A Fine-Grained, Dynamic Load Distribution Model for Parallel Stream Processing

Nathan Backman
Brown University

ABSTRACT
Our goal is to address the unique characteristics and limitations of emerging large-scale commodity clusters to leverage their potential for the parallel processing of multidimensional data streams. To this end, we describe a new distributed stream processing model that integrates data and task parallelism by partitioning workloads into self-describing chunks that are dynamically assigned to available computing resources. We adapt the degree and means of processing parallelism using simulation-driven heuristic search algorithms with underlying incremental bin-packing techniques to direct chunk assignments, facilitating adaptation to fluctuating resource availability and workloads characteristics. Our experimental study quantifies the potential yields of our approach under a variety of workload and computing configurations.

Keywords
distributed stream processing, multidimensional data, chunking

1. MOTIVATION
Dynamic load balancing, in the context of stream processing environments, has been the subject of study for several years. Approaches that have been taken to withstand fluctuations in processing requirements and variances in computing node speeds have included defining static operator-to-node assignments[6] that are meant to be resilient to workload fluctuations[11], transporting entire query operator workloads from one computing node to another [12][1], and interposing load-balancing operators in the query plan to migrate portions of an operator's workload between the nodes that have been assigned to the operation[9][2].

While these strategies have shown to be effective at withstanding fluctuations in workloads and resource availability, we believe that just half of the battle is being fought by only moving statically defined workloads between computing nodes. The other half of the battle concerns how we define the degree of parallelism for a parallelizable operator. It must be determined, for each parallelizable operator, how its data will be subdivided and to which computing nodes its chunks will be sent to. The previously mentioned approaches to dynamic load balancing establish this plan before execution whereas the number of nodes that can contribute to an operator is restricted to a subset of the available nodes and the subdivision of data is defined once.

Such previous attempts at exploiting parallelism in stream processing environments have resulted in the use of isolated and autonomous solutions. Operators within a query plan that can be parallelized are identified and then each becomes the recipient of its own dedicated set of computing nodes[4]. The workload of each parallelized operation is then distributed across its personal set of computing nodes. A downside of this strategy is that computing resources are not shared between these groups. If a parallelized operator faces a surge in increased workload, that operator's set of computing nodes will face the brunt of the impact alone while free processing cycles on nodes associated with other operators will be wasted.

We can illustrate the need for flexible computing nodes (those that aren't restricted to working for a single operator) with an even more simple example that eliminates any fluctuations in data complexity or computing node potential. Suppose we have the query plan depicted in Figure 1 along with two identical computing nodes. Now lets suppose that operator A, in the query plan, is twice as computationally demanding as operator B. To force the assignment of a computing node to only a single operator, in this case, would create a great imbalance in workloads which would have a detrimental effect on latency.

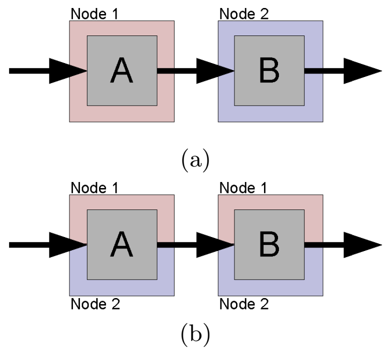

Figure 1: Fixed vs. Flexible computing nodes

If we simplify the example even further and suppose that the operators have an identical computational demand, we face yet another problem by pinning nodes to only a single operator. This time, although workloads are balanced, we sacrifice latency. However, if both nodes equally participated at both operators, the latency for each multidimensional data-item would be cut in half, since we can consistently focus the efforts of both nodes on the same item at the same time.

As the processing requirements of each parallelized operator in a query plan change with time and when processing potential of the computing nodes contributing to them vary, it can be beneficial to balance load not only by moving workloads between nodes but also by redefining an operator's computing node-set and by modifying the granularity of the subdivisions of data. Our work is focused on the use of both of these areas, reformulation of partitioning strategies as well as workload migration, in order to withstand fluctuations in the processing environment.

We focus on the domain of video image processing in which input stream rates are fixed and the frames of the video streams are all of identical size. Streaming video image processing applications differ from traditional streaming applications in that user-defined operators are the norm. These user-defined operators tend to be computationally expensive and their run-times may not necessarily be derived as a function of the number of pixels in each image. For instance, operations may employ feature detection and therefore search the space of an image to find these features. Heuristics can be used to quickly overlook background or non-descriptive regions while drilling down into regions that show characteristics indicative of a feature and worthy of further inspection. Therefore it is possible for the run-time of an operation to be derived as a function of the number of potential features contained within an image.

In the domain of streaming video, we can expect the computational complexity of sequential frames to change over time. Thanks to a general understanding of video streams, however, we can use the assumption that the feature-dependent complexity of images will not change too erratically from frame to frame as we know that visual features in video sequences tend to move gradually rather than erratically. Therefore we can be reasonably sure that the number of features from one frame to the next, in a video sequence, will be similar when video capture devices operate at standard frame-rates. Understanding and keeping track of complexity within images will play an important role in balancing workloads between computing nodes.

We should also note that, recently, it has been impossible to miss the rising trend of the use of large commodity clusters of computers which have been leveraged to get affordable boosts in processing potential. When deploying query workloads to systems such as these, it is important to be aware of the processing capabilities of each node. The computing nodes may be heterogeneous or they may be non-dedicated computers processing varying quantities of external workloads.

With environments such as these, in which data can be homogeneous in its physical nature (in terms of stream rate and data size) but can vary in complexity due to its contents and when we cannot count on the homogeneity of our computing nodes, we must make extra considerations to balance workloads while efficiently using the computing resources.

Determining how to create a good plan achieving parallelism while making efficient use of computing resources requires answering the following questions:

1) **How many chunks do we break the data into?** Even if we presume the simple case of homogeneous computing nodes and homogeneous data complexity, a naive decision may be to deliver a single, equal-sized chunk to each computing node contributing to an operation, but this can cause nodes to wait unnecessarily long for them to receive their chunk before they can begin processing data. At the cost of the additional overhead to create and send extra chunks, nodes can begin processing data sooner than later which can reduce underutilization time of the nodes.

1) **How do we assign an operators chunks to nodes?** We must keep in mind that some computing nodes may have external workloads or even different processor speeds, both resulting in heterogeneous processing potentials. Additionally nodes may have already been assigned workloads from other operators in the query plan who may contend for the node's processing efforts at varying points in time.

2. SYSTEM MODEL

2.1 Data Parallelism Model

The concept of data parallelism, in regard to the chunking of multidimensional data, is slightly different than data parallelism applied to a stream of non-multidimensional data items. When dividing streams of non-multidimensional data items, many previous approaches have relied upon content-based partitioning such as hash-based or range-based partitioning. This content based chunking of workloads is limited by the domain of the data in regard to the degree to which it can be parallelized. When we partition multi-dimensional frames, the content is less important, in terms of defining strict chunk bounds, and instead we focus on the context of the data, which corresponds to a chunks location in multidimensional space. We can find this contextual information in the meta-data of each chunk. In this way, the chunk's data has absolutely no bearing on how the boundaries on that chunk are defined. However, while the contents of the data can be analyzed to help define a chunking strategy, the physical description of each chunk is not directly related to the content of the chunk but is instead related to the meta-data of the chunk. Our creation of regular-shaped chunks can be likened to the content-based partitioning of the ranges of the dimension-space of a chunk.

The model we use to achieve data-parallel processing contains the following three components outlined below and depicted in Figure 2:

2.1.1 Scatter

When a multidimensional array in the query plan arrives at a parallelized operation, the array will be broken down into sub-arrays (chunks) and distributed to the computing nodes that will process them. This happens at a *scatter* point specific to each parallelized operator. A scheduler, which will be discussed in more depth in sections 3 and 4, will inform the scatter point how many chunks to physically partition the image into and will assign each of these chunks to a computing node. These chunks will then be distributed by the scatter point to the computing nodes they have been assigned to.

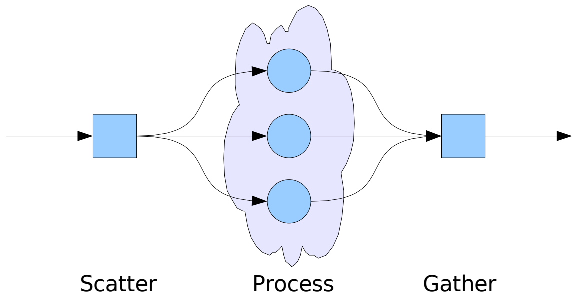

Figure 2: Data Parallelism Model

2.1.2 Processing Self-Describing Chunks

This stage occurs strictly at the computing nodes. The nodes will asynchronously receive data from any of potentially many scatter points (each associated with different parallelized operations). As these chunks arrive, they will be placed in a queue that prioritizes them with respect to their system timestamps to ensure that the data that has been in the query plan the longest is processed first. As a node takes a chunk from its queue, it will determine the parallelized operation it is associated with by conferring with an indicator in the header of the chunk. This will tell the node exactly which operation to perform on the chunk. The node will then begin to process the chunk with respect to the correct operation and then forward the result on to the *gather* point associated with that operation (the location of which can also be extracted from the chunk header).

2.1.3 Gather

Like the scatter point, there is one gather point for each parallelized operation. These will wait to receive processed chunks from the computing nodes. Once it has been established that all chunks of a multidimensional array have arrived, the chunks will be recombined. The array can then be propagated along the query plan in the usual manner.

This can easily be integrated into existing query plans. Issues of fault tolerance can be addressed in ways similar to [3] and [7] for node failures, and [5] for gather or scatter point failures.

Lets consider an existing query plan, depicted in Figure 3a. Suppose that this plan contains two operations that both take advantage of data parallelism. To insert our model to take advantage of data parallelism, we replace each operator we wish to parallelize with its own scatter and gather operators which will interface with a network of computing nodes as seen in Figure 3b.

2.2 Statistics Managing

In order to make educated decisions as to how to pick good partitioning strategies for data=parallel operations, we must have an understanding of the processing speeds of the computing nodes as well as the complexity of the chunks within each multidimensional array. This is done via regular statistics collection from the computing nodes.

As was mentioned earlier, the domain of video image pro-

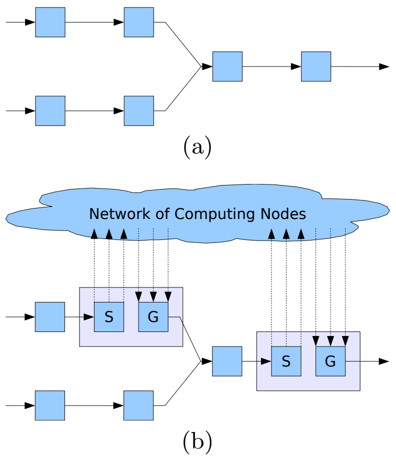

Figure 3: Integration into query plans

cessing is interesting in that we can infer, to some degree, what a frame in a video sequence will look like by observing the previous frame. We use this same concept to estimate the complexity of sequential frames and chunks in a video stream.

With the knowledge of the speed of a node's processor and its current external workload, we can make a judgement about the complexity of a chunk that the node has just processed. By simply using the wall-clock time that it took to process the frame and comparing it to the speed and load of the processor, then we will have an estimate of the number of CPU cycles that it took to process the chunk.

Having collected the per-chunk complexities as well as CPU load measurements of each computing node, we pass this information off to be used to pick partitioning strategies and create corresponding chunk-to-node assignments.

3. BALANCING WORKLOADS

Generally, when we think of balancing workloads, the goal is to make sure that all computing nodes in a set are allocated the same amount of work as the others. When we are dealing with heterogeneous nodes, however, such an allocation does not result in nodes spending an equal amount of time processing which can have negative effects on our goal of minimizing latency. Another way to state our goal is to say that we have x CPU cycles that must be run and we would like to distribute them to the available nodes in a way that minimizes the time it takes to finish all cycles. If we realize that our CPU cycles are lumped into chunks which, given the stream rate, can represent CPU cycles per unit of time, we can begin to visualize this as a bin-packing problem. Our available set of computing nodes are likened to bins and the oddly shaped chunks are what needs to be packed into those bins. What makes this problem difficult to visualize, in the sense of bins and chunks, is that the nodes may be running at differing speeds, and even fluctuating, which calls to question how we perceive chunks fitting into these bins. Since we would like to minimize the latency of processing all chunks, the height of the bins corresponds with time, however, we can imagine the width of the bins changing in real-time to reflect the external workload of the computing nodes.

If we attempt to tackle the entire problem all at once, we will see that there are multiple sets of chunks (one set for each operator) and all available nodes. Standard bin-packing algorithms are meant to balance workloads, not latency, therefore we must look at the case of balancing work belonging to operators in a different light in order to minimize latency. This is because bin-packing algorithms do not encourage data parallelism when workloads for multiple operators are present.

To understand this, lets consider a scenario with four identical computing nodes and four identical parallelized operators that each produce four chunks, all of which are identical in complexity. Figure 4a and 4b both depict "optimal" assignments for this scenario with regard to balancing the workloads (colors denote chunks derived from a particular operator). Notice that Figure 4a does not truly utilize data parallelism, since each operator's workload of chunks has been allocated to a single node, and therefore the latency of processing a single image through any of the operators may be quadrupled in comparison to the allocation seen in Figure 4b.

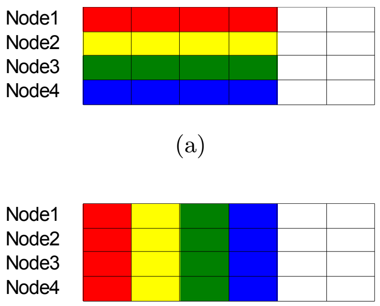

Figure 4: Two allocations of perfectly balanced workloads

However, it is important to realize that, by analyzing these allocations, we cannot determine the true latency of processing all of an operators chunks due to the fact that we do not know when chunks will arrive at the node. In 4b we see four operators contending for resources of all 4 nodes. If we assume that all chunks arrive simultaneously at each node and prioritize chunks so that nodes perform the operations in the same order, the first frame of chunks processed will have a latency four times lower than that of the last frame processed. However, the average latency of processing all chunks would still be half of that latency of the non-data-parallel processing that occurs in 4a. For this reason we must realize that we are not actually looking to minimize the actual latency of processing the chunks of an operation, but instead we are attempting to minimize the lower-bound latency.

There is no different assignment of chunks to nodes that will results in a *more* even distribution of work than either of these examples, but each comes at very different expenses. Therefore it cannot be our priority to only balance the global workload of all chunks across all nodes.

3.1 Chunk-to-Node Assignments

We illustrate how to make latency an optimization goal by highlighting three different ways to apply a traditional incremental bin-packing algorithm.

Global Bin-Packing: The objective of this heuristic is to re-position the chunks of individual operators while considering the workloads of all other operators. For each operator, we follow the incremental bin-packing heuristic used in [9] and [12] which first sorts the set of computing nodes by the amount of time required by each node to process all chunks that have been allocated to that node (regardless of which operator those chunks come from). We then try to move as large of a chunk as possible, belonging to the current operator, from the most burdened node to the least burdened node such that the resulting change lowers the maximum time spent processing of the two nodes. Similarly we attempt to move a chunk from the second most burdened node to the second least burdened node and so on. Thus no more than a single chunk is moved for each pair, per operator, during this bin-packing phase. Since this Global method considers, as its scope, all workloads, this strategy effectively balances the global set of workloads – not end-to-end latency as we find that data parallelism is not encouraged by the global balancing of many workloads.

Local Bin-Packing: This strategy is similar to Global Bin-Packing in that we apply the same heuristic to each individual operator, but it is different in that while sorting the set of computing nodes from most burdened to least burdened, we consider only the workload of the current operation. By changing the scope to only balance the workload of an individual operator we benefit from the fact that this will also optimize for the latency of individual operators. This latency, however is only a lower-bound latency because, since we ignore other workloads, we do not consider the fact that other operators may be contending for the same computing nodes that the current operator is using.

Tiered Bin-Packing: This strategy is actually quite different from the Local and Global strategies because it applies the bin-packing heuristic not necessarily just to individual operators but to non-overlapping subsets of operators in the query plan. We try this strategy because Local does not consider contention for computing nodes between different operators (since it disregards their workloads) while Global tends to do the opposite by considering more that it should. This method creates subsets by walking down the query plan, starting from the outputs, and creating subsets composed of the operators of each successive level of the operator tree. Thus, if the query plan tree has a depth of k, there would be k subsets of operators to apply our heuristic to. The incremental bin-packing heuristics are applied to each of these subsets since those in the set are free to actively contend with each other, whereas any previous subset that have already been balanced in the round, those closer to the output, will not contend over computing nodes. We can say this because when computing nodes asynchronously receive chunks from the scatter points of operators, those chunks will be deposited into that nodes input queue which is implemented as a priority queue. The priority queue gives precedence to chunks that are being processed at operators closest to the outputs as they have been in the query plan the longest and for the sake of minimizing latency we wish to largely focus our efforts on the oldest items in our system. Therefore, this tiered method tries to schedule together those operators that have the same priority and compete with one another for computing resources.

We found, in our simulations of these algorithms, that the cost of applying workload balancing for small query plans was usually less than a quarter of a millisecond, whereas complex query plans with upwards of 20 operators would take on average 1 millisecond.

3.2 Algorithm Comparisons

While all three of the workload balancing algorithms utilize the same incremental bin-packing heuristic, each has a slightly different objective function corresponding to their scope. The effect of those objective functions are evident as we compare the three algorithms.

The Global algorithm is best suited for simple query plans that have little or no operator dependencies. In such scenarios, when there is simply a set of completely independent operators and we would like to minimize the latency of processing all chunks, then Global is an ideal strategy. This configuration of operators prevents computing nodes from congesting pipelines and preventing other nodes from doing work because operators are not dependent on the output of other operators. Therefore the situation will never arise in which computing nodes must wait on another node to produce the last necessary chunk of a frame before that frame can be redistributed to the waiting nodes. A simple and pure workload balancing algorithm, that considers the workloads on all computing nodes, will therefore produce better results than those that only consider workloads of subsets of nodes. This arrangement of operators in a query plan, however, is highly unrealistic since even the most simple real-world plans usually have some degree of operator dependencies. Therefore, as plans become more complex, we see that Global falls behind Local and Tiered since it is not able to adequately promote data-parallel processing, causing computing nodes to congest operators in the query plan.

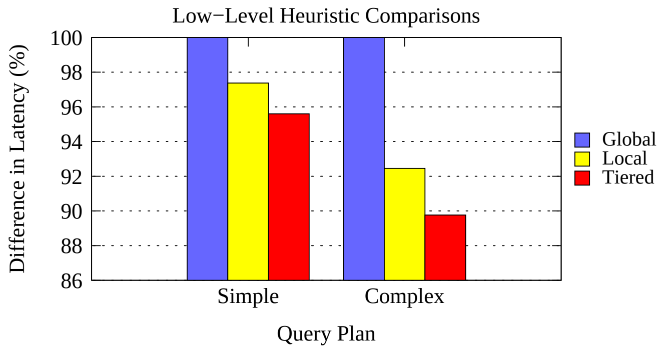

Figure 5: Algorithm Comparisons

In figure 5 we show a comparison of the algorithms from testing two query plans in which the data has been scaled to show latency relative to the worst performer in each test. The 'Simple' plan consists of three parallelized operators with two receiving data from inputs while a third joins the output of the previous two. The 'Complex' plan is a large asymmetrical plan consisting of 20 operators (6 of which are joins, and 7 of which are fed from input streams). Both of these query plans are already complex enough that the Global algorithm falls short compared to Local and Tiered which have an advantage in that they both promote data parallelism across subsets of operators. We also see that Tiered performs better than Local in both tests, with the gap slightly increasing as more operator dependencies are introduced in the Complex query plan. Since the Local algorithm only considers the workload of a single operator at a time, it is unable to capture operator dependencies in the query plan. It is therefore common for the Local strategy to over-utilize nodes by forcing them to heavily contribute to areas of contention instead of distributing a node's computing resources evenly across a query plan. For instance, in the Small query plan it would be more advantageous for a computing node to participate at an operator at an input and the join operator instead of at both inputs. To participate at both inputs can cause those operators to simultaneously contend for the computing node causing it to become a bottleneck. The tiered algorithm is able to mitigate this affect to some degree by grouping together tiers of operators in the operator tree that directly contend with each other for processing time. This ensures that a single computing node will not be overly utilized by any one tier.

4. PICKING PARTITIONING STRATEGIES

As previously stated, for each parallelized operator we must determine the number of chunks to break the data into and then assign those chunks to nodes. To determine the domain of possible numbers of chunks the frame could be broken into, we find all possible ways to subdivide the frame into regular chunks and make a set consisting of the number of chunks in each valid partitioning.

With a domain for set of possible partitioning strategies, we can search over the space of partitions for all operators with the aid of a few search heuristics and a simulator. The simulator provides us with the benefit of being able to replicate the layout of the query plan, processing mechanisms, chunk distribution strategies, workload balancing strategies, and provide us with end-to-end latency measurements all without the cost of actually processing or transmitting data. We sidestep this cost by regularly capturing statistics, as mentioned earlier, that inform us of current node speeds and chunk complexities that we can use to estimate processing times in our simulation environment.

Another benefit that we receive from the simulation is that it can provide us with the chunk-to-node assignments that correspond with the best set of partitioning strategies found by the search heuristic. Due to the nature of trying new configurations with a fresh snapshot of statistics for node speeds and chunk complexities, it is beneficial to run the incremental balancing algorithms multiple times to allow for enough iterations to find good workload balances. If we were to find a better plan from the simulation and pass off just that plan to the actual processing environment, the incremental load balancing heuristics will not have the luxury of having a starting point to increment from and could come to a poor allocation with a potentially very good plan. Since the simulation performs multiple iterations on its own, however, we can pass off the final chunk-to-node allocations associated with the best configuration to allow the balancing heuristics to continue to progress incrementally.

Likewise, we can apply the same strategy to seed the heuristics searching the space of partitioning strategies. The most recently used set of partitioning strategies can seed the heuristics in order to reduce the time it takes to find additional good configurations.

Cyclical Optimization: The first technique we used to search the space of partitioning strategies involves starting with a base set of partitioning strategies, one for each operator, whether randomly picked or seeded with the previously used set of partitioning strategies. We then cycle through operators in the query plan and, for each operator, iterate through all possible partitioning strategies available for that operator and choose the configuration that, when simulated with the partitioning strategies associated with the rest of the operators, provided the lowest latency in simulation. By doing this we find, for each operator the best partitioning strategy available given that the rest of the strategies for other operators are fixed. This process can be looped until no additional benefit can be found, or simply for a predetermined amount of time.

Genetic Algorithm: Our second approach utilized a genetic algorithm to quickly generate and compare populations of candidates [8]. Each candidate, a set of partitioning strategies – one for each operator, is evaluated in the simulator to identify the latency of using those configurations. We then follow the steps outlined below:

1. **Rank all candidates** by their latencies.

2. **Eliminate the poorest performers** in the population, leaving only the "winners". In our tests, we eliminated the poorest performing 75% of the population.

3. **Breed pairs of winners** to increase the population size. To do this we randomly select two candidates and then create a new candidate whose partitioning strategy for each operator has an equal chance of being derived from either parent. This allows us to try to inherit good attributes from candidates who have proven themselves to perform well in the simulation. In our tests we restore 25% of the original population's size by breeding.

4. **Perform mutations** on individual winners to increase the population size. This is done by picking candidates randomly from the winners, cloning their partitioning strategies onto a new candidate and then randomly modifying a percentage of those strategies. This allows us to explore extra options that are "near" existing and well performing candidates in the search space. In our tests we restore 25% of the original population's size with mutations.

5. **Add random candidates** to increase the population size. The logic for this step is to continually search out new and unforeseen candidates. This helps us in maintaining breadth in our search to reduce the chances of getting stuck in local minima in our search for low-latency candidates. In our tests we restore 25% of the original population size with these random additions.

The genetic algorithm is then simply looped, ranking populations, culling, and repopulating to exploit good features of the winners while maintaining a level of randomness. The algorithm can continue until a stopping criteria has been reached, which can be defined in one of many ways. For instance, the algorithm can simply stop iterating after a time limit has been reached and return the best performing candidate, the algorithm can proceed for a set number of iterations, the algorithm can be stopped if the latency of the best performing candidate has been the same for the last x iterations, the algorithm can be stopped once the difference in latency between the best candidates in subsequent iterations is less than a desired threshold, or a combination of these methods or other methods can be used.

4.1 Algorithm Comparison

To compare the search heuristics, we used the same Complex query plan used in the previous tests to provide us with a search space. We then fed that plan to both algorithms and allowed them each to run for one minute in order to see the quality of the partitioning sets each would produce over time. We re-ran the tests multiple times to find the average quality of plans produced over time for both algorithms.

We can also improve on this quick decent into low latencies by seeding the input, as mentioned previously. We conducted another test in which we ran both algorithm seeded with a set of previous partitioning strategies. The seed was derived from a processing environment that had computing nodes that were 30% faster than the current environment to illustrate rerunning the parallelism picker after a sudden degradation in processing potential whereas each node degraded to a different degree. The results of both algorithms, over time, with and without seeds can be seen in Figure 6. Even starting with a configuration for parallelism derived from a processing environment with a 30% difference in computing potential allows for an improved entry into regions of the search space that provide good plans for parallelism configurations.

We notice that both algorithms are able to quickly find better configurations and that seeds help improve the decent into better configurations. We see, in this case, that the cyclical algorithm outperforms the genetic algorithm. The cyclical algorithm, we have found, is good at optimizing portions of the query plan (for instance, chains of dependant operations), whereas improving a single operator in the chain will improve the entire chain. However, the cyclical algorithm is only able to improve on one operator at a time and is therefore unable to modify sub-trees of the query plan at once which has difficulties in optimizing two bottleneck inputs to a join in a timely manner. While the genetic algorithm, in this instance, is beaten out by the seeded cyclical algorithm it is able to consistently provide good plans for

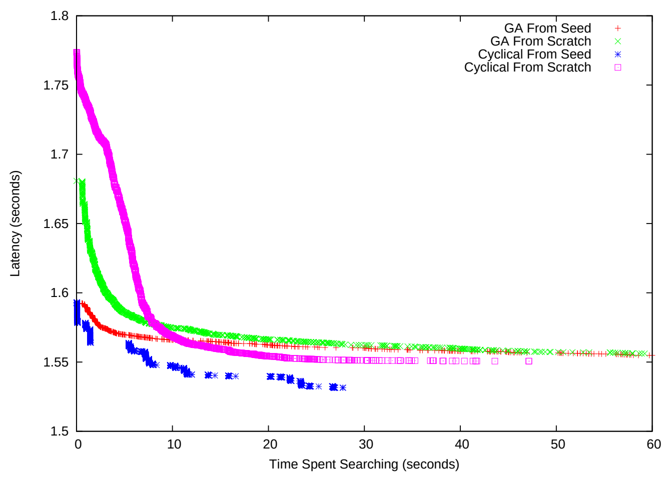

Figure 6: Quality of Search Heuristic Results Over Time

parallelism configurations regardless of the organization of the query plan.

Additionally, genetic algorithms can be run with varying population sizes and, with additional tests, we saw somewhat predictable results. Given larger population sizes, a genetic algorithm is likely to find better results since it is able to cover a large portion of the search space and keep some degree of breadth in order to avoid falling victim to local minima. However, due to the size of these large populations, iterations take longer than they would with a smaller population and therefore we are slower to come to those better configurations. With a small population size of 16, we are able to iterate very fast and can quickly take advantage of the genetic algorithm's ability to capitalize on exploiting good performing traits found in the top-performing candidates of the population which can lead to a steeper initial decent to low latencies.

One downside to using heuristics to search this very large space is that it is difficult to know exactly how close to optimal the results are. For this reason, we contrived a query plan for which it was easy to identify the optimal set of partitioning strategies in order to see how closely the genetic algorithm could approximate the optimal configuration. The query plan simply consisted of a singe long chain of 10 identical operators. The solution for the entire query plan, therefore was to replicate the optimal partitioning strategy for a single and isolated instance of any of the operators and to apply that partitioning strategy to all of the operators. While this plan is conceptually extremely simple, the genetic algorithm is completely agnostic to the structure of a query plan or the similarity of operators. We fed this plan to the genetic algorithm, starting with randomized inputs, and averaged the results found over time from many tests. Those results are depicted in Figure 7. We can see that, even without the advantage of having a useful seed, the genetic algorithm was able to come very close to matching the optimal configuration. Even though, out of all of the tests, none of the attempts by the genetic algorithm perfectly matched the latency of the optimal configuration, the average of best latencies of all tests was within 2.5 milliseconds of the optimal.

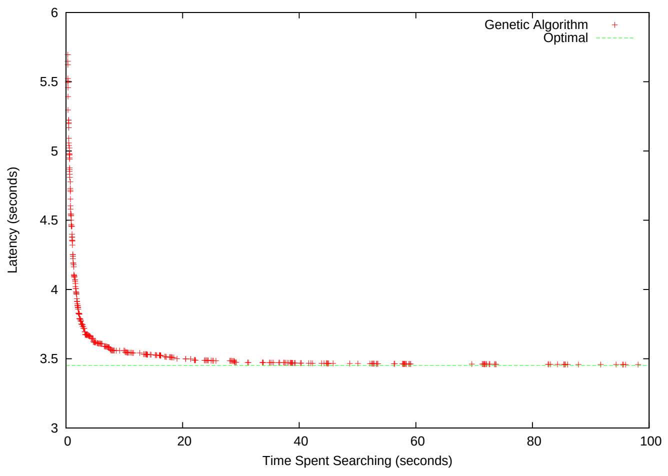

Figure 7: Genetic Algorithm vs. Optimal Configuration

5. SIMULATION AND ANALYSIS

We developed a discrete event simulator to evaluate the joint efforts of our partitioning strategy picker and workload balancer. This allows us to simulate the processing of diverse query plans on large clusters of non-dedicated heterogeneous computing nodes whose different processing potentials (defined by cycles per second) are all capable of fluctuating in order to express the effects of external workloads on the computing nodes. The multidimensional data that are propagated through the query plans can also be defined in terms of their complexity (CPU cycles required) in relation to each operator in the query plan. This gives us the ability to change chunk complexities between sequential frames in the data stream in order to represent complexity migration throughout a multidimensional frame or to simply increases and decreases operator workloads. Network transmission of data between computing nodes is also included in our simulation with all nodes presumed to be connected to a 100Mb switch. The processing performed by the operator is simulated as a wait-time which are derived from the complexity of the chunk and the processing potential of the computing node assigned to that chunk.

5.1 Adapting to Dynamic Environments

To demonstrate how the partitioning strategy picker and the workload balancer work together to suggest new configurations of resource utilization, we conducted a series of experiments that involved manipulating the processing environment to cause previous configurations to perform poorly, thus enforcing the need for adaptation. To do this, we used the Simple query plan, mentioned previously, and modified the cost of processing the multidimensional data at each of the operations. This update in chunk complexity statistics was then passed to the search heuristic which would, in turn, update its internal simulator so that its own instance of the workload balancer would have a more up-to-date view of the processing environment. As the search heuristic then investigate the search space of all partitioning strategies, it will discover that the current configuration (which the algorithm was seeded with) may be less optimal than some of the new configurations it may discover (potentially with the help of the seed).

While modifications were made to the cost of producing output for each operator, the cost of the entire query plan was maintained. To do this, we mimicked a migration of cost throughout the query plan. Initially all operators were treated identical in processing complexity. We then migrated a large portion of the costs of one of the input operators to the other input operators. This would result in a large bottleneck if the workload distribution had stayed fixed with the configuration developed for operators with homogeneous costs. The cost of the join operator actually stayed the same, however, complexity was migrated within its multidimensional data. Instead of complexity being evenly distributed through the array, as was the previous case, we moved a large amount of the complexity to one half of the array. This causes half of the output chunks to be much more costly to produce than the other half. In this way, the total processing demand for the operator has stayed the same, but the chunks have now been unevenly altered in complexity. In addition to this migration of complexity within the query plan, we made available 6 computing nodes, three of which were twice as fast as the other three.

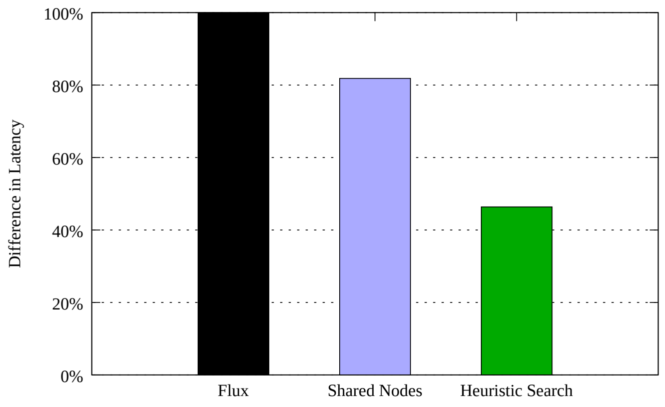

Figure 8: Genetic Algorithm vs. Optimal Configuration

In this experiment we compared 3 different load distribution strategies as seen in Figure 8. The first is an imitation of Flux[9] and is based upon the initial assumption of homogeneous operators with heterogeneous computing nodes. Each operator was a distinct set of computing nodes there dedicated to that operator, with a Local Bin-Packing strategy applied. This allows workload to be migrated between the computing nodes in an operator's private node set, but does not allow the sharing of resources. Therefore, we see in the test that this strategy results in the computing nodes which have been assigned to the less burdened input operator become very much under-utilized, while the other input operator's node are over-utilized. This strategy is able to well balance the migration in complexity associated with the join operator. However, we see that since the operator node-sets are distinct, that latency has the possibility of suffering due to the fact that the efforts of computing nodes are not able to be combined to concurrently progress the oldest data in the system. For instance, if the join operator has nothing to process while the input operators due, the computing nodes at the join operator are not being effectively used to minimize latency as they would sit idle.

The second strategy, labeled "Shared Nodes", allowed for operators to now have non-distinct node-sets, however they were still fixed node-setts without the ability to change nodes over time. Each of the slower nodes was now shared by two operators so that each operator had, in its node-set, one fast and two slow nodes. Here we employed the Tiered bin-packing strategy which was able to alleviate many of the underutilization problems we saw in the previous strategy while minimizing contention between the concurrently running input operations.

The third strategy includes our heuristic search to find good partitioning strategies for each of the operators. Now, we allow the number of chunks each operator produces to be modified since we have noticed the complexity migrations between operators in the query plan and within the join operator. This gives the bin-packing algorithms finer grained control as to how well they are able to pack chunks onto heterogeneous nodes. For instance, it could be beneficial to further partition the more complex input operator in order to provide a more fine-grained distribution of the workload across the computing nodes so that it fits in well with other concurrently running workloads.

5.2 When to Reevaluate

There is obviously a cost to running the search heuristic invoking multiple simulations of partitioning configurations and the corresponding workload balancing. The question of how often to run the search heuristic is primarily affected by how dynamic the processing environment. If a processing environment is completely static, such that frame complexities and computing node processing potentials may be heterogeneous but never change, then we can simply run the search heuristic once and stick with that configuration indefinitely. If a processing environment is mildly dynamic we may only have to call the search heuristic once every couple of minutes or hours. Likewise, if the environment is extremely dynamic it would be required to run the search heuristic more frequently to adapt to changes. It is not straightforward to simply suggest a time line for which a streaming application should rerun the search heuristic, since time, for each application comes with different amounts of environmental fluctuations. Instead we suggest running the search heuristic in the background when idle cycles are available. The nature of these search heuristic is that they are able to work with whatever time they are given, whether small or large. Additionally heuristic and corresponding simulation are easy to increment and update to reflect the changing environment. Search heuristics can be seeded with recent partitioning configurations to reflect the recent-history, while the statistics in the simulator can be updated to better reflect the external workloads on individual computing nodes as well as newly observed chunk complexities.

6. OPERATOR PARALLELISM

Not all operators can benefit from, or even take advantage of, data-parallel processing. Many operator are inherently not parallel, however, they still have a place within our system. To represent such an operator, the partitioning strategy going into the workload balancing heuristics would simply denote that the operator had 1 chunk to deliver. The workload balancing heuristic would still be able to move that single chunk from node-to-node if it could improve the latency of that operator to do so.

Sometimes achieving data parallelism for certain classes or operators is not as clean-cut as simply breaking an array into distinct pieces and sending them on their way. While there are many classes of data-parallel operations, most that process multidimensional data seem to fall largely into these three categories [10]:

- **Cell-to-Cell Mappings:** This category is the most straightforward and simple whereas each cell in the domain array is transformed and mapped directly onto an individual cell on the array in the co-domain. In this case, frames simply can be subdivided into distinct chunks without the need for any inter-chunk processing.
- **Region-to-Cell Mappings:** This category relates to much of the image processing domain including any operation that has a sliding window that iterates over an image to apply a transformation. We now, clearly, have the case in which such a region can overlap multiple chunks thus giving rise to the inter-chunk processing problem. There could be multiple locations in co-domains associated with differing chunks that rely on similar regions in the domain. In order to perform these operations in a data-parallel manner, it is necessary to transmit extra information with each chunk. The amount of extra information would generally be defined by the size of the region or sliding window that pans across the original array in order to produce border-cells of chunks in the co-domain. This buffer would allow such sliding windows to span across, what were previously chunk boundaries.

  We now face a trade-off. It must be established if the cost of transmitting extra data is worth the price of data-parallel processing. If it is believed that data-parallel processing is still advantageous, then the extra information can be packaged into each chunk with definitions in the chunk header as to coordinate boundaries of the co-domain chunk so that the computing nodes will produce the appropriate chunks as output. Not only is the question raised as to whether or not chunking should be done in the first place, but also as to what granularity of chunking should be performed. With additional chunks comes the potential for increased degrees of data-parallel processing while potentially reducing underutilization of computing nodes, but each additional chunks comes with the increased transmission cost of sending additional data.
- **Global-to-Cell Mappings:** This can be considered an extreme case of region-to-cell mappings in which the "region" is defined as the entire input array for each cell in the co-domain. In such circumstances, each chunk would consist of the entire array with the specific coordinate boundaries of the co-domain's output chunk in the header. Obviously, there must now be a substantial benefit to data-parallel processing in order for the trade-off of broadcasting an entire array to all computing nodes in the operator's node-set to be worthwhile.

In our work we assume that operators have already been classified into the previous categories and that the costs of redundant data transmission and computations is captured in our chunk sizes and complexities.

7. FUTURE WORK

One of the next logical steps is to implement the system we have defined. This would require measures to establish a network communication protocol which allows all computing nodes, whether on separate physical computers or multiple cores on a single CPU, to communicate with the scheduler in order to deliver statistics and to receive updates to chunk routing tables. It would also be necessary to implement a scheme for chunk headers to carry information relative to processing the data in the chunk (which should include information on how to reconstruct the array data), an indicator as to which operation to perform on the data, a definition of the output chunk to produce (should the domain of the operation differ from the co-domain), and an address used to send a chunk to its operation's gather point for aggregation.

Another direction we can take with this research is to allow for more fine grained control of the amount of complexity contained within each chunk by allowing for irregularly shaped chunks. Even if such a non-trivial problem is solved a variety of new questions would be available to explore. For instance, how do we process irregular and non-rectangular chunks without requiring a large degree of extra and redundant information for operations involving region-to-cell mappings?

With the proliferation of multi-core processors there are many additional optimizations we can make for these architectures. Each core would not necessarily have to act as an individual and isolated computing node, but simply as a collection of processors all working for the same computing node. In this way, all chunks that would previously have been destined for these cores can just be deposited at a queue in the main memory of the computer housing these cores. The cores can then simply pull chunks from that queue to process (whether they would have previously been destined for that core or another core on the same computer). This strategy would reduce time that cores spend waiting while their queues would previously have been empty and those of other cores potentially not empty. As long as a chunk is resident on memory accessible by all cores, it does not matter which core actually does the processing work. An extra feature to explore would be in trying to benefit from the locality of cores with respect to where chunks are located. For instance, if a computer houses a scatter point for an operator and cores on the computer are currently available to process data, it could be beneficial to not transfer those chunks over a network to cores on other computers. It would be interesting to find out when it would be advantageous to keep data locally and when it is not. Yet another benefit from memory locality in multi-core computers is that redundant data need not be sent to these cores. For operations involving global-to-cell mappings (or even region-to-cell), an entire array could be sent once allowing all cores to read the same data while processing chunks of differing co-domains.

Ideally, a user of the system should be able to create a query plan and just let it run without having to previously specify by hand the classifications of data-parallel operators in the query plan and valid partitioning strategies. It should be transparent to the user how and what kind of internal chunking we perform on data for each operator and how much extra and redundant information must accompany chunks. It would be interesting to explore strategies of black-box analysis to discover access patterns on the input frame and observe how data is written to the output frame in order to infer how data-parallel each operator is automatically.

8. REFERENCES

[1] D. J. Abadi, Y. Ahmad, M. Balazinska, U. Çetintemel, M. Cherniack, J.-H. Hwang, W. Lindner, A. Maskey, A. Rasin, E. Ryvkina, N. Tatbul, Y. Xing, and S. B. Zdonik. The design of the borealis stream processing engine. In *CIDR*, pages 277–289, 2005.

[2] R. H. Arpaci-Dusseau, E. Anderson, N. Treuhaft, D. E. Culler, J. M. Hellerstein, D. Patterson, and K. Yelick. Cluster I/O with River: Making the fast case common. In *Proceedings of the Sixth Workshop on Input/Output in Parallel and Distributed Systems*, pages 10–22, Atlanta, GA, 1999. ACM Press.

[3] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. In *OSDI*, pages 137–150, 2004.

[4] A. Gounaris, R. Sakellariou, N. Paton, and A. Fernandes. Resource scheduling for parallel query processing on grids, 2004.

[5] J.-H. Hwang, M. Balazinska, A. Rasin, U. Cetintemel, M. Stonebraker, and S. Zdonik. High-Availability Algorithms for Distributed Stream Processing. In *The 21st International Conference on Data Engineering (ICDE 2005)*, Tokyo, Japan, April 2005.

[6] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: Distributed data-parallel programs from sequential building blocks. In *European Conference on Computer Systems (EuroSys)*, pages 59–72, Lisbon, Portugal, March 21-23 2007. also as MSR-TR-2006-140.

[7] V. Raman, W. Han, and I. Narang. Parallel querying with non-dedicated computers. In *VLDB '05: Proceedings of the 31st international conference on Very large data bases*, pages 61–72. VLDB Endowment, 2005.

[8] S. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach*, pages 116–119. Prentice-Hall, Englewood Cliffs, NJ, 2nd edition, 2003.

[9] M. Shah, J. Hellerstein, S. Chandrasekaran, and M. Franklin. Flux: An adaptive partitioning operator for continuous query systems. Technical Report UCB/CSD-2-1205, U.C. Berkeley, 2002.

[10] C. Soviany. *Embedding Data and Task Parallelism in Image Processing Applications*. PhD thesis, Technische Univ. Delft, 2003.

[11] Y. Xing, J.-H. Hwang, U. Çetintemel, and S. Zdonik. Providing resiliency to load variations in distributed stream processing. In *VLDB '06: Proceedings of the 32nd international conference on Very Large Data Bases*, pages 775–786. VLDB Endowment, 2006.

[12] Y. Xing, S. Zdonik, and J.-H. Hwang. Dynamic load distribution in the borealis stream processor. In *ICDE '05: Proceedings of the 21st International Conference on Data Engineering*, pages 791–802. IEEE Computer Society, 2005.