output cell is computed from the value of a set of input cells using an algebraic function F [12]. Additionally, the final result array must have a smaller state than the input array (*i.e.*, fewer cells, fewer non-null cells, or smaller state inside each cell).*

Gray *et al.* define *algebraic functions* in the context of OLAP data cubes as follows: "Aggregate function F() is algebraic if there is an M-tuple valued function G() and



(a) Unbounded clusters. (b) Bounded clusters

Figure 1: A 2D, 16×16 array divided into four 4×4 chunks and containing five clusters, which are groups of connected black cells. (a) Clusters do not have any bound on their size. A cluster can span the entire array. (b) Clusters are known to be bounded in size. A cluster is never larger than 2 cells along each dimensions.

a function H() such that $F(\{X_{i,j}\}) = H(\{G(\{X_{i,j}|i = 1,\ldots,I\})|j = 1,\ldots,J\}$. Average(), standard deviation(), MaxN(), MinN(), center of mass() are all algebraic. For Average, the function G() records the sum and count of the subset. The H() function adds these two components and then divides to produce the global average." [12]

Hence, a function is algebraic, if it is possible to aggregate the values of subsets of cells and then combine these partial results into a final output cell value. For example, if an operation identifies clusters in a 2D image and extracts their centroids, the operation can compute the count and sum of the coordinates of pixels in each of N subsets of a cluster. It can then average these partial sums to get the cluster centroid.

In addition to this standard definition of an algebraic function, however, we add the following requirement: We consider an array operation to be algebraic if the output array has smaller state than the input array. This extra requirement is important for parallel processing because it means that the state from partial computations can be combined incrementally and each combination step will reduce the total state size. Smoothing is an operation that illustrates this distinction. Consider a 2D array that we want to smooth by computing for each output cell the average value of its 8 direct neighbors. This function is algebraic in the OLAP sense since we can sum the values of subsets of neighbors of a cell and then average these partial sums to compute the output cell value. The overall operation is not algebraic in our sense, however, because the final array has the same size as the original array.

If an array operation is not algebraic, we call it *holistic*. Examples of holistic array operations include computing the *medoid* of a cluster or returning all cluster data points annotated with the cluster identifier. For many holistic array operations, all input cell values are needed to compute the output value of a cell. *Median*, *mode*, and *rank* are all examples of holistic functions.

Table 1 summarizes these five types of array operations. When an operation processes an input cell to produce one output cell, we call the operation *independent*. When an operation consumes multiple input cells, we call the operation either *bounded dependent* or *unbounded dependent* depending on the number of input cells that it needs. The latter two types of operations are further divided into either *algebraic* or *holistic* operations depending on whether they use an algebraic or holistic aggregation function and whether or not they produce a smaller output array than the input array. The table also shows example applications with each combination of properties.

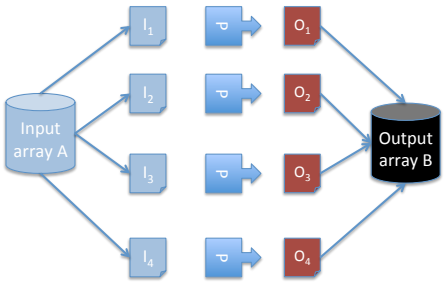


Figure 2: Array A is divided into four chunks I_1 through I_4. Each chunk I_i is given to the independent process P as input and generates output chunk O_i. Chunks O_1 through O_4 form the output array B.

3. EXISTING PARALLEL ARRAY PROCESSING TECHNIQUES

In this section, we discuss how the above types of array operations can be processed in parallel. Table 2 repeats the array operation taxonomy from the previous section and indicates, for each type of operation, the applicable array processing techniques. The table shows three techniques that we discuss in this section: *Independent*, *Merge*, and *Overlap*. All three techniques have previously been proposed in the literature. Our contribution is to show how two of these techniques: merge and overlap can be combined into a new, hybrid strategy that we present in the next section.

3.1 Independent

Array operations that transform each cell of an input array into one cell in the output array (*a.k.a.*, independent operations) are trivial to parallelize: the array processing system can partition the array into chunks as shown in Figure 2 and each chunk can be processed in parallel, independent of the other chunks, producing one chunk of the result array. We call this technique the *independent array processing* technique and do not further consider it in this paper.

3.2 Merge

For dependent operations, the above processing strategy fails: if an array is partitioned into chunks and each chunk is processed separately from the others, data will be missing to compute the values of output cells whose input cells are spread across multiple chunks.

For dependent operations, an alternate parallel processing technique is thus commonly used. The key idea behind this approach, as shown in Figure 3, is to partition an array into chunks, *partly* process these chunks independently in parallel, then *merge* (*a.k.a.*, roll-up or post-process) the intermediate results to produce the final output [1, 4, 5, 10, 11, 13, 14, 21]. The merge phase serves to compute the value of array cells whose input crosses chunk boundaries.

In the example of image smoothing, this approach works as follows: The system partitions the array into chunks and applies smoothing to all cells within each chunk except those

Table 1: Types of array operations and example applications.

	Independent	Bounded Dependent	Unbounded Dependent
Algebraic	N/A	Regrid, Cluster centroids (bounded-size clusters)	Cluster centroids (clusters with no bound on size)
Holistic	Filter, Slice	Smooth, Cluster medoids (bounded-size clusters)	Cluster medoids (clusters with no bound on size)

Table 2: Array execution techniques. For holistic functions, a naïve merge technique can fail.

	Independent	Bounded Dependent	Unbounded Dependent
Algebraic	N/A	Merge & Overlap	Merge
Holistic	Independent	Merge* & Overlap	Merge*

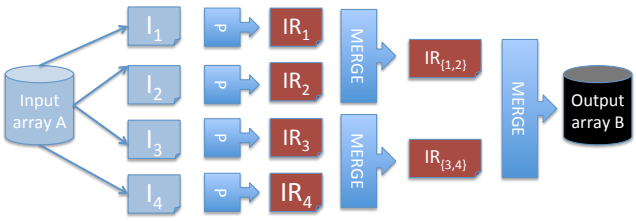


Figure 3: Array A is divided into four chunks I_1 through I_4. Each chunk I_i is given to the process operation P as input and generates an intermediate result IR_i. Intermediate results IR_1 through IR_4 are merged hierarchically until they generate the output array B.

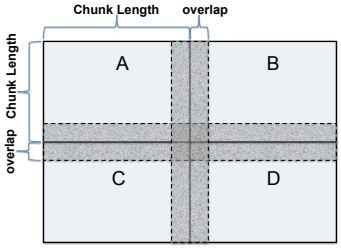


Figure 4: Example of four overlapping chunks A, B, C, and D. The overlapping region is shaded and the boundary of chunks are specified by dashed lines.

cells near a chunk boundary. To compute the value of the remaining cells, a second merge phase occurs. During that phase, the system shuffles data near chunk boundaries such that all data necessary to compute the value of each boundary cell goes to a single location, where the computation can occur.

In the cluster extraction application, the input array is a 2D image of pixels. The operation identifies clusters of bright pixels and returns either their centroid, their medoid, or the original data annotated with a cluster identifier. Such a cluster extraction operation can also be done in two phases. First, the system partitions the image into chunks and extracts clusters inside each chunk. Clusters that cross chunk boundaries are then reconciled during the merge phase.

The exact type of data that must be exchanged between chunks and reconciled during the merge phase depends on the type of array operation. For holistic operations –such as extracting the medoid of a cluster or tagging all input data points with the identifier of their cluster– the original input data that contributes to each output cell must be brought together. For algebraic operations –such as extracting the centroid of a cluster– partial aggregate values can suffice. For example, the partial sums of cell coordinates for a cluster suffice to eventually compute the centroid of the cluster. The type of the array operation thus determines the size of the state processed during the merge.

One approach to performing the merge is to send all intermediate data to a single, centralized location. The problem, however, is that the size of the state to be merged can overwhelm the node performing the merge operation. Instead, another approach is to perform the merge hierarchically [13], where increasingly large neighborhoods of chunks are reconciled. This approach works well for algebraic operations because each merge step should reduce the total size of the intermediate state. Holistic functions still suffer from the problem that the state to merge grows as increasingly many chunks are being reconciled and it is possible that a single node may end-up processing a significant fraction of the input data.

Different strategies are possible to ensure an efficient merge of holistic functions. One approach is to materialize to disk as much intermediate data as possible to keep the merged-state small. For example, when extracting clusters from a dataset, only data at cluster boundary is necessary to reconcile clusters across chunks. If the operation is to also tag the input data with cluster identifiers or extract the cluster medoids, the cluster data can be set aside during the merge phase and only processed at the end once the final clusters are found [13]. We do not implement such optimizations in the paper but our approach could be extended to include them.

3.3 Overlap

For bounded dependent operations, an alternate processing technique also exists. This approach is based on creating and using *overlap* [21, 23] data between chunks. The key idea behind this technique is to partition an array into multiple subarrays that overlap by a certain number of cells along each dimension as illustrated in Figure 4. Each partition can then be processed independently of the others because all needed data is available either in the form of core or overlap cells. This approach can potentially, though not necessarily, be faster than the two-phase merge-based approach. It works especially well when the amount of overlap data needed is much smaller than a chunk size.

In the example of image regridding from Section 2, we want to change the resolution of an image by collapsing cells

10:3 . Hence, if we partition the image array with an overlap of 10 cells along each dimension, the value of each output cell can be computed using locally available core and overlap data.

Similarly, in the case of the cluster extraction application, if clusters are always smaller than a certain maximum size, we can partition an image array with an overlap equal to that maximum size. Each partition can then be processed separately from the others. To avoid duplicates, each cluster can be reported by the partition that holds its centroid.

4. A HYBRID PARALLEL ARRAY PROCESSING TECHNIQUE

The merge and overlap techniques present a fundamental trade-off in how an array is processed. With the overlap technique, each array chunk uses up more space on disk because of the extra overlap data and takes longer to process for the same reason. In contrast, with merge, individual chunks are smaller and are processed faster but intermediate data, which can be large, must later be transferred between processing nodes and merged. Depending on the type of array operation and also on the input data, one approach can thus be faster than the other.

In the case of unbounded operations, only the merge parallel processing strategy can safely be applied. However, for many operations, not all output cells are based on data from a large number of input cells. For example, most clusters in an image are likely to be small even though there is no theoretical bound on the cluster size. Hence, a large fraction of an array can successfully be processed using overlap. If overlap provides higher performance given the operation and input data, processing part of an array using this technique can outperform a uniform strategy based on a hierarchical merge over the entire array.

Motivated by these two observations, we propose a hybrid approach that enables a system to combine the merge and overlap strategies while processing a single array. In this section, we present the API that this hybrid technique exposes to developers. When an array operation implements all operations in this API, our system can automatically apply that operation to the array using the hybrid processing method. We then present the hybrid execution technique, the back-end implementation necessary to support it, and an additional full-length example.

4.1 API

In order for an array operation to be processed using a hybrid merge/overlap strategy, the operation developer must implement the three functions shown in Table 3.

Process. The `process` function performs the core operation. It takes as input an array chunk (`Input_chunk`). It produces two outputs: a `Result_chunk` and a `Merge_chunk`. Either one can be null. The `Result_chunk` is a chunk of the output array, possibly in some intermediate state, such that the final result will be the union of all `Result_chunks`. The `Merge_chunk` contains data for the given input chunk that needs to be further processed by combining it with *adjacent* `Merge_chunks` before updating the corresponding `Result_chunks`.

In the image smoothing or regridding examples, the process function takes as input an array chunk. It computes the smoothed or regridded values for cells in `Result_chunk` for which all input data is locally available. It leaves the other cells null. The process function then copies all data near the chunk boundary into the `Merge_chunk`.

For the application that identifies clusters and labels input data with cluster identifiers, `process` outputs the labeled data in the `Result_chunk` for all clusters fully contained in the chunk. It re-copies the data for clusters that span chunk boundary into the `Merge_chunk`. Figure 5 illustrates the output of the process phase for this operation applied to a 4-chunk array.

In the case of the cluster extraction application that computes cluster medoids, the `process` function identifies clusters inside a chunk. For each cluster fully contained in the chunk, process registers its medoid in the `Result_chunk`. If a cluster spans chunk boundary and cannot be processed locally, its corresponding, partial input data is copied over to the `Merge_chunk`. In contrast, for the cluster extraction operation that computes cluster centroids, `Merge_chunk` need not contain all input data. Instead, it needs only hold a pre-aggregated summary of that data: *i.e.*, sum of cell coordinates and the range of cells at the chunk boundary that can potentially extend the cluster into an adjacent chunk.

In order to support overlap processing, when implementing the `process` function, the developer can call two functions:

```
Range max_overlap()
Chunk get_overlap(range)
```

Function `max_overlap` returns a range predicate indicating the overlap area that is available, if any. Given a range r expressed as a predicate over the array index values, `get_overlap` returns the overlap that falls within this range. The overlap data takes the form of an array chunk with null cells in the place of the core data.

Merge. The `merge` function takes as input the output of a set of `process` functions executed on *adjacent* chunks. It merges intermediate results as needed and updates the corresponding `Result_chunks` and `Merge_chunks`.

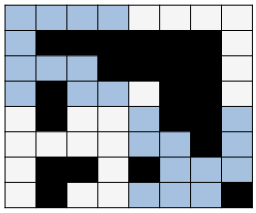
In the smoothing and regridding examples, `merge` combines the data from adjacent `Merge_chunks`. It uses that newly combined data to compute the value of the missing cells in the `Result_Chunk`.

Similarly, for our running image cluster extraction application, `merge` identifies clusters that span partition boundaries. Figure 5 illustrates the merge process for the cluster extraction application that labels the output data with cluster identifiers. In this example, array A is divided into 4 chunks. Each chunk is given to the `process` function and produces a pair of (`Result_chunk`, `Merge_chunk`). All the pairs are fed into the `merge` phase in order to generate the final result. In this example, there are two levels of merges. The first level merges chunks along the `X` dimension. The second level merges along the `Y` dimension. At each step, merge combines the data from adjacent `Merge_chunks`. If it finds a new cluster that is now fully discovered, it outputs the corresponding labeled data into the appropriate `Result_chunks`. If a cluster is still not fully contained in the available chunks, its data goes back to the corresponding `Merge_chunks`.

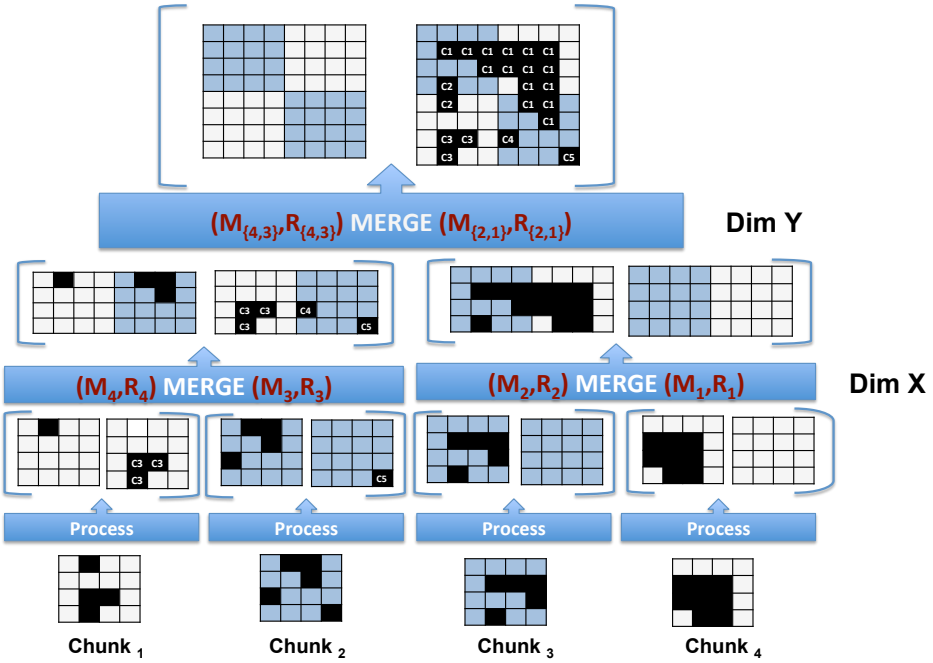
Filter. The process and merge functions above are sufficient to implement *either* the merge or overlap techniques. In order to mix-and-match these techniques, our framework requires that a user provides one more function. The `filter` function takes as input a chunk with intermediate re-

Table 3: Hybrid Execution Technique API

Output	Function Name	Input
(Result_Chunk, Merge_Chunk)	process	Input_chunk
{ (Result_Chunk, Merge_Chunk) }	merge	{ (Result_Chunk, Merge_Chunk) }
Merge_Chunk	filter	(Merge_Chunk, Bitmap)



(a) Array A contains five clusters shaded in black and is split into four chunks.



(b) `Process` and `merge` phases of the cluster extraction operation. The `Merge_chunks` (left) and `Result_chunks` (right) are shown at each step. As clusters are discovered, the corresponding cells are labeled with a unique identifier and copied over to `Result_chunks`.

Figure 5: Extracting clusters from array A. The output array is a copy of the input array but the operation labels each cell inside a cluster with the unique identifier of that cluster. Each of array A's chunks is going through the `process` (clustering) phase and generates intermediate clusters in a `Merge_chunk` (M_i) and final clusters in a `Result_chunk` (R_i). `Results_chunks` and `Merge_chunks` are given to the *hierarchical merge*. Each $M_{i,j}$ is a `Merge_chunk` that contains intermediate results from original chunks M_i and M_j. Similarly for each $R_{i,j}$.

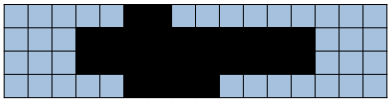
sults that need to be merged and a bitmap indicating if the surrounding chunks in the same array were processed using either the overlap or merge techniques. This function filters out all results that do not need to be merged because an adjacent chunk was processed using overlap data and already incorporated these results into the final output. When implementing function `filter`, developers have access to the the `max_overlap` and `get_overlap` methods described above. We illustrate how the filter function is used with an example in Section 4.2.

4.2 Execution Technique

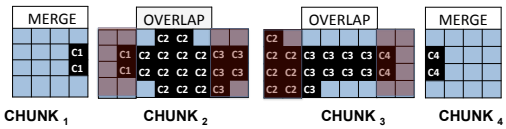
We now describe how the hybrid parallel execution technique processes an array. We use Figure 6 as illustrative example. Figure 6(a) represents a (4×16) array A with a large cluster shaded in black.

First, the array is split into chunks as shown in Figure 6(b) and each chunk is labeled as either "merge" or "overlap". Labeling the chunks can be scheduled as an offline task. An auxiliary system could derive the right labeling of a given chunk based on the data distribution of the chunk and its neighboring chunks. In this paper, we do not address the problem of how this choice is made. We leave the automated selection of the techniques for future work.

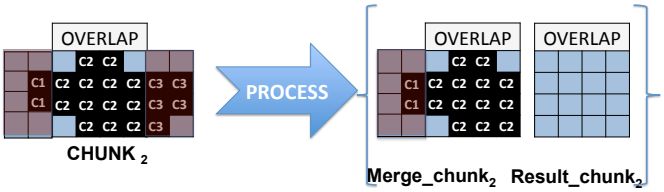
Second, the `process` function is applied to each chunk. For chunks that are to be processed with overlap, overlap data is available from within the process function through calls to `max_overlap` and `get_overlap`. Otherwise, `max_overlap` returns an empty interval. Maximum overlap size provided in this example is two cells. As above, each `process` function outputs a `Merge_Chunk` and a `Result_Chunk`. Normally, with the overlap approach, `Merge_Chunks` are always null because there is no merge phase. However, because one can use our hybrid strategy with unbounded dependent operations, overlap processing may sometimes fail. As shown in Figure 6(c), if a cluster exceeds the maximum available overlap data, `process` does not output anything in the `Result_Chunk` but instead places the cluster data into the `Merge_Chunk`. Furthermore, it only outputs the core data plus the fully processed overlap data. For example in Figure 6(c), `process` function only outputs core and overlap cells that it labels with C_2 and C_1 into the



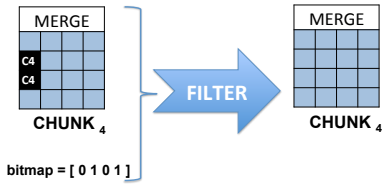
(a) Original array A with one cluster shaded in black.



(b) Array A is divided into 4 chunks. Each is labeled with either `merge` or `overlap`. Overlap cells are highlighted.



(c) The result of the `process` phase in $chunk_2$ does not extract any final result cluster because the locally-found cluster possibly stretches past the overlap region. Instead, the cluster data is copied into the Merge_Chunk.



(d) Intermediate cluster in $chunk_4$ is removed in `filter` operation, because it is entirely inside the overlap region of the adjacent $chunk_3$ and already incorporated into the final output of this latter chunk.

Figure 6: Example hybrid execution technique on a 4×16 array with a single large cluster.

Merge_chunk. All the overlap cells labeled with C_3 need to be removed from the output cluster. In those cases, a portion of the overlap processing time is thus wasted. This waste adds overhead to the hybrid strategy whenever the choice of overlap *v.s.* merge is not appropriate.

Third, the output of each `process` function is pre-processed with `filter`. Figure 6(d) illustrates a case where the intermediate result is filtered out. The `process` operation applied to $chunk_4$ identifies a cluster that it labels with C_4. This cluster potentially extends into the adjacent chunk $chunk_3$ and is thus placed into the `Merge_Chunk`. The `Filter` operation, however, removes all C_4 tagged data from the `Merge_Chunk` because there is an adjacent chunk, $chunk_3$, that was processed with the overlap method and has already incorporate the C_4 data into its final result. In order to perform this check, the `filter` function must access overlap data for $chunk_4$. Indeed, if C_3 data did not exist, C_4 data should not be filtered out.

Finally, intermediate data from *neighboring* chunks is merged as in the vanilla merge approach, except that for all `Merge_Chunks`, the merge phase must also process any overlap data possibly output by an earlier `process` function.

4.3 Back-End

Finally, we present the necessary back-end support for the above hybrid parallel execution technique. This support is a set of extensions that must be implemented on top of a parallel array processing system. We assume a parallel array processing system such as SciDB [7, 21] that splits an array into chunks and has the ability to process these chunks in parallel.

Overlap-enabled storage manager. In order to support overlap-based processing, an array processing system must have a storage manager that provides access to overlap data for each array chunk. There are several possible implementations. The storage manager can store overlap data together with core data [21, 23]. This approach, however, is inefficient because overlap data must always be read from disk and processed whether it is necessary for an operation or not. Alternatively, the overlap data can be stored separately. This optimization helps operations that do not use overlap data. However, operations that need the overlap face the problem of having access to a single overlap region, which must be large-enough to satisfy all queries. To address both challenges, in recent prior work [24], we developed a technique where overlap data is stored as a set of materialized overlap views. These views are defined like a set of onion-skin layers around chunks: *e.g.*, layers L_1 through L_3 in Figure 7. A view definition takes the form $(n, w_1, \ldots, w_d)$, where n is the number of layers requested and each w_i is the thickness of a layer along dimension i. Multiple views can exist for a single array. With this approach, an operator processing a chunk can request exactly as much overlap data as it needs, which we found to be more efficient than alternate implementations. In our experiments we use the overlap processing technique where overlap data is stored as a set of materialized overlap views.

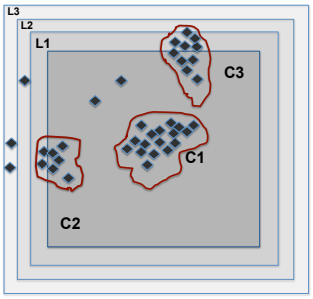


Figure 7: Example of multi-layer overlap in a sparse array. In this array, extracting cluster $C2$ necessitates that the operator loads a small amount of overlap data denoted with $L1$. $C3$ requires an additional overlap layer, $L2$. None of the clusters requires the third layer $L3$.

Overlap-enabled processing. The storage manager must implement the `max_overlap` and `get_overlap` functions described above.

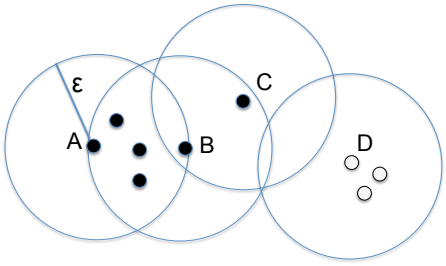


Figure 8: FOF clustering algorithm: A and B are friends and B and C are friends, but A and C are not. A and C hold the Friend of Friend relation (FoF) via B. The FoF relation induces a partition on the particles: all black points are in one cluster and all white points are in another cluster.

Hybrid parallel array-processing executor. The system executor must be extended to take as input an array divided into chunks and a bitmap indicating which chunks should be processed with overlap and which chunks should be processed with merge. For the chunks processed with overlap, the executor must make overlap data available. For other chunks, overlap data must be hidden and `max_overlap` should return an empty range. The executor must also be extended to apply the `filter` function to all `Merge_Chunks`. Finally, the filtered `Merge_Chunks` must be merged and the corresponding `Result_Chunks` must be updated through a hierarchical application of the `merge` function. Hierarchical aggregation techniques are commonly used in various systems, including multidimensional data processing systems [13, 14].

Overall, the hybrid execution strategy requires only a few changes to a parallel array processing system and should thus be reasonably practical to add to any system.

4.4 Additional Example

The Friends-of-Friends (FoF) algorithm (c.f.Davis et al. 1985 [8] and references therein) has been used in cosmology for at least 20 years to identify interesting objects and quantify structure in simulations [25, 20]. FoF is a simple clustering algorithm that accepts a list of particles (pid; x; y; z) as input and returns a list of cluster assignment tuples (pid; clusterid). To compute the clusters, the algorithm examines only the distance between particles. FoF defines two particles as friends if their euclidean distance is less than a certain threshold ϵ, named *friendship threshold*. Two particles are friends-of-friends if they are reachable by traversing the graph induced by the friend relationship. To compute the clusters, the algorithm computes the transitive closure of the friend relationship for each unvisited particle. All particles in the closure are marked as visited and linked as a single cluster. Figure 8 illustrates this clustering algorithm. FOF can be implemented in our framework as follows:

First, the implementation of the `process` operation in FOF algorithm is straightforward. We run the original implementation of the FOF algorithm on each input chunk and produce pairs of `Result_chunk` and `Merge_chunk` as a result. The `Result_chunk` and the `Merge_chunk` comprise all the finalized and intermediate FOF clusters, respectively. FOF cluster is considered as intermediate if it is either close enough to the chunk boundary (less than the friendship threshold) or it exceeds beyond the maximum overlap size for chunks labeled with "overlap".

Second, the `filter` function in FOF clustering algorithm checks whether the intermediate clusters are incorporated into the result of any adjacent chunk. If the answer is 'yes', that cluster is dropped and no further processing is required for that particular cluster. Three conditions must hold in order for the `filter` function to drop a cluster. First, the cluster must be entirely inside the overlap region of some adjacent chunk $Chunk_a$. Second, $Chunk_a$ must be labeled with "overlap". Third, there exists one point in $Chunk_a$ that is friend of at least one point in the examined cluster.

Finally, our implementation of the `merge` function is similar to the approach described in [14]. It consists in performing a hierarchical merge. Each `merge` function invocation receives a set of `Merge_chunks` as inputs and checks whether any clusters are *mergeable*. Two clusters are mergeable if there is a friend relationship between any two points of those two clusters. If two clusters are mergeable, they are merged into one cluster. If the newly formed cluster is contained entirely in the union of the `Merge_chunks` being processed, that cluster is processed just like any other cluster and the result is written into the appropriate `Result_Chunk`. Otherwise, the intermediate data for the cluster must be put into the appropriate `Merge_Chunk(s)` for further processing.

Because the output of FoF is the list of all input data points, labeled with the identifiers of the clusters where they belong, FoF is a holistic function with a potentially expensive merge phase. One possible optimization for the merge phase is to only keep inside a `Merge_chunk`, the points that are potentially mergeable for a given intermediate cluster [13]. Distance of those points to the chunk boundary must be less than the friendship threshold. We name those points *boundary points*. The rest of the points could be materialized and added to the final cluster at the end of the hierarchical merge operation. This optimization makes the `merge` phase significantly faster.

FoF is either a bounded or an unbounded dependent operation depending on whether the clusters in the underlying input data are bounded in size or not.

5. EVALUATION

In this section, we evaluate the performance of the hybrid strategy on one real dataset from the astronomy simulation domain. All the experiments are run on a dual quad-core 2.66GHz Intel/AMD OpteronPentium-based machine with 16GB of RAM running RHEL5. The dataset is snapshot $S92$ from the University of Washington N-body group cosmo25 simulation [15, 26]. The dataset is 37GB in size. The simulation models the evolution of cosmic structure from about 100K years after the Big Bang to the present day. Snapshot $S92$ is an early snapshot in this simulation. The snapshot represents the universe as a set of particles in a 3D space, which naturally leads to the following schema: `Array Simulation {id,vx,vy,vz,mass,phi} [X,Y,Z]`, where X, Y, and Z are the array dimensions and `id, vx, vy, vz, mass, phi` are the attributes of each array cell. This array is sparse. Most cells are null, except for cells that contain a particle. `id` is a unique particle identifier. It is a signed 8 byte integer while all other attributes are 4 byte floats and correspond to particle attributes such as velocity and mass. For the experiments, we partition the array into $(16 \times 16 \times 4)$ chunks. For the overlap and hybrid execution techniques, we

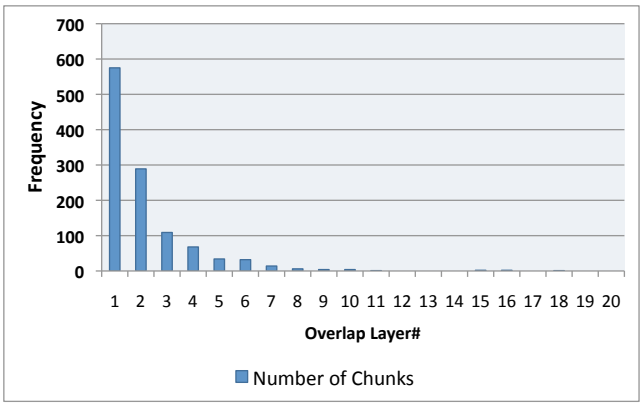


Figure 9: Overlap layer histogram for bounded FOF application.

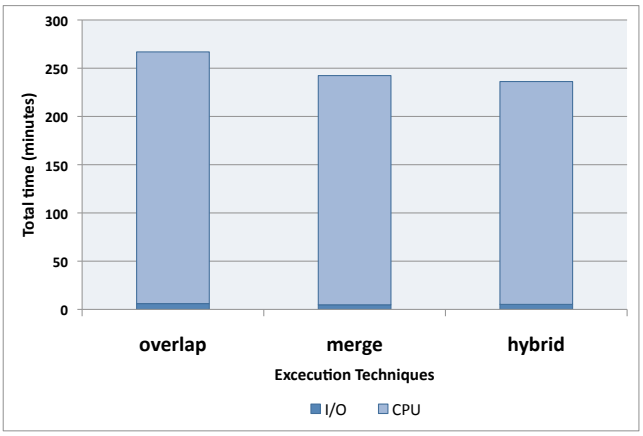


Figure 10: Total time for different execution techniques to run FOF algorithm.

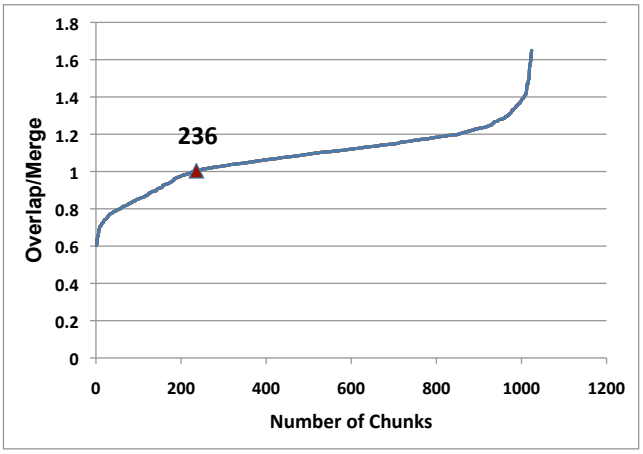


Figure 11: Difference in total time to process and load each chunk between "overlap" and "merge" techniques. y-axis plots Δt = (time(overlap)/time(merge)). Chunks are sorted by increasing Δt value. "overlap" outperforms "merge" for 20% chunks

materialize 20 layers of overlap data for each chunk, which cover a total of 0.5 of each dimension length. 20 layers is an arbitrarily chosen large value. Note that this is the maximum overlap size provided by the system to the developers.

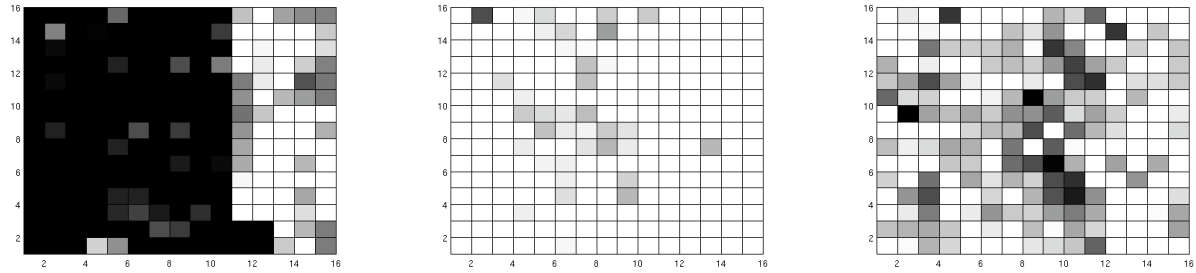
Bounded Dependent Array Operation. We run the experiment for the bounded version of the FOF algorithm. In order to make the problem bounded, we choose a small friendship threshold such that all the final clusters can be found solely using the overlap execution technique (no cluster exceeds the boundary of maximum range of overlap). The threshold is approximately $\frac{1}{2000}$ of a dimension length. Figure 9 shows the count of the number of chunks that needed a certain number of overlap layers in order to be processed using the "overlap" technique. The figure shows that more than 85% of chunks only need 3 out of the 20 layers of overlap, but the maximum overlap required is 18.

Figure 10 illustrates the performance of "overlap", "merge", and "hybrid" execution techniques. The figure shows the total time for each technique as the sum of processing times of all `process`, `merge`, and `filter` invocations, which is equivalent to running all operations in series. Because we picked a small friendship threshold, most clusters are small, and the `merge` phase is relatively cheap. It is interesting to observe that although "overlap" produces an overall worse performance than "merge", "hybrid" outperforms "merge" in total processing time. In order to understand why "hybrid" is a better approach than "merge", we need to look at the time at a finer granularity. Figure 11 reveals the reason why "hybrid" is outperforming "merge". Although "merge" performs better overall, "overlap" outperforms "merge" for 235 out of 1145 chunks, which is almost 20% of the total chunks processed. Intermediate clusters keep track of meta information such as cluster boundaries in order to do the `merge` operation during the next phase. When multiple clusters need to be merged, it is quickly more expensive to create and merge intermediate clusters than process some extra overlap data. This is the main reason why in a portion of chunks "overlap" technique is superior to the "merge" technique.

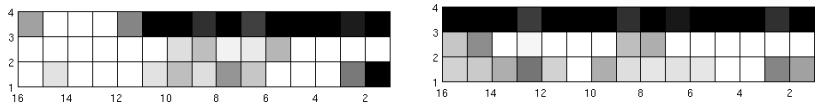
Figure 12 illustrates the result of running FOF with two techniques "overlap" and "merge" on different array slices. Black cells represent chunks where "overlap" outperforms "merge" and white cells are the ones where "merge" outperforms "overlap". It is interesting to see that "overlap" technique is outperforming in more than half of the chunks in the X-Y slice when Z=1. "Merge" is dominating in the X-Y slice when Z=2 and in the rest of the sample slices we observe a mix trend. We thus see clusters of chunks where each technique wins depending on the data distribution in these portions of the array.

6. RELATED WORK

Many engines are being built today to support multidimensional arrays [2, 4, 5, 10, 11, 21, 30]. Many of them supports parallel array processing [2, 4, 5, 10, 11, 21]. A well-studied approach for processing array operations in parallel is to divide the original array into multiple subarrays and run the operation independently on each subarray. If necessary, the results from local computations are post-processed (*a.k.a.*, "rolled-up" or "merged") to obtain the final output [2, 4, 5, 10, 13, 14, 21]. Additionally a few engines [21] also have the ability to run queries with overlap and potentially avoid



(a) X-Y slice of the array with Z=1. (b) X-Y slice of the array with Z=2. (c) X-Y slice of the array with Z=4.



(d) Y-Z slice of the array with X=8. (e) X-Z slice of the array with Y=8.

Figure 12: Different kinds of slices of the 3D array A with $(16 \times 16 \times 4)$ **chunks. Black cells represent chunks where "overlap" outperforms "merge". Vice versa for white cells. Any shading between white and black means** Δt **= (overlap-merge) is less than 1 second. In 12(a) "overlap" dominates, in 12(b) "merge" dominates, and the rest is a mix.**

the merge processing overhead, but none of the previous approaches has considered a hybrid strategy.

Traditional MOLAP systems store data in multidimensional arrays [19, 28]. Their workload comprises primarily aggregate queries. Aggregate queries can be processed with a two-phase, merge-style approach [11, 12]. To the best of our knowledge, however, none of the MOLAP systems supports overlap processing.

SciDB [21] is a multidimensional array system which supports overlap processing. However, overlap data is co-located with core data and each function is executed either with overlap usage or without it on a given array. SciDB does not have the capability to execute a non-consecutive portion of the array with overlap and the rest without overlap.

Loebman *et al.* [15] study the performance of parallel databases versus data processing systems such as MapReduce [9] in one specific scientific application: massive astrophysical simulations. Their use case comprises five representative selection and join queries. However, they do not study overlap processing and how this execution strategy would affect the overall performance.

Parallel programming languages such as UPC [29], Global Arrays [17], and Co-Array Fortran [18] faciliate the task of coding parallel array processing applications. Those parallel languages provide flexible access to remote array partitions providing the abstraction of a shared memory address space. They target applications where a given local chunk may need to access an arbitrary set of remote chunks. In contrast, we study "overlap", "merge", and "hybrid" execution techniques, which provide more loosely coupled parallelism for applications where the computation of a given chunk is restricted to its adjacent chunks in the context of shared nothing architectures.

Seamons and Winslett [23] studied different array storage strategies. They propose to store overlap data either separately or together with core data. However, their implementation, similarly to SciDB [21], only considers the co-located option. In prior work, we introduced a storage manager for complex, parallel array processing called ArrayStore [24]. ArrayStore efficiently supports overlap array processing through materialized overlap views. ArrayStore enables operators to efficiently access data from adjacent array fragments during parallel processing. In that prior work, we showed that an efficient overlap strategy could significantly outperform approaches that do not use overlap, but we did not study hybrid execution techniques.

7. CONCLUSION

In this paper, we presented a hybrid parallel array processing technique. Our approach enables an array data management system to combine two existing processing methods, merge and overlap, into one by selecting either option for any subset of an array. As part of this hybrid processing approach, we introduced an API, an execution method, and the back-end implementation necessary to support the approach. Through experiments on a real dataset from the astronomy simulation domain, we showed that the hybrid technique outperforms either uniform one.

Future work includes evaluation of the "hybrid" execution technique on unbounded parallel array operations along with the effort to address the problem of automated selection of the execution techniques.

8. ACKNOWLEDGEMENTS

The astronomy simulation dataset was graciously supplied by Tom Quinn and Fabio Governato of the University of Washington Department of Astronomy. The simulation was produced using allocations of advanced NSF-supported computing resources operated by the Pittsburgh Supercomputing Center, NCSA, and the TeraGrid. We thank Jeffrey P.

Garner and YongChul Kwon for their help with the astronomy simulation data and the FoF algorithm. We thank the SciDB team for insightful discussions and the anonymous reviewers for helpful comments on drafts of this paper. This work is partially supported by NSF CDI grant OIA-1028195, gifts from Microsoft Research, and Balazinska's Microsoft Research New Faculty Fellowship.

9. REFERENCES

[1] `http://mahout.apache.org/`.

[2] Ballegooij et. al. Distribution rules for array database queries. In *16th. DEXA Conf.*, pages 55–64, 2005.

[3] G. Berti. A calculus for stencils on arbitrary grids with applications to parallel PDE solution. In *Proc. of GAMM Workshop "Discrete Modelling and discrete Algorithms in Continuum Mechanics"*, pages 37–46, 2001.

[4] Chang et. al. T2: a customizable parallel database for multi-dimensional data. *SIGMOD Record*, 27(1):58–66, 1998.

[5] Cohen et. al. Mad skills: new analysis practices for big data. *vldbj*, 2(2):1481–1492, 2009.

[6] P. Cudre-Mauroux, H. Kimura, K-T. Lim, J. Rogers, S. Madden, M. Stonebraker, S. Zdonik, and P. Brown. SS-DB: A standard science DBMS benchmark. Under submission.

[7] Cudre-Mauroux et. al. A demonstration of scidb: a science-oriented dbms. In *Proc. of the 35th VLDB Conf.*, pages 1534–1537, 2009.

[8] M. Davis, G. Efstathiou, C. S. Frenk, and S. D. M. White. The evolution of large-scale structure in a universe dominated by cold dark matt er. *Astroph J*, 292:371–394, May 1985.

[9] Jeffrey Dean and Sanjay Ghemawat. MapReduce: simplified data processing on large clusters. In *Proc. of the 6th OSDI Symp.*, 2004.

[10] Baumann et. al. The multidimensional database system RasDaMan. In *Proc. of the SIGMOD Conf.*, pages 575–577, 1998.

[11] S. Goil and A. Choudhary. Parsimony: An infrastructure for parallel multidimensional analysis and data mining. *Journal of Parallel and Distributed Computing*, pages 285 – 321, 2001.

[12] J. Gray, S. Chaudhuri, A. Bosworth, A. Layman, D. Reichart, M. Venkatrao, F. Pellow, and H. Pirahesh. Data cube: A relational aggregation operator generalizing group-by, cross-tab, and sub-totals. *dmkd*, 1(1):29–53, 1997.

[13] Y. Kwon, M. Balazinska, B. Howe, and J. Rolia. Skew-resistant parallel processing of feature-extracting scientific user-defined functions. In *Proc. of SOCC Symp.*, June 2010.

[14] Y. Kwon, D. Nunley, J.P. Gardner, M. Balazinska, B. Howe, and S. Loebman. Scalable clustering algorithm for N-body simulations in a shared-nothing cluster. In *Proc of 22nd SSDBM*, 2010.

[15] S. Loebman, D. Nunley, Y. Kwon, B. Howe, B. Balazinska, and J.P. Gardner. Analyzing massive astrophysical datasets: Can Pig/Hadoop or a relational DBMS help? In *Proceedings of the Workshop on Interfaces and Architectures for Scientific Data Storage*, 2009.

[16] LSST data management: DC3b processing flow. `http://dev.lsstcorp.org/trac/wiki/DC3bProcessingFlow`.

[17] J. Nieplocha, R.J. Harrison, and R.J. Littlefield. Global arrays: a nonuniform memory access programming model for high-performance computers. *J. Supercomput.*, June 1996.

[18] R.W. Numrich and J Reid. Co-array fortran for parallel programming. *SIGPLAN Fortran Forum*, August 1998.

[19] Pedersen et. al. Multidimensional database technology. *IEEE Computer*, 34(12):40–46, 2001.

[20] Reed et. al. Evolution of the mass function of dark matter haloes. *Monthly Notices of the Royal Astronomical Society*, 346:565–572, December 2003.

[21] J. Rogers, R. Simakov, E. Soroush, P. Velikhov, M. Balazinska, D. DeWitt, B. Heath, D. Maier, S. Madden, J. Patel, M. Stonebraker, S. Zdonik, A. Smirnov, K. Knizhnik, and Paul G. Brown. Overview of SciDB: Large scale array storage, processing and analysis. In *Proc. of the SIGMOD Conf.*, 2010.

[22] Sarawagi et. al. Efficient organization of large multidimensional arrays. In *Proc. of the 10th ICDE Conf.*, pages 328–336, 1994.

[23] Seamons et. al. Physical schemas for large multidimensional arrays in scientific computing applications. In *Proc of 7th SSDBM*, pages 218–227, 1994.

[24] E. Soroush, M. Balazinska, and D. Wang. Arraystore: A storage manager for complex parallel array processing. In *Proc. of the SIGMOD Conf.*, 2011.

[25] Springel et. al. Simulations of the formation, evolution and clustering of galaxies and quasars. *NATURE*, 435:629–636, June 2005.

[26] J. G. Stadel. *Cosmological N-body simulations and their analysis*. PhD thesis, University of Washington, 2001.

[27] Stonebraker et. al. Requirements for science data bases and SciDB. In *Fourth CIDR Conf. (perspectives)*, 2009.

[28] Tsuji et. al. An extendible multidimensional array system for MOLAP. In *Proc. of the 21st SAC Symp.*, pages 503–510, 2006.

[29] `http://upc.lbl.gov/`.

[30] Zhang et. al. RIOT: I/O-efficient numerical computing without SQL. In *Proc. of the Fourth CIDR Conf.*, 2009.