Translating Chapel to Use FREERIDE: A Case Study in Using an HPC Language for Data-Intensive Computing

Bin Ren Gagan Agrawal Brad Chamberlain[†] Steve Deitz[†]
Department of Computer Science and Engineering
The Ohio State University
Columbus, OH 43210
{ren, agrawal}@cse.ohio-state.edu
Cray Inc.
901, Fifth Avenue, Suite 1000
Seattle, WA 98164
bradc@cray.com, stdeitz@microsoft.com

***Abstract*—In the last few years, the growing significance of data-intensive computing has been closely tied to the emergence and popularity of new programming paradigms for this class of applications, including Map-Reduce, and new high-level languages for data-intensive computing. The ultimate goal of these efforts in data-intensive computing has been to achieve parallelism with as little effort as possible, while supporting high efficiency and scalability. While these are also the goals that the parallel language/compiler community has tried meeting for the past several decades, the development of languages and programming systems for data-intensive computing has largely been in isolation to the developments in general parallel programming.**

Such independent developments in the two areas, i.e., data-intensive computing and high productivity languages lead to the following questions: I) Are HPC languages suitable for expressing data-intensive computations? and if so, II.a) What are the issues in using them for effective parallel programming? or, if not, II.b) What characteristics of data-intensive computations force the need for separate language support?.

This paper takes a case study to address these questions. Particularly, we study the suitability of Chapel for expressing data-intensive computations. We also examine compilation techniques required for directly invoking a data-intensive middleware from Chapel's compilation system. The data-intensive middleware we use in this effort is FREERIDE that has been developed at Ohio State. We show how certain transformations can enable efficient invocation of the FREERIDE functions from the Chapel compiler. Our experiments show that after certain optimizations, the performance of the version of Chapel compiler that invokes FREERIDE functions is quite comparable to the performance of hand-written data-intensive applications.

I. Introduction

The availability of large datasets and the increasing importance of data analysis in commercial and scientific domains is creating a new class of high-end applications. Recently, the term *Data-Intensive SuperComputing* (DISC) has been gaining popularity [2], reflecting a growing class of applications that perform large-scale computations over massive datasets. The deluge of available data for analysis demands the need to scale the performance of data mining implementations.

The growing significance of data-intensive computing has been closely tied to the emergence and popularity of the Map-Reduce paradigm [5]. Map-reduce is a high-level API, with associated runtime libraries, to help develop parallel data-intensive applications. Over the last few years, multiple projects have focused on improving the API or implementations [7], [12], [20], [21], [23], [24], [26]. At the same time, there are studies focusing on the suitability of the Map-Reduce model for a variety of applications across different platforms [8], [22], [11].

One particular development of interest to the parallel languages/compiler community has been towards new high-level languages for data-intensive computing. Specifically, Google has developed Sawzall [21] and Yahoo has developed Pig Latin [20]. They can both be viewed as higher level interfaces, from which runtime systems like Map-Reduce can be directly invoked. The ultimate goal of these efforts in data-intensive computing has been to achieve parallelism with as little effort as possible, while supporting high efficiency and scalability. While these are also the goals that the parallel language/compiler community has tried meeting for the past several decades, the development of languages and programming systems for data-intensive computing has largely been in isolation to the developments in general parallel programming. In recent years, the main focus of the parallel language community has been on high productivity languages. To this end, efforts like Chapel [6] and X10 [4] are developing highly expressive languages coupled with advanced tools support, to enable high productivity for parallel programming. As is common in the high performance community, these efforts have targeted scientific compute-intensive applications.

Such independent developments in the two areas, i.e., data-intensive computing and high productivity languages lead to the following questions: I) Are HPC languages suitable for expressing data-intensive computations?, and if so, II.a) What are the issues in using them for effective parallel programming? or, if not, II.b) What characteristics of data-intensive computations force the need for separate language support?.

This paper takes a case study to address these questions. Particularly, our work is in the context of the Chapel programming language. We study the suitability of Chapel for expressing data-intensive computations. We also examine compilation techniques required for directly invoking a data-intensive middleware from the Chapel compilation system. The data-intensive middleware we use in this effort is FREERIDE [17], [15]. FREERIDE (FRamework for Rapid Implementation of

Datamining Engines) has been developed at Ohio State. It shared many similarities with Google's Map-Reduce, but has been shown to clearly outperform the Hadoop implementation of Map-Reduce for data mining applications [14].

Chapel has advanced support for reductions, and reduction computations arise often in data-intensive applications [16]. Thus, we find that Chapel is well suited for expressing many sub-classes of data-intensive applications. At the same time, it can achieve better performance by invoking a specialized runtime library for data-intensive computations. This, however, leads to several challenges, since data-intensive computing runtime systems have not been written to support data-structures in advanced languages like Chapel. Thus, substantial compiler transformations are required to be able to invoke a system like FREERIDE with Chapel.

This paper reports on these transformations and their implementation in the Chapel compiler. We have evaluated our implementation using two popular data mining algorithms. The main observations from experiments are as follows. By invoking FREERIDE from Chapel automatically, the system is able to effectively scale these data-intensive applications on a multicore machine. Moreover the performance of the compiler generated code, after several optimizations, is quite comparable to the hand-written code using the same runtime library.

Note that the reduction computations we support have been quite extensively studied in the context of various shared memory parallelization projects [1], [3], [18], [19], [25]. Our work is distinct in considering issues associated with a specific high productivity language and a data-intensive runtime system.

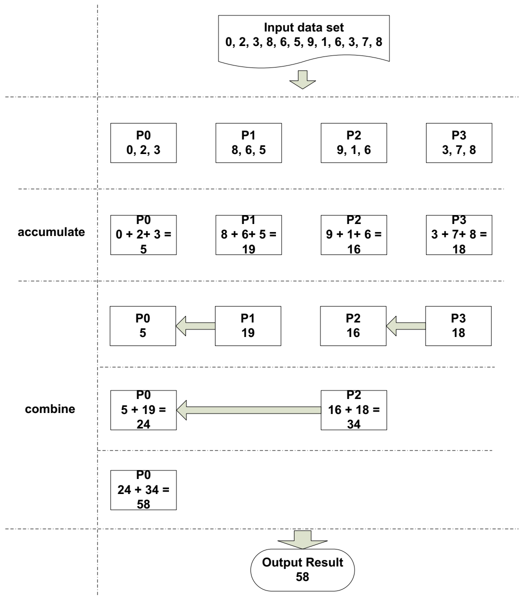

Fig. 1. Sum Reduction

```
/*The sum reduction class*/
class SumReduceScanOp: ReduceScanOp {
      type eltType; //the data type parameter
      var value: _sum_type(eltType).type;

      /*The local reduction function*/
      def accumulate(x){
            value = value + x;
      }

      /*The global reduction function*/
      def combine(x){
            value = value + x.value;
      }

      /*The function output the final result*/
      def generate(){
            return value;
      }
}
```

Fig. 2. Sum Reduction in Chapel

II. Chapel Language and Reduction Support

Chapel is a high-level parallel programming language developed by Cray Inc, which is mainly designed to improve the productivity and portability of parallel programs. This section focuses on the design of the *Reduction mechanism* in Chapel.

A. Global-view Abstraction of User-defined Reduction in Chapel

In order to implement a reduction operation, there are two kinds of models in Chapel: *local-view abstractions* and *global-view abstraction*. Both of them can be used to support user-defined reduction operations.

In the *local-view abstraction*, the programmer needs to manage data distribution as well as communication between different processors explicitly. It is a lower-level reduction model, with the obvious tradeoff that it is very straight-forward for a compiler to implement. Chapel also supports a *global-view abstraction* model, which is a higher-level model and hides the data distribution and communication details.

B. Basic Chapel Reduction Structure

Figure 1 illustrates a typical reduction operation: `sum`. We show how this operation is separated into two stages: the local accumulation and the global combination, for execution in a parallel environment. Figure 2 specifies the Chapel implementation of this operation.

In Chapel, both the built-in reduction operations and the *user-defined* operations are designed as a sub-class, which is inherited from the class `ReduceScanOp`. This characteristic reflects the object-oriented feature of the Chapel language. A reduction operation in Chapel can also take different data types as parameters. For example, in this `sum` operation, the programmer can pass integer, float, as well as other numbers to do the calculation. This feature is similar to the *templates* feature in C++.

Figure 2 shows that a typical Chapel Reduction can be divided into three main stages, where the first two stages correspond to the two steps shown in Figure 1:

Accumulate: This is the local reduction function, which is performed by each thread on each processor. The input of this function is typically one element of the input dataset. More specifically, the entire process can be described as follows: 1) the input dataset for the entire Chapel reduction is split among the threads, with each thread holding its own *split*, 2) the *accumulate* function is applied on each data split, where the elements in the split are processed one by one. For instance, if the input data is an array, say `a` with n elements, and let there be p threads. Then, the input array will be divided into p splits: a_1, a_2, …, a_p. One invocation of the *accumulate* function is operated on one split, say, a_i. The accumulate function adds each element in a_i to the local sum result, denoted as `value`. By such local reduction, each thread has a local result.

Combine: This is a global reduction function, where the local results on each thread can be combined together to form the final result of the entire reduction. The input for this function is an object created from the local reduction. The *combine* operation is handled by the compiler automatically, and the user only needs to provide the combine logic between two local results. For instance, in the `sum` reduction, the user just needs to provide a simple logic to add the current local result `value` with the local result `value` from any other thread.

Generate: This is a general post-processing step, which can be used to return the final result to the `callee`.

```
class kmeansReduction: ReductionScanOp{
    type eltType;
    var RO: [1..k] redObj;
    def accumulate (da: eltType){
        min_distance = max(int);
        min_disposition = 1;
        for each centroid_i {
            find the nearest centroid,
            mark its index as min_disposition;
            mark the distance as min_distance;
        }
        update RO[min_disposition] by min_distance;
    }
    def combine(km: kmeansReduction(eltType)){
        for i in 1 to k {
            merge RO[i] with km.RO[i];
        }
    }
    def generate(){ return RO; }
}
```

Fig. 3. Pseudo-code for k-means using Chapel Reduction

In our previous work, we have shown how a number of data mining algorithms [16] and scientific data processing algorithms [10], [9] follow a *generalized reduction* structure. Chapel, with its strong support for specifying reductions, appears like a promising language for specifying these applications at a high-level.

As an example, we consider a popular data mining algorithm, k-means clustering [13]. The main steps for k-means can be described as follows: 1) select k points as the initial centroids randomly, 2) scan the entire dataset to form k clusters by assigning each point to its closest centroid; 3) update the centroid of each cluster according to their current points; 4) repeat steps 2 and 3 until the centroids are stable.

Figure 3 shows the code for one iteration of the k-means algorithm using the Chapel reduction. During the *accumulate* phase, each point is assigned to the closest centroid and the explicit *reduction object* is updated. During the *combine* phase, the copies of the reduction object are merged together and the final result is output by the *generate* phase.

III. FREERIDE Middleware

This section describes the FREERIDE middleware, which has been developed at Ohio State over the last 9 years. We focus on the API for data-intensive computing offered by this system. The experiments presented in this paper were conducted using a recent implementation of the FREERIDE API [15]. This implementation is specifically for multi-core machines, and is based on the Phoenix system [22].

A. Generalized Reduction Structure

FREERIDE shares many similarities with the Map-Reduce framework. However, there are some subtle but important differences in the API offered by these two systems. First, FREERIDE allows developers to explicitly declare a reduction object and perform updates to its elements directly, while in Hadoop/Map-Reduce, the reduction object is implicit and not exposed to the application programmer. Another important distinction is that, in Hadoop/Map-Reduce, all data elements are processed in the map step and the intermediate results are then combined in the reduce step, whereas in FREERIDE, both map and reduce steps are combined into a single step where each data element is processed and reduced before the next data element is processed. This choice of design avoids the overhead due to sorting, grouping, and shuffling, which can be significant costs in a Map-Reduce implementation. Furthermore, this also alleviates the need for storage of intermediate $(key, value)$ pairs, which can require a large amount of memory, and slow down several data mining implementations [14].

The following functions must be written by an application developer as part of the API:

Local Reductions: The data instances owned by a processor and belonging to the subset specified are read. A local reduction function specifies how, after processing one data instance, a *reduction object* (declared by the programmer), is updated. The result of this process must be independent of the order in which data instances are processed on each processor. The order in which data instances are read from the disks is determined by the runtime system.

Global Reductions: The reduction objects on all processors are combined using a global reduction function.

Throughout the execution of the application, the reduction object is maintained in the main memory. After every iteration of processing all data instances, the results from multiple threads in a single node are combined locally depending on the shared memory technique chosen by the application developer. After local combination, the results produced by all nodes in a cluster are combined again to form the final result, which is the global combination phase. The global combination phase can be achieved by a simple all-to-one reduce algorithm. If the size of the reduction object is large, both local and global combination phases perform a parallel merge to speed up the process. The local combination and the communication involved in the global combination phase are handled internally by the middleware and is transparent to the application programmer.

Fig. 4 further illustrates the distinction in the processing structure enabled by FREERIDE and Map-Reduce. The function *Reduce* is an associative and commutative function. Thus, the iterations of the for-each loop can be performed in any order. The data-structure *RObj* is referred to as the reduction object.

B. API Details

The functions that are important for our code generation task are summarized in Table I.

In order to make application development easier, this version of FREERIDE provides a default splitter and combination function. In our work, these default splitter and combination functions are used. Corresponding to the Figure 3, Figure 5 shows the one iteration of processing for the k-means clustering algorithm using the API listed above.

```
FREERIDE
{* Outer Sequential Loop *}
While() {
    {* Reduction Loop *}
    Foreach(element e) {
        (i, val)    =   Process(e) ;
        RObj(i)   =   Reduce(RObj(i),val) ;
    }
    Global Reduction to Combine RObj
}
```

```
Map-Reduce
{* Outer Sequential Loop *}
While() {
    {* Reduction Loop *}
    Foreach(element e) {
        (i, val)    =   Process(e) ;
    }
    Sort (i,val) pairs using i
    Reduce to compute each RObj(i)
}
```

Fig. 4. Processing Structure: FREERIDE (left) and Map-Reduce (right)

TABLE I
DESCRIPTIONS OF THE FREERIDE APIs

Functions Defined by Users
*void (*reduction_t)(reduction_args_t*)* The main part of FREERIDE reduction designed for users to provide the reduction logic to each data split and update the implicit reduction object by other APIs
*void (*combination_t)(void*)* Combine the copies of the *reduction object*, during the *duplicated* model
*(*finalize_t)(void*)* Execute the finalizing task and output special information at the end of the reduction operation
Functions Provided by the Middleware
*int (*splitter_t)(void*, int, reduction_args_t*)* Split the whole input data set according to the number of the threads provided by the initialization part
int reduction_object_alloc() Initialize the reduction object and assign a unique ID for each element of the reduction object as the index
void accumulate(int, int, void value)* Update the reduction object
void get_intermediate_result(int, int, int)* Get the intermediate result from the reduction object

```
{*Initialization of FREERIDE including initialization
of the reduction dataset and the reduction object*}

{*Call reduction functions by function pointers*}

{*Main function pointers*}
void splitter(void* data_in, int req_units, reduction_args_t* out){
    {*Using default splitter*}
}
void reduction(reduction_args_t* args){
    for each points in args→data {
        for each centroid in args→centroids {
            find the nearest centroid,
            mark its index as min_disposition;
            mark the distance as min_distance;
        }
        update reduction object by accumulate function;
    }
}
void combine(){
    {*Using default combine function*}
}
```

Fig. 5. Pseudo-code for k-means using FREERIDE Reduction

IV. Translation Issues and Implementation

This section describes the main technical issues we addressed in automatically generating FREERIDE calls from Chapel. Initially, we compare the reduction support in two systems and list the key steps in conversion.

A. Comparison and Main Translation Issues

Consider a reduction based computation like the one shown in Figure 3. To use FREERIDE support for processing, we need to invoke functions for the following tasks.

- Invoke the *split* function in FREERIDE to divide input data into chunks or units that can be processed by each thread.
- Call reduction function to update the reduction object.
- Call combine (and finalize) functions.

Based on the comparison of Figure 3 and Figure 5, the main issues in translation from Chapel Reduction to FREERIDE are as follows. FREERIDE is based on a simple 2-D array view of the input dataset. Such a simple view allows the runtime system to partition the data for processing between threads (and nodes), and simplifies the APIs. In comparison, Chapel allows nested data structures and even allows reductions to be used on these. Thus, to use FREERIDE reductions, data must be translated to *simpler* data structures that the runtime libraries support. This process is called *linearization* and is described in the next subsection.

The second main step involves translation of the reduction itself. Here, a key issue is to map the operations on the original data structures to the linearized data structures our linearization step creates.

To formally state the compilation problem, if the D_v is the data view used for computation and D_s is the low level data storage (a dense memory buffer), our goal is to compute 1) $F_t \subseteq \{f \mid f : D_v \rightarrow D_s\}$, a linearization function, which can transform the high level data view to low level data storage in the data set definition code section, and 2) $M \subseteq \{m \mid m : D_v \rightarrow D_s\}$, a mapping function established to support applying the high level operation logic to low level data storage.

In following subsections, we will introduce the functions F_t and M in detail.

B. Linearization

We now describe the linearization algorithm that is used in our translation process. We believe that this issue is not specific to Chapel and FREERIDE, but can arise in mapping of high-level or high productivity languages to efficient runtime systems. On one hand, support for nested structures allows more elegant and reusable code. On the other hand, low-level

Algorithm 1 computeLinearizeSize(Xs)

```
1: $size$ = 0
2: if $Xs$.type = $isPrimitive$ then
3:    $size$ = sizeof($Xs$)
4: else if $Xs$.type = $isIterative$ then
5:    for $x$ in $Xs$ do
6:       $size$ += computeLinearizeSize($x$)
7:    end for
8: else if $Xs$.type = $isStructureType$ then
9:    for each member $m$ in $Xs$ do
10:      $size$ += computeLinearizeSize($m$)
11:   end for
12:   ...
13: end if
14: ...
15: return $size$
```

Algorithm 2 linearizeIt(Xs, $size$)

```
1: ▷ allocate memory with the size of size
2: if $Xs$.type = $isPrimitive$ then
3:    copy($Xs$)
4: else if $Xs$.type = $isIterative$ or $Xs$.type = $isArray$ then
5:    for $x$ in $Xs$ do
6:       linearizeIt($x$)
7:    end for
8: else if $Xs$.type = $isStructureType$ then
9:    for each member $m$ in $Xs$ do
10:      linearizeIt($m$)
11:   end for
12:   ...
13: end if
14: ...
15: return $addressOfLinearizeData$
```

runtime libraries can support simpler APIs and efficiency by limiting to simpler data structures.

More specifically in our context, Chapel supports very general reductions, which can be applied to standard arrays of some primitive types, expressions over arrays, loop expressions, records of some mixed types and so on. For instance, `min reduce A+B` can be used in Chapel to find the minimum sum of corresponding elements from arrays A and B.

The basic linearization algorithm is shown as Algorithms 1 and 2. Both *computeLinearizeSize* and *linearizeIt* are recursive functions, and are used for computing the data size, which need to be linearized, and copying the data from the old data structure to the continuous memory space, respectively.

The algorithm considers a number of different data types in Chapel. The linearization of primitive types in Chapel, such as numeric (int, real), bool, string, and enumerated is straightforward, as these are single elements that are mapped directly to the intermediate C code. Linearizing an array can be reduced into linearizing each element of it. In addition, for an iterative expression like $A+B$ over which a reduction may be specified, the linearization function is invoked iteratively on each sum of corresponding elements in A and B.

A special consideration arises for records in Chapel. As a record is compiled into a *struct* in C. The *computeLinearizeSize* and *linearizeIt* steps are invoked iteratively on each member.

Figure 6, left-hand-side, shows a nested data structure in Chapel. The same data structure is shown graphically in Figure 7, along with a depiction of the corresponding linearized

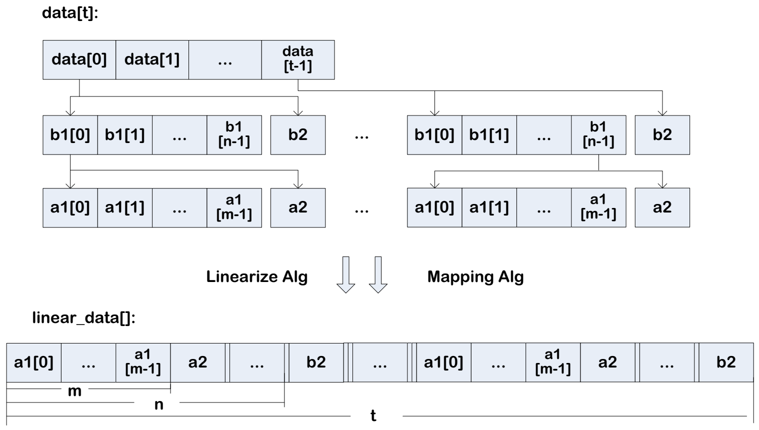

Fig. 7. Data Structure Before and After Linearization

data structure. The right-hand-side of Figure 6 shows some information we collect during linearization, to enable code generation for reductions.

C. Reduction Translation

We now explain the issues in reduction translation. The key conceptual challenge arises because the data structures have been linearized. Thus, operations now need to update the linearized structures, as opposed to the original Chapel structures.

A general *mapping* method has been designed and is shown as Algorithm 3. This algorithm can be divided into two main phases: in the first phase, during *linearization* of the reduction data, we record the necessary information. In the second phase, we compute the mapping index according to the original index and a recursive strategy, shown in Algorithm 3.

Algorithm 3 computeIndex($unitSize$[], $unitOffset$[][], $myIndex$[], $position$[][], i, $levels$)

```
1: ▷ During the linearization phase, collecting necessary information
2: if $i < levels - 1$ then
3:    $index$ = $unitSize[i] \times myIndex[i] + unitOffset[i][position[i][]]$
4:    $index$ += computeIndex($unitSize$[], $unitOffset$[][], $myIndex$[], $position$[][], $i$++, $levels$)
5: else
6:    $index = unitSize[i] \times myIndex[i]$
7: end if
8: return $index$
```

The algorithm uses the following parameters:
unitSize[]: This is a 1-dimensional array, which is used to store the unit size of the element in each level of the data, with unitSize[$levels - 1$] storing the inner-most element. This information should be collected during the linearization procedure.
unitOffset[][]: This is a 2-dimensional array, in which there are the offsets of the variables at each level. The first dimension is used to indicate the level and the second one indicates the position of the variables at the current level. This information should also be collected during the linearization procedure.
myIndex[]: This is a 1-dimensional array, which records the index for each level. This information is collected from the *accumulate* function of Chapel.

Before Linearization

```
record A {
    a1: [1..m] real;
    a2: int;
}
record B {
    b1: [1..n] A;
    b2: int;
}
data: [1..t] B;
```

Information Collected During Linearlization

```
levels = 3;
unitSize[levels] = {unitSize_B, unitSize_A, sizeof(data_type_a1)};
unitOffset[levels-1][2] = {{unitOffset_B[]}, {unitOffset_A[]}};
position[levels-1][2] = {{0, 1}, {0, 1}};
{*This should be collected in the accumulate function*}
myIndex[levels] = {i, j, k};
```

Fig. 6. A Data Structure Before Linearization (left) and the Information Collected During Linearization (right)

Before Linearization

```
for i in 1..t do {
    for j in 1..n do {
        for k in 1..m do {
            sum += data[i].b1[j].a1[k];
        }
    }
}
```

After Linearization

```
for i in 1..t do {
    for j in 1..n do {
        for k in 1..m do {
            index = computeIndex(unitSize, unitOffset,
                        myIndex, position, 0, levels);
            sum += linear_data[index];
        }
    }
}
```

Fig. 8. An Example of Using the Mapping Algorithm

position[][]: This is a 2-dimensional array, which provides the position information for calculating the $unitOffset$. This information is also collected during the linearization procedure.
levels: An indicator to store the total number of levels of the data.
i: An indicator to show current level, normally, it starts from 0, which means the current level is the outer-most one.

Earlier in Figure 6, we had shown the information that should be collected to apply the mapping algorithm. Most of the information should be collected during the *linearization* stage, while the index information is obtained from the reduction loop. One issue that should be noticed is as follows. Since we only use the member of $b1$ and $a1$ in each level of the record, the size of the second dimension for *position[][]* in line 3 of Algorithm 3 should be collected as 1. Because $b1$ and $a1$ are both in the first position of each level, the values in the array of position should be as follows: $position[0][0] = 0$, $position[1][0] = 0$.

The entire mapping process is recursive. It starts from the outer-most level and terminates with the inner-most. At each level, we calculate the offset caused by the index and the position.

To illustrate reduction processing with linearization, we show an example in Figure 8. The left-hand-side shows a reduction on the Chapel data structure, whereas the right-hand-side shows the same reduction on the linearized dataset. The code on right-hand-side can be trivially modified to invoke FREERIDE support for handling reductions.

By carefully observing Figure 8, we can identify an opportunity for optimization in the generated code. Since the inner-most level of the data is continuous, we can move the *computeIndex* function outside of the k loop, and only calculate the address of the first element in the inner-most level. Other addresses can be obtained by increasing the first index gradually one by one. This optimization method is evaluated in the next section.

V. Experimental Results

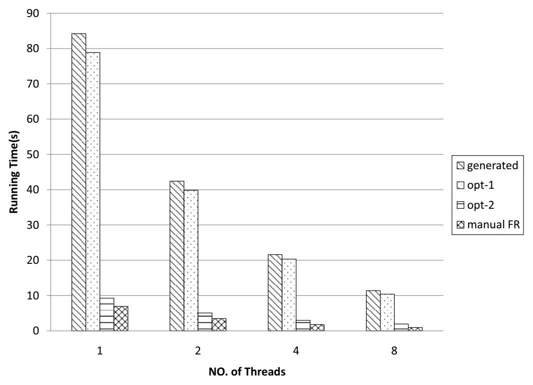

Fig. 9. K-means: Comparison of Performance of Different Versions (12 MB dataset, $k = 100$, $i = 10$)

This section evaluates the performance of our version of the Chapel, i.e, the one modified to use FREERIDE for reduction computations. There were two goals in our experiments. First, we compare the performance of the compiler generated code against hand-written data-intensive applications using FREERIDE middleware. Our second goal is to understand the reasons of the overheads in the compiler generated code, and to determine what optimizations can help reduce these overheads.

All the experiments were performed on an 8-core system with Intel Xeon CPU E5345 consist of two quad-core CPUs. The frequency for each core is 2.33 GHz and the main memory size is 6 GB. The operating system for our experiments is the 64-bit Linux. One thread is allocated on one CPU in all experiments.

In our experiments, we compare four versions. The `generated` is the C code generated by Chapel compiler with FREERIDE, without any optimizations. The next two versions

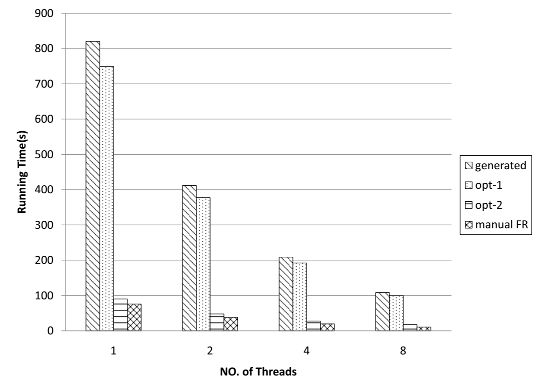

Fig. 10. K-means: Comparison of Performance of Different Versions (1.2 GB dataset, $k = 10$, $i = 10$)

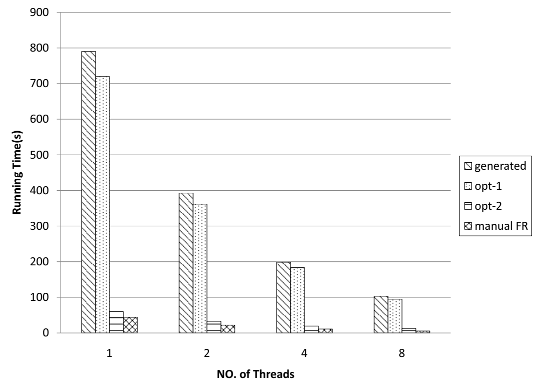

Fig. 11. K-means: Comparison of Performance of Different Versions (1.2 GB dataset, $k = 100$, $i = 1$)

improve this versions with a series of optimizations, and are referred to as `opt-1` and `opt-2`. The last version is `manual FR`, where the parallel application is written manually using the FREERIDE runtime system.

To further explain the two optimized versions, let us first consider the sources of overhead in the generated code. They are: 1) linearization of the input datasets, which has memory and computation overheads, 2) mapping, which involves computing the mapping index for the reduction data, and 3) accesses to complex Chapel structures. For example, for k-means, centroids (or cluster centers) are implemented using Chapel's *record*. Frequent accesses through a complex data structure cause significant overheads.

Among these, we currently do not try to reduce the first overhead, though in the future, a *pipelining strategy*, can be used to reduce this overhead. The `opt-1` and `opt-2` versions we report focus on the second and third costs.

Particularly, `opt-1` is similar to the classical strength reduction optimization. The *computeIndex* function is removed from the inner-most loop. The start point for the continuous data split is computed before the first iteration, and an appropriate pre-computed offset is added for each iteration. In `opt-2`, besides strength reduction described above, the frequently accessed output or temporary variables are only *linearized*, and are accessed through the mapping algorithm. Thus, the overheads of accessing complex and/or nested structures are eliminated.

We used two data mining applications that are suitable for processing using the FREERIDE middleware. The first application is k-means clustering [13], which was described in Section II.

The second application we use is Principal Component Analysis (PCA), which is a dimensionality reduction technique proposed in 1901. Specifically, PCA converts high-dimension data into the low-dimension one by calculating the *mean vector* and the *covariance matrix*. Normally, the dataset handled by PCA is in the form of a *data matrix*. There are two reduction phases in PCA: calculating the mean vector and computing the covariance matrix.

A. Results for K-means Clustering

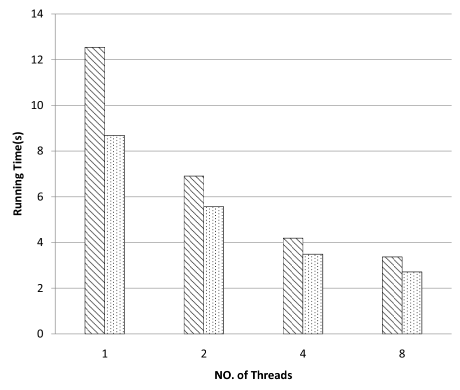

Fig. 12. PCA: Comparison of Performance of Different Versions (row: 1000, column: 10,000)

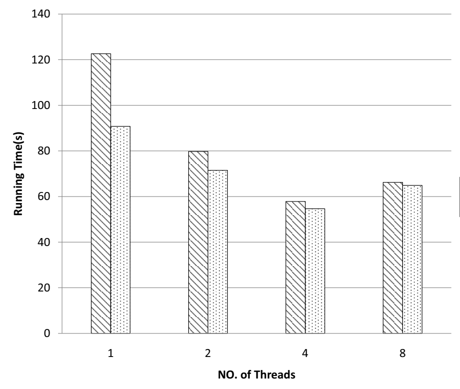

Fig. 13. PCA: Comparison of Performance of Different Versions (row: 1000, column: 100,000)

Two datasets are used for studies involving k-means clustering. The first dataset is small (12 MB), whereas the second is larger (1.2 GB). There are two key factors that impact the computations in k-means. The first is the number of the clusters or centroids and the second is the number of iterations. Figure 9 shows the results for the first dataset with 100 centroids (k) and 10 iterations (i). We can see that the running time can be deducted by a factor around 10% by the first optimization. For k-means, the centroid is chosen as the frequently accessed variable that is linearized in `opt-2` version. By using this optimization, the running time can be reduced by a factor around 8. Particularly, after this optimization, the difference between generated and hand-written code is very small. With 1 thread, this overhead is less than 20%.

All versions in Figure 9 show good scalability. One issue, however, is that the relative slow-down of `opt-2` version over the manual version increases as the number of threads increase. This is because linearization is done sequentially. This points to the need for performing linearization in parallel and/or overlapping linearization with processing of data.

Figures 10 and 11 show the results from execution on the larger dataset (1.2GB), with 10 centroids and 10 iterations and 100 centroids and only 1 iteration, respectively. The trends from Figure 10 are very similar to the results from the small dataset. The overheads of linearization are higher in Figure 11, as the computation is performed over a single iteration.

B. Results from PCA

Two datasets are used for this experiment: one is `small`, with 1000 rows and 10,000 columns, and the other is `large`, with 1000 rows and 100,000 columns. Note that the number of rows denotes the dimensionality of the dataset, whereas the number of columns denotes the number of data elements.

Figure 12 and Figure 13 show the experimental results. We are only comparing two versions of the application here. PCA is a compute-intensive application and does not use complex or nested data structures in Chapel. As a result, the benefits of the two levels of optimizations we have developed are not significant. Thus, we show only two versions of this application, which are `opt-2` and `manual FR`. Similar to the previous application, the overheads of `opt-2` over manual are within 20%. For both datasets, there is good scalability up to 4 threads. Additional speedups for 8 threads are limited because of the difficulty in achieving perfect load balance for this application.

VI. Conclusions

We have presented a case study for the possible use of a new HPC language for data-intensive computations. Particularly, we show how reduction features of Chapel can be used to express data mining computations. Moreover, by a series of transformations, the Chapel compiler can automatically generate calls to a specialized runtime library that supports the same class of computations. Thus, we combine the productivity of a modern language with the performance of a specialized runtime system.

Acknowledgments

This work was supported by NSF awards CCF-0833101 and IIS-0916196 to the Ohio State University and by the Defense Advanced Research Projects Agency under its Agreement No. HR0011-07-9-0001 to Cray Inc.

References

[1] W. Blume, R. Doallo, R. Eigenman, J. Grout, J. Hoelflinger, T. Lawrence, J. Lee, D. Padua, Y. Paek, B. Pottenger, L. Rauchwerger, and P. Tu. Parallel Programming with Polaris. *IEEE Computer*, 29(12):78–82, December 1996.

[2] Randal E. Bryant. Data-Intensive Supercomputing: The Case for DISC. Technical Report CMU-CS-07-128, School of Computer Science, Carnegie Mellon University, 2007.

[3] Barbara M. Chapman, Lei Huang, Eric Biscondi, Eric Stotzer, Ashish Shrivastava, and Alan Gatherer. Implementing OpenMP on a High Performance Embedded Multicore MPSoC. In *Proceedings of IPDPS 2008*, 2008.

[4] Philippe Charles, Christian Grothoff, Vijay A. Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph von Praun, and Vivek Sarkar. X10: an Object-oriented Approach to Non-uniform Cluster Computing. In Ralph E. Johnson and Richard P. Gabriel, editors, *OOPSLA*, pages 519–538. ACM, 2005.

[5] Jeffrey Dean and Sanjay Ghemawat. Map-Reduce: Simplified Data Processing on Large Clusters. In *Proceedings of OSDI*, pages 137–150, 2004.

[6] Steven J. Deitz, David Callahan, Bradford L. Chamberlain, and Lawrence Snyder. Global-view Abstractions for User-defined Reductions and Scans. In *Proceedings of the eleventh ACM SIGPLAN symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 40–47, 2006.

[7] J. Ekanayake, S. Pallickara, and G. Fox. Map-Reduce for Data Intensive Scientific Analyses. In *IEEE Fourth International Conference on e-Science*, pages 277–284, Dec 2008.

[8] Dan Gillick, Arlo Faria, and John Denero. Map-Reduce: Distributed Computing for Machine Learning. 2008.

[9] Leo Glimcher, Gagan Agrawal, Sameep Mehta, Ruoming Jin, and Raghu Machiraju. Parallelizing a Defect Detection and Categorization Application. In *Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS)*, 2005.

[10] Leo Glimcher, Xuan Zhang, and Gagan Agrawal. Scaling and Parallelizing a Scientific Feature Mining Application Using a Cluster Middleware. In *Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS)*, 2004.

[11] Bingsheng He, Wenbin Fang, Qiong Luo, Naga K. Govindaraju, and Tuyong Wang. Mars: a Map-Reduce Framework on Graphics Processors. In *Proceedings of PACT 2008*, pages 260–269. ACM, 2008.

[12] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. Dryad: Distributed Data-parallel Programs From Sequential Building Blocks. In *Proceedings of the 2007 EuroSys Conference*, pages 59–72. ACM, 2007.

[13] A. K. Jain and R. C. Dubes. *Algorithms for Clustering Data*. Prentice Hall, 1988.

[14] Wei Jiang, Vignesh Ravi, and Gagan Agrawal. Comparing Map-Reduce and FREERIDE for Data-Intensive Applications. In *Proceedings of Conference on Cluster Computing*, 2009.

[15] Wei Jiang, Vignesh Ravi, and Gagan Agrawal. A Map-Reduce System with an Alternate API for Multi-Core Environments. In *Proceedings of Conference on Cluster Computing and Grid (CCGRID)*, 2010.

[16] Ruoming Jin and Gagan Agrawal. A Middleware for Developing Parallel Data Mining Implementations. In *Proceedings of the first SIAM conference on Data Mining*, April 2001.

[17] Ruoming Jin and Gagan Agrawal. Shared Memory Parallelization of Data Mining Algorithms: Techniques, Programming Interface, and Performance. In *Proceedings of the second SIAM conference on Data Mining*, April 2002.

[18] Yuan Lin and David Padua. On the Automatic Parallelization of Sparse and Irregular Fortran programs. In *Proceedings of the Workshop on Languages, Compilers, and Runtime Systems for Scalable Computers (LCR - 98)*, May 1998.

[19] Bo Lu and John Mellor-Crummey. Compiler Optimization of Implicit Reductions for Distributed Memory multiprocessors. In *Proceedings of the 12th International Parallel Processing Symposium (IPPS)*, April 1998.

[20] Christopher Olston, Benjamin Reed, Utkarsh Srivastava, Ravi Kumar, and Andrew Tomkins. Pig Latin: a Not-so-foreign Language for Data Processing. In *Proceedings of SIGMOD Conference*, pages 1099–1110. ACM, 2008.

[21] Rob Pike, Sean Dorward, Robert Griesemer, and Sean Quinlan. Interpreting the Data: Parallel Analysis with Sawzall. *Scientific Programming*, 13(4):277–298, 2005.

[22] Colby Ranger, Ramanan Raghuraman, Arun Penmetsa, Gary R. Bradski, and Christos Kozyrakis. Evaluating Map-Reduce for Multi-core and Multiprocessor Systems. In *Proceedings of 13th HPCA*, pages 13–24. IEEE Computer Society, 2007.

[23] Sangwon Seo, Ingook Jang, Kyungchang Woo, Inkyo Kim, Jin-Soo Kim, and Seungryoul Maeng. Hpmr: Prefetching and Pre-shuffling in Hared Map-Reduce Computation Environment. In *Proceedings of the 2009 IEEE Cluster*. IEEE, 2009.

[24] Ashish Thusoo, Joydeep Sen Sarma, Namit Jain, Zheng Shao, Prasad Chakka, Suresh Anthony, Hao Liu, Pete Wyckoff, and Raghotham Murthy. Hive - a Warehousing Solution over a Map-Reduce Framework. *PVLDB*, 2(2):1626–1629, 2009.

[25] Hao Yu and Lawrence Rauchwerger. Adaptive Reduction Parallelization Techniques. In *Proceedings of the 2000 International Conference on Supercomputing*, pages 66–75. ACM Press, May 2000.

[26] Matei Zaharia, Andy Konwinski, Anthony D. Joseph, Randy H. Katz, and Ion Stoica. Improving Map-Reduce Performance in Heterogeneous Environments. In *Proceedings of OSDI*, pages 29–42. USENIX Association, 2008.