Proceedings of the World Congress on Engineering 2011 Vol II
WCE 2011, July 6 - 8, 2011, London, U.K.

Improving Performance of Collection-Oriented Operations through Parallel Fusion

Bruce P. Lester

Abstract—**To more fully utilize the potential offered by multi-core processors, programming languages must have features for expressing parallelism. One promising approach is collection-oriented operations, which are easily expressed by the programmer and can be implemented by the runtime system in a parallel fashion for improved performance. However, the ordinary implementation requires a barrier synchronization among all the processors after each parallel operation, thereby creating a performance penalty that grows with the number of processors. For large numbers of processors, this inhibits scalability and reduces performance especially for smaller size data sets. This paper explores a optimization technique called operator fusion, which removes the necessity for barrier synchronization. The general principles and rules governing the use of operator fusion are described and then illustrated with a specific collection-oriented parallel library that we have developed for the object-oriented programming language Scala, which is an extension of the language Java. Performance improvement resulting from operator fusion is analyzed for several benchmark programs on a computer with a multi-core processor.**

Index Terms— **data parallel, multi-core processor, parallel programming, Scala**

I. Introduction

To help the programmer specify parallelism in a program, the programming language must have some special parallel programming features. The predominant approach used so far is multi-threading, in which the programmer explicitly assigns computing tasks to individual parallel threads. If the parallel threads modify shared data, then locking is used to provide atomic access. This approach has several drawbacks. The programmer is involved in many of the low-level details of management and synchronization of parallel tasks. Also, multi-threaded programs have potential data races that essentially create a nondeterministic program: a program that may produce different outputs for the same input data during different executions. Program deadlocks may also occur in a nondeterministic fashion. This nondeterminism complicates the software development process, and makes it more difficult to develop reliable software.

One promising approach to solve many of these problems is high-level *collection-oriented* operations, in which every element of the collection is operated upon in parallel by the same operation. This is often called *data parallel* programming. One example is the array operations of the language Fortran 90 [1], which may have a sequential or parallel implementation. A more sophisticated set of operations is found in High-Performance Fortran [2, 3], including data distribution directives and user-defined data parallel functions. The widely publicized MapReduce [4] operation used by Google is another example of a collection-oriented parallel operation.

One of the earliest commercial applications of data parallel programming during the late 1980s was in the Connection Machine [5] of Thinking Machines Corporation. The programming languages available for the Connection Machine included data parallel versions of both Lisp and C. Much of what was known at that time about data parallel programming is summarized in the book by Guy Blelloch, *Vector Models for Data-Parallel Computing* [6]. Looking back even earlier, the array operations of the language APL [7] can be considered as primitive examples of collection-oriented operations that can have a data parallel implementation. In the case of APL, the array operations were not introduced for the purpose of parallel execution, but simply to make the programming process easier by providing higher level programming abstractions.

More recent examples of data parallel languages include Ct [8], a language under development by Intel for their experimental terascale processor architectures. The company RapidMind has successfully marketed a data parallel extension of the C++ language with collective operations on arrays [9]. Intel has just released a software package called Array Building Blocks [10] that combines and extends many of the features of Ct and the RapidMind extensions. Researchers at Stanford University have developed a data parallel extension of the C language called Brook [11], intended for efficient execution on computers with GPU coprocessors. Brook allows user-defined data-parallel functions on streams, which are essentially large data arrays. The language X10 under development by IBM [12] and HPJava [13] are both data parallel versions of Java, intended for scientific and engineering applications on high-performance computer clusters.

II. Operator Fusion

One of the main performance issues in data parallel programming is the need for synchronizing the processors after each data parallel operation. Data parallel operations are more widely applicable for general purpose programming if they are fairly primitive in nature. Then the programmer can construct a parallel program by combining

Manuscript received March 3, 2011; revised April 5, 2011.
Bruce P. Lester is Professor, Computer Science Department, Maharishi University of Management, Fairfield, Iowa 52556 USA (e-mail: blester@mum.edu).

large numbers of these primitive operations. However, the *barrier synchronization* required after each data parallel operation creates a performance penalty, which grows with the number of processors. For large numbers of processors, this inhibits scalability and reduces performance especially for smaller size data sets.

In this paper, we explore a technique called *operator fusion*, which removes the necessity for barrier synchronization after each data parallel operation. Data parallel operations are implemented by dividing the work among the available processors. To be sure that the operation is complete, all of the processors execute a barrier before moving on to the next operation. However, under certain circumstances, it is possible for a processor to move immediately to the next data parallel operation without waiting for the other processors. For example, consider a sequence of four data parallel operations: P, Q, R, S. In an ordinary parallel implementation, all processors are required to finish their assigned portion of the computing for operation P, and then execute a barrier before any can begin on operation Q. Similarly, all processors must finish their work on operation Q before beginning R. With an implementation based on operator fusion, a processor completes its work on data parallel operation P and then moves on to operation Q immediately; similarly for operations R and S. Thus, the sequence of four data parallel operations can be executed with only one barrier operation after S, instead of four barriers in the implementation without operator fusion.

In this paper, we explore the use of *operator fusion* in a collection-oriented parallel library. The library is an add-on to an ordinary object-oriented programming language. The library implementation is done completely at runtime. In section III we present the general principles underlying the operator fusion optimization, including a general algorithm for determining when it can be used. In subsequent sections, we describe our collection-oriented library and analyze the performance improvement resulting from operator fusion for several benchmark programs.

Removing processor synchronization barriers to improve performance of parallel programs is not a new idea. Some parallel programming languages have explicit instructions that the programmer can use to indicate a barrier is not necessary in certain circumstances. For example, the language OpenMP allows a *NO WAIT* directive to prevent a barrier synchronization among the threads executing a parallel loop. In contrast to this, we are concerned with *automatic* operator fusion, done completely at runtime by the collection-oriented library without any knowledge or intervention by the programmer.

The Intel Array Building Blocks (ArBB) library for C++ does include some automatic operator fusion. However, this optimization is applied only in the limited context of function bodies that are invoked with a special ArBB *call* operation. Furthermore, all of the ordinary C++ flow of control instructions (*for*, *while*, *if*) in the function body must be replaced by special ArBB flow of control operations. In contrast to this, our implementation of operator fusion is automatically applied to every individual collection-oriented operation in the library at runtime.

III. General Principles

The first step is to explore the general principles of operator fusion and develop a simple algorithm for determining when operator fusion is possible. For this purpose, consider a very general framework with a User Program written in any high-level language. Embedded at various points in this User Program are calls to data parallel operations. These calls may be features of the programming language, or simply calls to library functions (methods). To allow the possibility of operator fusion of the data parallel operations, two simple assumptions are needed: *isolation* and *partitioning*. These will be explored in the next two subsections.

A. Isolation of Data Parallel Operations

The first assumption is the existence of a clean interface between the User Program and the data parallel operations. The data parallel operations perform transformations on a special group of collections (data structures), which we call *Data Parallel Collections* (abbreviated: *DP-Collections*). The User Program interacts with the DP-Collections through a fixed set of Data Parallel Operations (abbreviated: *DP-Operations*). The User Program passes parameters to these DP-Operations, which are used to carry data into the operations and return data back to the User Program. However, the User Program has no direct access to the DP-Collections, except via one of these special DP-Operations. Furthermore, the DP-Operations have no *side-effects*: they can only read/write the DP-Collections and the data passed as parameters from the User Program. In other words, the DP-Operations are *isolated* from the User Program data, and similarly the User Program is *isolated* from the DP-Collections.

This *isolation* assumption is very reasonable and will probably be valid for a wide range of data parallel libraries, beyond the specific data parallel library described in subsequent sections of this paper. For example, the Intel Ct library [8] and Array Building Blocks library [10] both satisfy the isolation property. For now, let us determine to what extent operator fusion of data parallel operations is possible based only on this simple *isolation* assumption. For this purpose, a useful analytical tool is a *History Graph* of the DP-Operations. During each specific execution of the User Program, a series of DP-Operations (d_1, d_2, …, d_n) will be generated. Each DP-Operation d_i will have parameters, some of which may be a reference to a specific DP-Collection, and some of which may be a reference to a User Program data value (object). The execution history of the DP-Operations defines a directed, acyclic graph as follows:

- Each executed DP-Operation d_i is a node in the graph. The index i is called the sequence number or *timestamp* of the operation.
- Each DP-Collection referenced by a parameter of any DP-Operation is a node in the graph.
- Each User Program data value (object) referenced by a parameter of the DP-Operation is a node in the graph.

ISBN: 978-988-19251-4-5
ISSN: 2078-0958 (Print); ISSN: 2078-0966 (Online)

- If DP-Collection c is read by DP-Operation d_i , there is an edge from c to d_i. If c is modified by d_i , there is an edge from d_i to c.
- If User Program data value u is read by DP-Operation d_i , there is an edge from u to d_i. If u is modified by d_i , there is an edge from d_i to u.

An example of a History Graph is shown in Fig. 1. The particular sequence of DP-Operations (d_1, d_2, d_3, d_4) in the history is generated by the User Program, depending on the input data. We are not suggesting that such a History Graph actually be constructed during the execution of a real program. We are just using the graph as a conceptual tool to help analyze and understand the principles of operator fusion. Each DP-Operation in the graph is executed by a team of Worker Threads running in parallel. Now consider the following question: under what conditions can the barrier synchronization after each DP-Operation be safely removed?

For any given DP-Operation operation d_i in the graph with an input DP-Collection v, let I(d_i, v, m) denote the set of data elements in DP-Collection v that are directly read by Worker Thread m during DP-Operation d_i. For DP-Operation d_j with an output DP-Collection v, let O(d_j, v, k) denote the set of data elements in DP-Collection v that are directly written by Worker Thread k during DP-Operation d_j.

A data element e of a DP-Collection v is said to be a *cross-thread* data element if there exist DP-Operations d_i and d_j, such that $e \in$ O(d_j, v, k) and $e \in$ I(d_i, v, m) and $k \neq m$. In simple words, a *cross-thread* data element is one that is created (written) by one Worker Thread and then consumed (read) by a different Worker Thread. Cross-thread data items restrict the possibilities for operator fusion of the DP-Operations. If Worker Thread k writes a cross-thread data item e during DP-Operation operation d_j, and e is read by a different Worker Thread m during a subsequent DP-Operation operation d_i, then some kind of barrier synchronization among the Worker Threads is required after operation d_j. Otherwise, Worker m might attempt to read data item e before it is created by Worker k.

If the output DP-Collection v of any DP-Operation operation d_i in the history graph has no cross-thread data items, then no barrier synchronization is required after operation d_i. Thus, all the Worker Threads involved in the execution of operation d_i can immediately move on to the next DP-Operation operation d_j as soon as they complete their share of operation d_i. Thus, each Worker Thread experiences a *fusion* of its computing on operations d_i and d_j.

The general discussion of the last few paragraphs has assumed that the output of the DP-Operation is a DP-Collection. However, some DP-Operations may produce an output data value (object) that is not a DP-Collection. As an example, consider operation d_4 and its output x in Fig. 1. A reference to this object x is returned to the User Program by operation d_4. Since x is not a DP-Collection, the User Program may directly access the data of x. Thus, it is necessary to make sure the Worker Threads have completed their computation of x before returning to the User Program after the call to operation d_4. Thus, barrier synchronization is required after operation d_4 that includes all the Worker

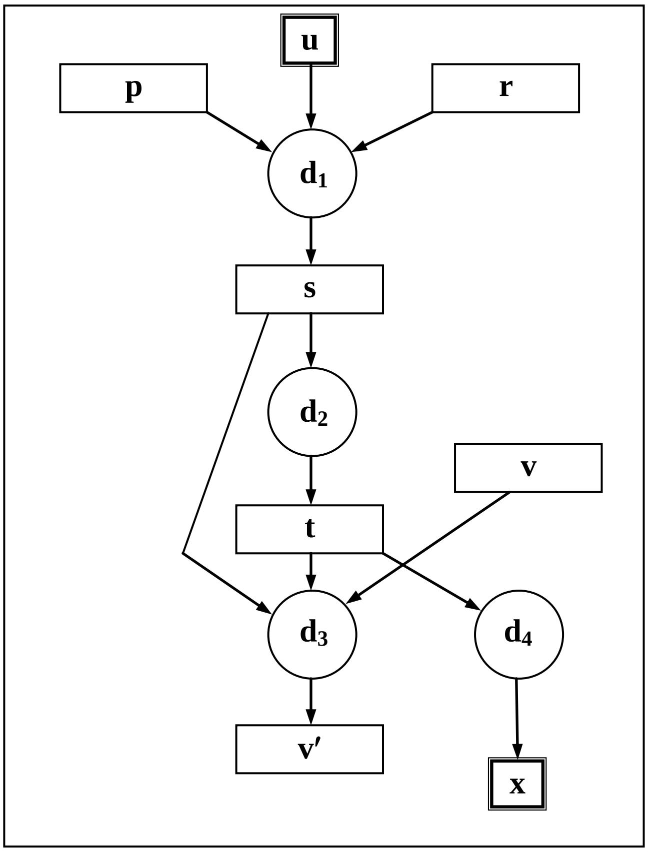


Figure 1. History Graph of PVector Operations.

Threads and also the Master Thread executing the User Program. When the Master Thread is included, we call it a *Strong Barrier*. However, as long as the output of any DP-Operation is a DP-Collection, then a strong barrier is not necessary.

The situation is similar for an input parameter to a DP-Operation that is not a DP-Collection, for example input u to DP-Operation d_1 in Fig. 1. The User Program may have another reference to data value (object) u and attempt to modify it. Therefore, d_1 must complete its work before the User Program is allowed to continue. Thus, a strong barrier is needed after d_1, unless u is an *immutable* object, which cannot be modified.

B. Partitioning of Data Parallel Collections

The above general discussion of operator fusion is based completely on the assumption of *isolation* between the User Program and DP-Collections. Now one additional assumption will allow the development of a simple and practical operator fusion algorithm: each DP-Collection has a standard (default) *partitioning*. The *partitions* are disjoint and cover the whole DP-Collection. The overall purpose of the partitioning is to facilitate data parallelism. Each Worker Thread can be assigned to work on a different partition in parallel with no interference.

Partitioning facilitates operator fusion if the same partitioning is used by many different DP-Operations. For example, consider a DP-Collection v with partitions p_1, p_2, …, p_m. Now assume that Worker Threads W_1, W_2, …, W_m are assigned to work on these partitions independently in parallel during a particular DP-Operation d_i. If the same group of m Worker Threads is assigned to the partitions in the same way during the subsequent DP-Operations d_{i+1},

ISBN: 978-988-19251-4-5
ISSN: 2078-0958 (Print); ISSN: 2078-0966 (Online)

then there is no need for a barrier after d_i — fusion of operations d_i and d_{i+1} is possible without introducing any data races or timing-dependent errors. After completing its share of the computing in operation d_i, Worker k can move on immediately to operation d_{i+1} without waiting for the other Workers — there is no need for a barrier after operation d_i. Using the terminology of the previous section, the standard *partitioning* prevents the possibility of any *cross-thread* data elements in this particular DP-Collection v.

At this stage of analysis, we are not specifying any details about the nature of the data structures allowed in the DP-Collections or the particular partitioning method. We only assume there is some standard partitioning method for each DP-Collection. If this standard partitioning is used by many of the DP-Operations for allocating Worker Threads, then there will be a lot of opportunity for fusion of the DP-Operations. However, all DP-Operations are not *required* to adhere to the standard partitioning. Some DP-Operation may use a different partitioning or may not have any distinct partitioning at all, in which case these DP-Operations will not be candidates for operator fusion.

This assumption of a standard (default) *partitioning* for each DP-Collection, along with the *isolation* assumption from Section IIIA, will allow us to develop a simple *operator fusion algorithm* to determine whether specific DP-Operations require a barrier synchronization or not. The input to this algorithm will be a descriptor for each DP-Operation that specifies certain important properties of that operation.

Each DP-Operation d has one or more input parameters, some of which may be DP-Collections, and an output parameter which may be a DP-Collection (see the History Graph of Fig. 1 for examples). For each of these DP-Collection parameters, the operator fusion algorithm needs to know whether or not DP-Operation d adheres to the standard partitioning of that DP-Collection. If the standard partitioning of any input parameter is violated, this is called an *input crossing*; similarly, violation for an output parameter is called an *output crossing*.

C. The Operator Fusion Algorithm

Following is summary of the seven properties of each DP-Operation that will serve as input to the operator fusion algorithm:

- **InputCross**: true if this operation has an input crossing
- **OutputCross**: true if this operation has an output crossing
- **OutDP**: true if the output of this operation is a DP-Collection
- **InUserData**: true if any of the inputs to this operation is not a DP-Collection
- **OutputNew**: true is the output of this operation is a newly created DP-Collection
- **HasBarrier**: true if this operation has an internal barrier synchronization of the Worker Threads
- **HasStrongBarrier**: true if this operation has an internal strong barrier synchronization

These seven properties are *static* – they only need to be determined once for each DP-Operation in the library. The properties do not depend on the particular User Program, but only on the definition of the DP-Operations and the particular implementation of the operations.

In addition to the above static information, the operator fusion algorithm also needs some *dynamic* information that must be gathered during the execution of the User Program. Each call to any DP-Operation by the User Program is assigned a unique *sequence number*, which serves as a kind of time stamp. Each DP-Collection also has a unique time stamp: the sequence number of the DP-Operation that created it. This is assigned dynamically when the DP-Collection is created. Also, each specific DP-Collection has a time stamp (sequence number) of the most recent DP-Operation to perform an input crossing on it (*InCrossTime*). One additional piece of dynamic information required is the time stamp of the most recent barrier operation.

The operator fusion algorithm is shown in Fig. 2 and has there separate procedures. *enterOp* is called by each Worker Thread before beginning each DP-Operation to determine whether a barrier is needed before execution of the DP-Operation. *exitOp* is called by each Worker Thread after each DP-Operation is complete to determine whether a barrier is needed before moving on to the next DP-Operation. *userProg* is called by the User Program after each DP-Operation call to determine whether the User Program must participate in a strong barrier with the Worker Threads before continuing.

All three procedures in this operator fusion algorithm depend heavily on the properties of the DP-Operations, which is contained in the array *OpInfoTab*. For example, `OpInfoTab[opcode].InputCross` is true if the DP-Operation identified by *opcode* has an input crossing on one of its DP-Collection input parameters. The algorithm also uses the time stamps of the DP-Operations and the DP-Collections. The variable *barrierTime*, which is the time stamp of the most recent barrier, is used and modified during the algorithm. The procedures of this algorithm have no loops and can therefore be executed in constant time.

The focus of this operator fusion algorithm is to detect those specific conditions that require a barrier operation, and avoid a barrier whenever possible. The algorithm is quite general and is based only the two assumptions described earlier: *isolation* and *partitioning*. The implementation dependent aspects of the algorithm are captured by the seven properties of the DP-Operations as found in the array *OpInfoTab*.

Now we will apply this operator fusion algorithm to a specific collection-oriented parallel library for the language Scala. The subsequent sections of this paper describe the library in detail and give an example program that uses the library to solve a partial differential equation. We also summarize the results of performance studies on several benchmark programs showing that operator fusion significantly improves the performance of the library.

IV. Data Parallel Library

We wish to be as general as possible in describing the

ISBN: 978-988-19251-4-5
ISSN: 2078-0958 (Print); ISSN: 2078-0966 (Online)

```
// OpInfoTab:  properties of DP-Operations
// barrierTime:  time of most recent barrier

enterOp(opCode, opTimeStamp, inData) {
 /* opCode: identifies the DP-Operation
    opTimeStamp: sequence no. of DP-
Operation
    inData: reference to the input
      DP-Collection with an input crossing
      (if any, for this DP-Operation)
 */
 if ((OpInfoTab[opCode].InputCross
     && inData.timeStamp > barrierTime)
   ||
    (!OpInfoTab[opCode].OutputNew
     && opTimeStamp > barrierTime+1
     && inData.InCrossTime > barrierTime)) {
   Execute a barrier
   barrierTime = opTimeStamp - 1
    }
 if (OpInfoTab[opCode].InputCross)
   inData.InCrossTime = opTimeStamp
}
  ---------------------------------
exitOp(opCode, opTimeStamp, outData) {
/* opCode: identifies the DP-Operation
   opTimeStamp: sequence no. of DP-Operation
   outData: reference to output DP-
Collection
*/
 if (OpInfoTab[opCode].OutDP){
   if (OpInfoTab[opCode].OutputCross
      && !OpInfoTab[opCode].HasBarrier) {
   Execute a barrier
   barrierTime = opTimeStamp
    }
 }
 else // output is not a DP-Collection
  if(!OpInfoTab[opCode].HasStrongBarrier) {
   Execute a strong barrier
   barrierTime = opTimeStamp
  }
 if (OpInfoTab[opCode].InUserData) {
   Execute a strong barrier
   barrierTime = opTimeStamp
 }
 else if (OpInfoTab[opCode].HasBarrier)
     barrierTime = opTimeStamp
 if (OutData != null)
   OutData.timeStamp = opTimeStamp
}
  ---------------------------------
userProg(opCode){
// opCode: identifies the DP-Operation
  if ((OpInfoTab[opCode].InUserData
       || !OpInfoTab[opCode].OutDP)
     && !OpInfoTab[opCode].HasStrongBarrier)
  Execute a strong barrier
}
```

Figure 2. Operator Fusion Algorithm.

principles and practice of operator fusion. However, to illustrate the principles and show that the technique is practical, it is necessary to focus on a specific implementation. For this purpose, we use a collection-oriented parallel library for the object-oriented language *Scala*, which extends the language Java by adding functional programming features. Scala is compiled into Java byte-code and is executed by the Java Virtual Machine. Any of the Java library functions may be called from within a Scala program. We chose Scala [14] as our implementation language because it is particularly well suited for creating runtime libraries. However, the collection-oriented parallel library presented in this paper could also be implemented with operator fusion in Java, C#, or any object-oriented language. Runtime operator fusion in data parallel libraries is in no way limited to the language Scala.

The basic parallel collection object we use in our library is called a *Parallel Vector* (abbreviated *PVector*). A *Parallel Vector* is an indexed sequence of data items, which bears some resemblance to a one-dimensional array. However, the range of operations available for Parallel Vectors is really quite different from a simple array, as described in the subsequent sections of this paper. Parallel Vectors are implemented in Scala with a generic library class `PVector[T]`. To create an instance of PVector in a Scala program, one must supply a specific type (or arbitrary class name) for the generic type `[T]`.

The PVector class in our data parallel Scala library provides several constructors for creating and populating new PVector objects. The PVector class also has a variety of methods that can be invoked by the user program to transform and/or combine PVector objects. The calls to the PVector constructors and methods are imbedded in the user program. The implementation of the constructors and methods is done completely within the library using parallelism. Thus, the parallelism is essentially hidden from the user program. The user does not have to deal with the complexities and problems associated with parallel programming, as briefly described in the Introductory section of this paper.

The *fusion* of the PVector operations is also contained within the library implementation, and is therefore hidden from the user program. Thus, the user may view the PVector as just another type of collection with a set of available operations implemented in the library. Using the terminology of section III, the PVectors are the DP-Collections, and PVector methods are the DP-Operations.

Our data parallel Scala library currently implements a total of fifteen primitive operations on PVectors. For purposes of understanding, these can be divided into five major categories: Map, Reduce, Permute, Initialize, Input/Output. Following is a brief description of the operations contained in each of these categories. This discussion uses some Scala code segments. Readers unfamiliar with Scala may refer to [14] or any of the online Scala tutorials that are easily found on the internet. However, the Scala syntax is so similar to Java that it should be understandable by any reader who has some familiarity with Java or C#.

A. Map Operations

The *map* operation is a very powerful data parallel operation that applies a user-defined function to each element of a PVector. The abstract execution model for this application is a virtual processor operating in parallel at each element of the PVector. In practice, this may be implemented in the library using a combination of parallel and sequential execution. Consider a `PVector[Boolean]` called *Mask*. The *map* method can be invoked as follows to create a new PVector whose elements are the logical negation of *Mask*:

```
B = Mask.map( !_ )
```

The notation "!_" represents an anonymous function with one parameter whose output is the logical negation of the input. As a complement to the *map* operation, our data parallel library also contains an operation called *combine* that has two input PVectors of the same generic type *T* and creates a single output PVector of generic type *U*. The *combine* method can be invoked to create a new PVector from the sum of the corresponding elements of PVectors *A* and *B*:

```
C = A.combine[Int]( _+_ , B )
```

The notation "_+_" represents an anonymous function with two parameters, whose output is the sum of the inputs. As with the *map* operation, the abstract execution model for this application is a virtual processor operating in parallel at each element of the PVector.

B. Reduce Operations

The *Map* operations work element-by-element on the inputs, and produce an output PVector with the same dimension. Whereas, the *Reduce* operations combine the elements of the input PVector. To allow the *Reduce* operations to be as general as possible, they also allow a user-defined function. Three basic operations are *reduce*, *scan*, and *keyed-reduce*. The following *reduce* operation sums the elements of the PVector *A*:

```
A = new PVector[Int](aListofIntegers)
result = A.reduce(_+_)
```

The *scan* operation is similar to *reduce*, except the reduction is performed on each prefix of the PVector. This is sometimes called a *parallel prefix* operation. The result of a *scan* is a PVector with the same base type and number of elements as the original. Element *i* of the output of the scan is defined as the reduction of the elements 0 to *i* of the input PVector.

A more general type of *Reduce* operation is the *keyed-reduce*, which in addition to the input data PVector also has two additional PVector parameters: the *Index* and the *Target*. The *Target* vector is the starting point for the output of the *keyed-reduce*, and must have the same element type as the *Data* vector, but possibly a different number of elements. The *Index* vector is a `PVector[Int]` with the same length as the *Data* vector. The *Index* vector specifies the destination location in the *Target* vector for each element of the *Data* vector. If the *Index* maps several data values to the same location in the *Target*, they are combined using the user-defined reduction operation.

C. Permute Operations

The *Permute* operations allow the elements of a PVector to be selected and/or reordered. They are all methods of the PVector class. Using the conceptual execution model with a virtual processor for each element of the PVector, we may intuitively think of the *Permute* operations as collective communication operations among the virtual processors. The simplest of these operations is called *permute*, and simply reorders the elements of an input `PVector[T]`, as illustrated in the following simple example:

Data Input:	[30 5 -2 10]	
Index:	[3 0 1 2]	(index in Data vector)
Output:	[10 30 5 -2]	

The *select* operation creates an output PVector by selecting a subset of the elements of the input *Data* PVector. The selection process is done using a boolean *Mask* with the same number of elements as the *Data* PVector. Elements in the *Data* PVector with a *true* value in *Mask* are copied to the output. Thus, the number of elements in the output will be less than or equal to the number in the original. The *select* operation simply creates a subset of the elements from the original *Data* vector in the same order as they appear in the *Data* vector.

D. Initialize Operations

The *Initialize* operations allow new PVectors to be created with initial data. One of the PVector constructors called a *Broadcast* operation can be considered as a member of this class of operations. The following creates an integer PVector with initial value 0 for all the *n* elements:

```
Zero = new PVector[Int](n,0)
```

The other *Initialize* operations are methods in the PVector class. The *Index* operation creates a PVector of length *n* with element values 0, 1, 2, …, *n*-1. The *append* operation creates a new PVector from the concatenation of two existing PVectors. The *assign* operation copies a source PVector into the destination PVector, which is the one that calls the *assign* method. The *assign* operation is quite different from the ordinary assignment denoted by '='. Consider the following two instructions using PVectors *A* and *B*, which both have the same base type:

```
B = A
B.assign(A)
```

In Scala, as in Java, a variable like *A* or *B* contains a reference to an object – in this case a reference to a PVector object. The first instruction (ordinary assignment) makes a copy of the object reference in variable *A* and writes it into variable *B*, so that *A* and *B* then refer to the same PVector object. Whereas, the second instruction (*assign*) copies the individual elements from the PVector *A* into the corresponding elements of PVector *B*. For the *assign* operation to succeed, PVectors *A* and *B* must conform: the same number of elements and the same base type.

E. Conditional Execution Using Masks

In many parallel algorithms, it is sufficient to have every virtual processor apply the same computation in parallel to its assigned element of the PVector. However, in more complex algorithms it is sometimes desirable to have the virtual processors apply different operations. This can be implemented by using a boolean PVector called a *Mask*. A *true* value in the *Mask* selects one operation, and a *false* value selects a different operation. This is analogous to an *if* statement in an ordinary program. This feature is implemented in our data parallel Scala library using an object called *Where*, as illustrated in the following example which sets each b_i to $1/a_i$:

ISBN: 978-988-19251-4-5
ISSN: 2078-0958 (Print); ISSN: 2078-0966 (Online)

```
A = new PVector[Int](aList)
Zero = new PVector[Int](n,0)
Where.begin(A != 0)       //  where A != 0
  B = A.map(1/_)          //    B = 1/A
Where.elsewhere           //  elsewhere
  B.assign(Zero)          //    B = 0
Where.end( )
```

In the above, a boolean mask is created by comparing each element of a PVector *A* to zero (`A!=0`). A *true* value in the mask indicates the corresponding element of *A* is not zero. The mask is used to specify two different PVector operations to set the elements of PVector *B*. For those positions a_i of PVector *A* that are not equal to zero, the value of the corresponding element b_i of *B* is set to $1/a_i$. For the positions where a_i equals zero, b_i is set to zero. The `A.map(1/_)` operation is executed in the normal way, but only by those virtual processors where the mask has *true* value. Virtual processors where the mask has a *false* value will execute the `B.assign(Zero)` statement.

The individual statements executed for *true* and *false* in the above example may be replaced by a whole group of statements. Thus, this *Where Mask* feature creates a data parallel version of a general purpose *if* statement in ordinary code. The *Where Masks* may also be nested in an analogous way to the nesting of ordinary *if* statements.

V. Sample Parallel Program: Jacobi Relaxation

After describing the PVector class and its associated methods (operations), we can now present a sample data parallel Scala program for solving Laplace's Equation using the Jacobi Relaxation algorithm. Consider a two-dimensional (square) conducting metal sheet with the voltage held constant along the four boundaries. The resultant voltage $v(x,y)$ at all the internal points is described by Laplace's Equation in two dimensions:

$$\frac{\partial^2 v}{\partial x^2} + \frac{\partial^2 v}{\partial y^2} = 0$$

This equation can be solved numerically using a two-dimensional array of discrete points across the surface of the metal sheet. Initially, the points along the boundaries are assigned the appropriate constant voltage. The internal points are all set to 0 intially. Then Jacobi Relaxation is used to iteratively recompute the voltage at each internal point as the average of the four immediate neighboring points (above, below, left, right). Convergence is tested by comparing a desired tolerance value to the maximum change in voltage across the entire grid.

The basic data structure is a two-dimensional (*n* by *n*) array of *Double* values, representing the voltage at each point on the metal sheet. For data parallel execution, a PVector *A* is created, each of whose elements is a single row from the two-dimensional array. Thus, PVector *A* has *n* elements, each one of which is a one-dimensional array: `Array[Double](n)`. This data parallel PVector provides a virtual processor for each row of the original two-dimensional array. To recompute the value at each point, the four immediate neighboring points are needed. The left and right neighboring points are easy to find because they are in the same row, and therefore the same element of the PVector

```
def JacobiRelaxation(n: Int) = {
  val tolerance: Double = 0.0001
  PV.setNumThreads(4)
  ... // Initialize data array A (not shown)
  var Done = new PVector[Boolean](n+2)
  val In = Init.Index(n+2)
  val lShift = In + 1
  val rShift = In - 1
  val Mask = new PVector[Boolean](n+2,true)
  Mask.set(0,false)
  Mask.set(n+1, false)
  Where.begin(Mask)
  do {
   B = A.map(leftAndRight)
   B = A.permute(rShift).combine(arraySum,B)
   B = A.permute(lShift).combine(arraySum,B)
   B = B.map(divideByFour)
   Done = A.combine(getChange,B)
   done = Done.reduce(_&&_)
   A.assign(B)
 } while(!done)
 Where.end
 PV.endThreads()
}

def leftAndRight(a: Array[Double]) =
    { ... // details not shown }

def arraySum(a: Array[Double],
       b: Array[Double]) =
    { ... // details not shown }

def divideByFour(a: Array[Double]) =
    { ... // details not shown }

def getChange(a: Array[Double],
      b: Array[Double]) = {
    { ... // details not shown }
```

Figure 3. Jacobi Relaxation.

A. However, the neighboring points in the rows above and below are in neighboring elements of the PVector *A*. Access to these is implemented by shifting *A* left or right using the *permute* operation described in section IV.C. The data parallel Jacobi Relaxation algorithm in Scala is shown in Fig. 3.

The main body of the algorithm is the *do-while* loop in the *JacobiRelaxation* function body. Prior to the loop are the initializations which create the boolean *Where Mask* and the *lShift* and *rShift* PVectors to assist in the left-shift and right-shift permutations, respectively. During the looping, PVector *A* contains the initial value of the voltage at each point, and the new recomputed values are stored in PVector *B*. At the end of each iteration, the *assign* operation copies the values from *B* back to *A*, in preparation for the next iteration. In the first operation of the loop, the user-defined operation *LeftandRight*( ) is used to add the left and right neighboring values to each point. The *map* operation causes each virtual processor to apply the function *LeftandRight*( ) to the corresponding element of PVector *A*. The result is stored temporarily in vector *B*.

In the following instruction, the *permute* operation shifts *A* right and then uses *combine* to add the corresponding element of *B* in each virtual processor. This requires an additional user-defined operation *arraySum*( ), which is then

ISBN: 978-988-19251-4-5
ISSN: 2078-0958 (Print); ISSN: 2078-0966 (Online)

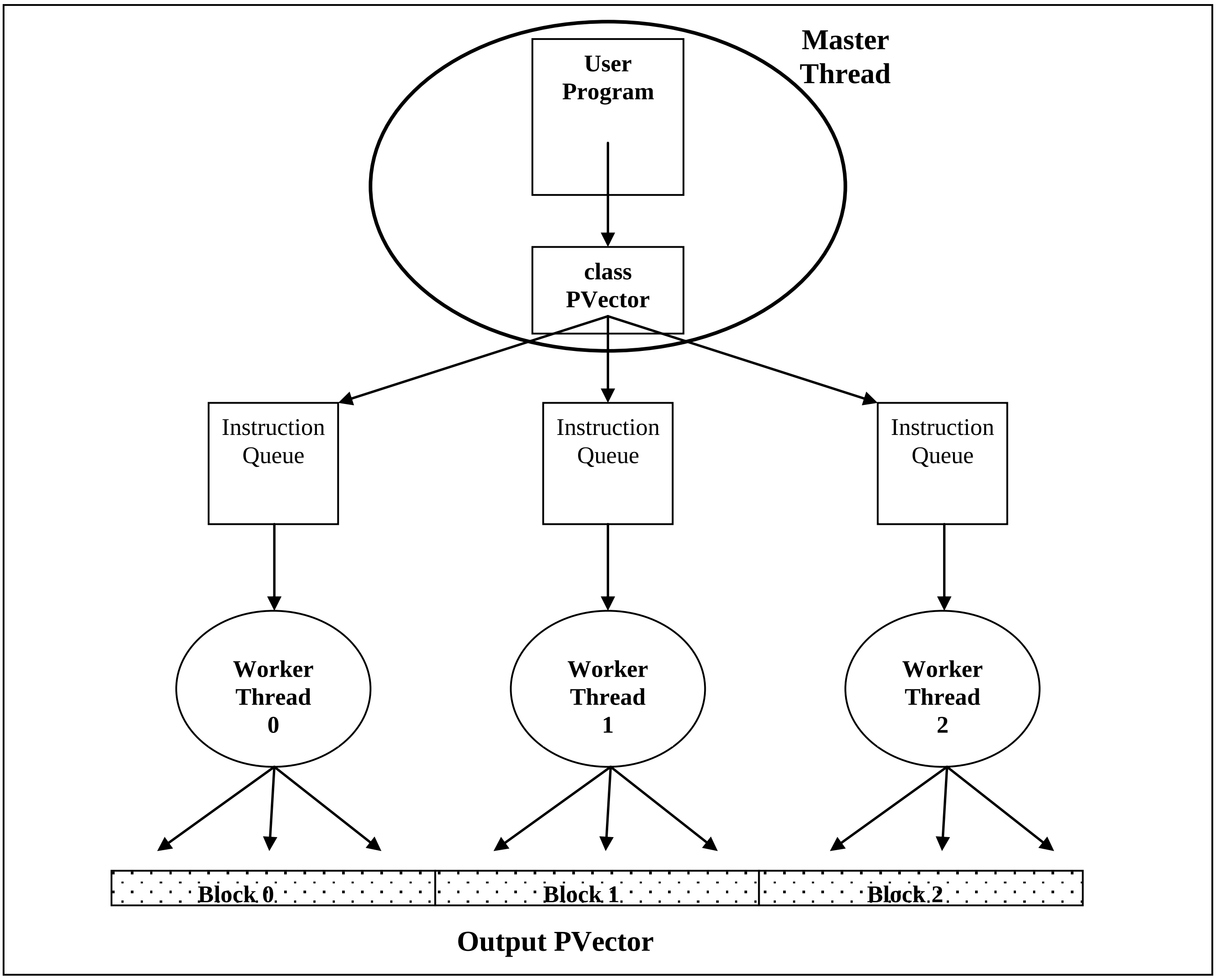


Figure 4. Parallel Implementation of PVector Operations.

used again to add the left-shift of *A* to *B*. Finally, the resultant sum of the neighbors is divided by four using the user-defined function *divideByFour*( ). This completes the calculation of the new value at each point as the average of the four immediate neighboring points. The user-defined operation *getChange*( ) determines if the change at each point is less than the desired tolerance. The result is a boolean PVector *Done* that is aggregated into a single boolean value done by the *reduce* operation.

Notice the use of the `Where.begin(Mask)` operation at the start of the *do-while* loop. This plays a key role in the correctness of the algorithm. Since the voltage at the boundary edges of the two-dimensional grid are held constant, the relaxation must only be applied to the internal points and not the boundaries. Element 0 of PVector *A* is the top row of the grid, and element *n*+1 is the bottom row. Both of these rows must be held constant as the internal points are modified by the relaxation. This is accomplished by setting *Mask*(0) and *Mask*(*n*+1) to false, so that all the PVector operations inside the *do-while* will not be applied to *A*(0) and *A*(*n*+1).

VI. Library Implementation

The basic structure of our implementation of PVectors in the library is illustrated in Fig. 4. The *User* Program is embedded in the Master Thread, along with the PVector class. Each collection-oriented library operation in the User Program will invoke a method in the class PVector. However, the actual computation to implement the operation is performed by the *Worker Threads* in parallel. Each Worker Thread has an *Instruction Queue* containing the sequence of operations it is to perform. The PVector class puts the instructions for the Workers into the Instruction Queues. Each Worker will have the same sequence of instructions in its Queue. We do not permit any out-of-order execution by the Workers.

The output data PVector is simply divided among the Worker Threads by using contiguous blocks as illustrated in Fig. 4. If the PVector has length 300, then the first block of 100 is assigned to Worker Thread 0, the next block of 100 to Worker Thread 1, and the last block of 100 to Worker Thread 2. Input PVectors are also partitioned into blocks in the same way. The total number of Worker Threads is determined by the user with the library function call *PV.setNumThreads*( ). Using this block allocation technique, it is completely predictable in advance which Worker Thread will be operating on each element of the PVector, based only on the length of the PVector and the total number of Workers.

The *operator fusion* is facilitated by the PVector class in

ISBN: 978-988-19251-4-5
ISSN: 2078-0958 (Print); ISSN: 2078-0966 (Online)

the following way: as soon as the PVector class receives a method invocation from the *User* Program, it allocates an empty PVector (with no data) to serve as the container for the output data from the Worker Threads, and returns an object reference to this PVector to the *User* Program. The User Program then continues executing even though the actual data to fill the output PVector has not yet been created by the Worker Threads. This implementation technique does not cause any errors because the User Program cannot directly access the data inside a PVector — it can only access the data indirectly by calling methods in the PVector class.

Since the collection-oriented parallel operations are fairly primitive in nature, the User Program will usually generate a long sequence of PVector operations. All of the real computational activity is done in the Worker Threads, which usually fall behind the User Program. Thus, the requested PVector operations will build up in the Instruction Queues. *Operator fusion* allows the Worker Threads to continue executing independently without having to synchronize with each other after each PVector operation.

However, depending on the nature and implementation of the specific sequence of requested PVector operations, it is sometimes necessary for some Thread synchronization. Sometimes the Worker Threads must execute a barrier synchronization, and sometimes the Master Thread must participate in this synchronization. Since our library implementation clearly has the *isolation* and *partitioning* properties described in section III, we can use the operator fusion algorithm of section IIIC to determine when a barrier (or strong barrier) is needed. The input data for this algorithm is the *OpInfoTab* shown in Table I with the basic properties of each of the fifteen operations in our collection-oriented library. Only the properties *InputCross*, *OutputCross*, and *HasBarrier* are shown because these are the most interesting and are implementation dependent. The other four properties follow obviously from the definition of each operation. The abundance of *False* values in this table shows that our implementation has ample opportunity for operator fusion.

To illustrate the principle of operator fusion, consider the following series of data parallel operations starting with data PVectors *A*, *B*, *C*:

```
T1 = A.map( _*2.0 )           // T1 = 2*A
T2 = T1.combine( _+_ , B )  // T2 = T1 + B
D = T2.combine( _/_ , C )   // D = T2/C
```

Assume three Worker Threads perform these operations by dividing the PVectors into blocks as described above. The Worker Threads could perform a synchronization barrier after each operation. However, this is not necessary because the intermediate results computed by the Workers do not cross the block boundaries. Worker 0 reads and writes only elements in block 0 of PVectors *A*, *B*, *C*, *D*, *T1*, *T2*. Similarly, Worker 1 reads and writes only elements in block 1 of all the PVectors. Worker 2 uses only elements in block 2. Therefore, there is no possibility of interference between the Workers: they read and write separate elements of the PVectors. Thus, the (*map*, *combine*, *combine*) sequence of data parallel operations could be **fused** within each Worker Thread without any intervening barriers. This operator fusion of data parallel operations greatly improves the performance.

TABLE I. PROPERTIES OF DATA PARALLEL OPERATIONS

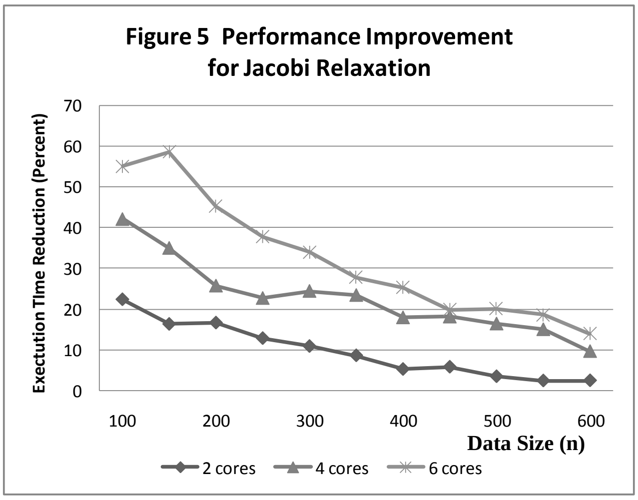
Operation	**Properties (True or False)**		
	Input Crossing	***Output Crossing***	***Has Barrier***
map	F	F	F
combine	F	F	F
reduce	F	F	**True**
scan	**True**	F	**True**
keyed-reduce	F	**True**	F
permute	**True**	F	F
select	F	**True**	F
broadcast	F	F	F
index	F	F	F
append	F	**True**	F
assign	F	F	F
list-input	F	F	F
read	F	F	F
get	F	F	F
set	F	F	F

As briefly explained in previous sections, the ability to do operator fusion of the data parallel PVector operations originates from the fact that all the Worker Threads are assigned to disjoint blocks of the PVectors. As long as the reading and writing of data values by each Worker remains within its own block, there is no possibility of any timing-dependent errors, and the Workers can just proceed independently at their own relative speeds. However, among the fifteen operations in our data parallel library, there are some operations that do require the Workers to cross block boundaries and either read or write an element in a block assigned to another Worker. This is indicated by a *True* value in the *Input Crossing* or *Output Crossing* columns of Table I. However, the vast majority of operations do not have crossings, resulting in ample opportunity for operator fusion.

VII. PERFORMANCE BENCHMARKS

To measure the performance improvement resulting from operator fusion, we created a version of our collection-oriented parallel library with no operator fusion: all of the Worker Threads execute a barrier synchronization after each operation, and the User Program waits for completion of the operation before executing the next instruction. In previous sections, we have called this a *strong barrier*. For three benchmark parallel programs, we determined the execution time using the two different versions of our library (one without operator fusion and one with operator fusion). The computer used for performance testing is a Dell Studio XPS 7100 Minitower with 8 GB of memory and a 6-core processor (AMD Phenom II X6 1035T).

ISBN: 978-988-19251-4-5
ISSN: 2078-0958 (Print); ISSN: 2078-0966 (Online)



For the Jacobi Relaxation program described in section V, Fig. 5 shows the performance improvement resulting from operator fusion for a range of data sizes and varying numbers of cores. The vertical axis shows the percentage reduction in the overall program execution time when the operator fusion optimization is turned on. The horizontal axis shows the number of elements (n) in the PVector. Recall that each element of the PVector is an array with n elelments, so the total data size is proportional to n^2. We see that operator fusion provides quite a significant performance improvement especially for smaller data sizes. Also as expected, the performance improvement is greater for larger numbers of cores, because the barrier execution time increases with the number of cores.

Now let us do a more general analysis of the expected performance improvement from operator fusion. The execution of each collection-oriented parallel operation in our library has three basic phases: *Setup*, *Operation Execution*, and *Barrier*. Fig. 4 shows the *Setup Phase* consisting of the original library function call in the User Program and the initial processing of this call in the PVector class. The *Operation Execution* Phase is the parallel execution of the PVector operation by the Worker Threads. Finally, the *Barrier Phase* synchronizes the Worker Threads. Operator fusion removes the need for this Barrier Phase. If n is the PVector size, and p is the number of Worker Threads (number of cores), then the expected execution time is as follows:

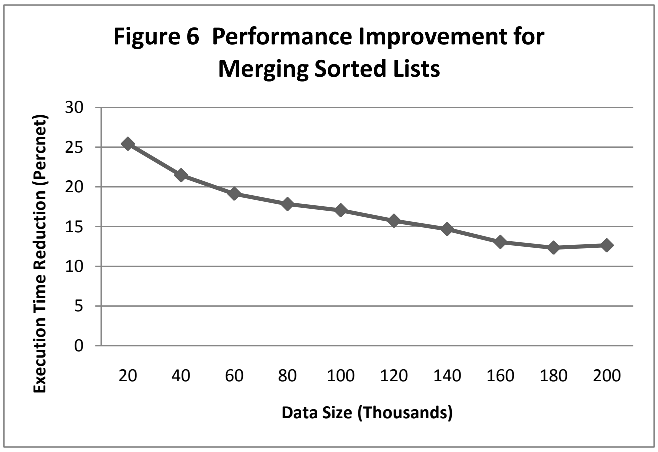
Setup: $\mathbf{O}(p)$; Operation Execution: $\mathbf{O}(n/p)$; Barrier: $\mathbf{O}(p)$;

If the p is very large, then the Setup and Barrier can be reduced to $\mathbf{O}(\log p)$. This analysis shows that decreasing the data size n will increase the relative impact of operator fusion on improving performance. This is clearly seen in the graphs of Fig. 5. Similarly, increasing the number of cores p will also increase the performance gain from operator fusion (also seen in Fig. 5). Even for a six-core processor we see a significant performance improvement from operator fusion of up to 60%. This general analysis indicates even more significant improvement is expected as the number of cores is increased. Therefore, as multi-core processor technology continues to evolve, operator fusion will become increasingly important as powerful technique for improving the performance of data parallel operations.

To further investigate the impact of operator fusion, we considered an additional benchmark program to Merge Two Sorted Lists X and Y into a single sorted list Z. A simple algorithm that is easily parallelized is as follows: for each x_i in list X, do a binary search of list Y to determine the position j where x_i should be inserted in list Y to preserve the ordering. Then the final position of x_i in the output list Z should be $Z[i+j]$. To create a parallel version of this algorithm using our collection-oriented library, X, Y, and Z are represented as PVectors. The binary search of Y by all the elements in X is done in parallel using the PVector library operations. Similarly, to find the proper location for the elements of Y, each y_i is used to do a binary search of list X (all y_i in parallel).

We executed this Merging Sorted Lists program for a range of data sizes using the two versions of our library, with and without operator fusion. The results for six cores are shown in Fig. 6. The vertical axis shows the percentage reduction in the overall program execution time when the operator fusion optimization is turned on. The horizontal axis shows the number of elements in list X (and Y). There is a significant performance improvement of up to 25%. However, this is much less than the 60% improvement for the six-core Jacobi Relaxation program. The performance improvement in a particular program just depends on how many data parallel library operations can be fused in that program.

To determine the maximum possible improvement from operator fusion, we considered one additional "best case" program: a simple iterative loop with a series of thirty PVector *map* and *combine* operations doing basic arithmetic on scalar floating point numbers. We call this the *Map Benchmark* program. As seen in Table I, the *map* and *combine* operations always can be fused. So the program requires no barriers. The performance improvement resulting from operator fusion is shown in Fig. 7. The horizontal axis is the PVector size. The graph shows a reduction in overall execution time of up to 82% by using the operator fusion optimization only.



ISBN: 978-988-19251-4-5
ISSN: 2078-0958 (Print); ISSN: 2078-0966 (Online)

REFERENCES

[1] J. C. Adams, W. S. Brainerd, J. T. Martin, T. Smith, and J. L. Wagener, Fortran 90 Handbook. McGraw-Hill, 1992.

[2] High Performance Fortran Forum. High Performance Fortran specification version 2.0, January 1997.

[3] C. Koelbel and P. Mehrotra, "An overview of high performance Fortran," SIGPLAN Fortran Forum, vol. 11, No. 4, pp. 9–16, 1992.

[4] J. Dean and S. Ghemawat, "MapReduce: Simplified data processing on large clusters," Commun. ACM, vol. 51, no. 1, pp.107–113, 2008.

[5] W. D. Hillis, The Connection Machine. MIT Press series in artificial intelligence, 1985.

[6] G. E. Blelloch, Vector Models for Data-Parallel Computing. The MIT Press, Cambridge, Massachusetts, 1990.

[7] K. E. Iverson, A Programming Language. John Wiley & Sons, Inc., New York, NY, 1962.

[8] A. Ghuloum, E. Sprangle, and J. Fang, "Flexible parallel programming for terascale architectures with Ct," Intel White Paper, Intel Corporation, 2007.

[9] M. D. McCool, "Data-Parallel programming on the Cell BE and the GPU using the RapidMind Development Platform," GSPx Multicore Applications Conference, Santa Clara, CA, November 2006.

[10] Array Building Blocks Application Programming Interface Reference Manual, Intel Corporation, 2011.

[11] I. Buck, T. Foley, D. Horn, J. Sugerman, K. Fatahalian, M. Houston, and P. Hanrahan, "Brook for GPUs: Stream computing on graphics hardware," ACM Transactions on Graphics, Volume 23, Issue 3, pp. 777-786, 2004.

[12] P. Charles, et. al., "X10: An object-oriented approach to non-uniform cluster computing," 20th ACM SIGPLAN OOPSLA Conference, San Diego, CA, pp. 519 – 538, 2005.

[13] B. Carpenter and G., "HPJava: A data parallel programming alternative," Computing in Science and Engineering, vol. 5, no. 3, pp. 60-64, May/June 2003.

[14] D. Wampler and A. Payne, Programming Scala, Chapter 12, O'Reilly, 2008.