Abstract

To more fully utilize the potential offered by multi-core processors, programming languages must have features for expressing parallelism. Data parallel programming features allow high-level collection-oriented operations to be easily expressed by the programmer, and then implemented by the runtime system in a parallel fashion for improved performance. This paper describes a data parallel class library, for the Scala language. This data parallel library includes a new type of data structure called a Parallel Vector, and fifteen basic operations on this data structure. These operations are implemented within the library using ordinary multi-threading. However, this implementation is hidden from the programmer. The expressive power of many of these data parallel operations is enhanced by allowing the programmer to supply user-defined functions, which are utilized during the data parallel operations. The library and its implementation in Scala have several novel aspects including Where-Masks for conditional execution and vertical integration of data parallel operations in the runtime library implementation.

Categories and Subject Descriptors D.1.3 Concurrent Programming

General Terms Performance, Languages.

Keywords Scala; data parallel; parallel programming; multi-core processor; vertical integration

1. Introduction

During the past few years, parallel computing has become mainstream. All the new processors have multiple cores, and therefore, all the new computers are essentially parallel computers. With the number of cores per processor expected to double every 2-3 years, this offers a potential for enormous performance improvement. However, for this potential to be fully realized, techniques for exposing parallelism in computer programs are needed. Experience has shown that parallelizing compilers can find some parallelism in ordinary sequential computer programs. However, this approach is limited because of the sequential constraints found in these programs. To more fully expose the potential parallelism in programs, the programmer must rethink the algorithm at a more abstract level, and then write a parallel version of the program. This conversion from sequential to parallel requires the creativity and intelligence of a human programmer. It can not be effectively accomplished automatically by a compiler alone.

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 data parallel programming, which has an extensive history spanning several decades. The essence of data parallel programming is high-level collection-oriented operations, in which every element of the collection is operated upon in parallel by the same operation. 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 data parallel 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 data 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 landmark book by Guy Bliehochen, _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 data parallel 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 C.t [8], a language under development by Intel for their experimental tera-scale processor architectures. The company RapidMind has successfully marketed a data parallel extension of the C++ language with collective operations on arrays [9]. Researchers at Stanford University have developed a data parallel extension of the C language called Brook [10], 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 [11] and HPJava [12] are both data parallel versions of Java, intended for scientific and engineering applications on high-performance computer clusters.

2. Scala Library

This paper is a progress report on a continuing research project to develop a data parallel version of the Scala language. The first phase of this research has been to implement data parallelism in Scala completely with new library classes. The parallel data structures and the allowable operations on them are implemented in a self-sufficient class library that can be used by any standard Scala program. We have a working version of this library that we are using for research purposes. This paper contains a detailed description of this library, including a sample data parallel Scala program, and the results of performance testing of this program on a computer with a multi-core processor. The paper also contains details on the implementation of the data parallel operations.

The data parallel library we present here is similar to other data parallel libraries for other languages, and contains many of the standard operations found in other libraries. However, there are two novel aspects of our data parallel library. Firstly, we have greatly expanded the range of algorithms that can be easily represented in data parallel form, by introducing a new “Where” class that allows selective execution of parallel operations. Also, our library uses a novel implementation of the data parallel operations called _vertical integration_, which removes the necessity of an implied barrier operation after each data parallel operation, and therefore significantly improves performance.

3. Parallel Vectors

The basic data parallel object we use 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]. For example, _PVector[Double]_ creates a Parallel Vector, all of whose elements have type _Double_. Similarly, _PVector[String]_ creates a Parallel Vector, each of whose elements is a _String_. If the Scala program contains a class called _Employee_ with data fields and methods related to maintaining information about employees of a company, then the program could use _PVector[Employee]_ to create a Parallel Vector whose individual elements are _Employee_ objects.

This very general definition of Parallel Vector as composed of any type of object is easy to do in Scala using the sophisticated type system and generic capability of the language [13]. This is a deviation and improvement over many of the data parallel implementations done by others, which allow only primitive data types, such as integer, float, and boolean, to be the component type of data parallel structures (see discussion of previous research above). This capability of _PVectors_ to consist of arbitrary user-defined object types creates a very powerful and general parallel programming vehicle. Other data parallel languages have sometimes allowed nesting in the data parallel structures. For example, the TVEC of the language C.t [8] has two levels of nesting. This is necessary because the component type of a TVEC must be selected from a small set of primitive scalar types. However, the elements of _PVectors_ in Scala may be arbitrarily complex data structures. Therefore, multiple levels of nesting is not necessary. For example, _PVector[Array[Double]]_ defines a Parallel Vector each of whose elements is an array of _Double_.

The _PVector_ class in our data parallel Scala library provides several constructors for creating new _PVector_ objects, including the following:

- _PVector(n: Int) _ constructs an empty _PVector_ with initial capacity _n_.
- _PVector(aList: List[T]) _ constructs a _PVector_ from the elements of the specified list.
- _PVector(n: Int, value: T) _ constructs a _PVector_ composed of _n_ elements of type _T_, each with the same specified value.

For example, the following creates a new object _Mask_ consisting of a _PVector_ with one thousand elements, each of which is the _boolean_ value _true_:

```
Mask = new PVector[Boolean](1000, true)
```

The following Scala statements create a new _PVector_ _A_ with _n_ elements, each of which is an array of _n_ zeroes:

```
blank = new Array[Double](n)
```
for {i <- 0 to n-1} blank(i) = 0.0
A = new PVector[Array[Double]](n,blank)

4. Operations on Parallel Vectors

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.

4.1 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. The signature of the map method is as follows:

\[ \text{map}[U](\text{unaryop}: (\text{T}) \Rightarrow \text{U}): \text{PVector}[\text{U}] \]

The PVector that invokes the map method becomes the input for the operation and has generic element type T. The resultant output PVector after applying the user-defined function unaryop has generic element type U. Again we see the power of generic types in the Scala language to define a very general and powerful operation on PVectors. Consider a PVector[Boolean] called Mask (see definition in previous section). The map method can be invoked as follows to create a new PVector whose elements are the logical negation of Mask:

\[ \text{B} = \text{Mask.map}(\_\_\_) \]

Notice how simply and compactly this can be expressed in the Scala language. The notation “\_” represents an anonymous function with one parameter whose output is the logical negation of the input. One of the reasons we have chosen Scala as the language for implementing our data parallel library is the ease of dealing with user-defined functions, which play an important role in data parallel programming.

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 two input PVectors must have the same length.

\[ \text{combine}[U](\text{op}: (\text{T},\text{T}) \Rightarrow \text{U}, \text{bVec}: \text{PVector}[\text{T}]): \text{PVector}[\text{U}] \]

Assume PVectors A and B both have the same length and component type Int. The combine method can be invoked to create a new PVector from the sum of the corresponding elements of A and B:

\[ \text{C} = \text{A}.\text{combine}[\text{Int}](\_\_\_\_\_\_\, \text{B}) \]

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.

Since PVectors may have an arbitrary element type, the map and combine operations may be used to apply very powerful high-level user-defined operations to PVectors. For example, the Jacobi Relaxation algorithm described in a subsequent section of this paper uses PVectors with element type Array[Double], i.e. each element of the PVector is itself an array of Double. One operation needed in this algorithm is to replace each element in each Double array by the sum of its left and right neighboring values, except for the end points of the array, which remain unchanged. This is accomplished with the combine operation and the following user-defined function:

\[
\text{def leftAndRight}(a: \text{Array}[\text{Double}]) = \{
\text{var m} = a.\text{length}
\text{var b} = \text{new Array}[\text{Double}](\text{m})
\text{for } (i < 1 \text{ to } \text{m}-2) \{
\text{b}(i) = a(i-1) + a(i+1)
\}
\text{b}(0) = a(0); b(m-1) = a(m-1)
\}
\]

In the Scala language, functions are objects and are therefore easily passed as parameters to the map and combine operations. The syntax and implementation of these operations is considerably simpler in Scala than in other languages such as Java, in which functions must be embedded in other objects.

4.2 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. When used for Reduce operations, there is an additional restriction that the user-defined operation must be associative (this allows an efficient parallel implementation). Three basic operations are reduce, scan, and keyed-reduce. The operation called reduce has the following signature:

\[ \text{reduce}(\text{binop}: (\text{T},\text{T}) \Rightarrow \text{T}): \text{T} \]

The reduce method is contained in the PVector class. The specific PVector object that invokes the reduce method is the input vector for the reduction operation. The following reduce operation sums the elements of the PVector A:

\[ \text{A} = \text{new PVector[Int]}(\text{aListofIntegers})
\text{result} = \text{A}.\text{reduce}(\_\_\_\_\_\_\_) \]

This kind of parallel reduction operation is quite commonly used in parallel algorithms to aggregate data. For example, the Jacobi Relaxation program described later in section 6 uses a reduce operation to do a convergence test at the end of each iteration.

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. The signature of scan is as follows:

$$\text{scan}(\text{binop}: (T,T) \Rightarrow T): \text{PVector}[T]$$

The scan operation turns out to be useful in a wide range of parallel algorithms, as described in detail by Blelloch [6]. 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. If the Index does not map any values to a specific location of the Target vector, then this location defaults to its original value from the Target vector.

As with the reduce and scan operation, keyedReduce is a method in the PVector class, and the input Data vector is the one that invokes the keyedReduce method:

$$\text{keyedReduce}(\text{Index}: \text{PVector}[\text{Int}], \text{Target}: \text{PVector}[T], \text{binop}: (T,T) \Rightarrow T): \text{PVector}[T]$$

### 4.3 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]:

$$\text{permute}(\text{Index}: \text{PVector}[\text{Int}]): \text{PVector}[T]$$

The parameter Index selects specific elements from the input Data vector to be placed into the Output vector, as illustrated in the following simple example:

- Data: [ 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.

### 4.4 Initialize Operations

The Initialize operations allow new PVectors to be created with initial data. One of the PVector constructors already described in section 3 called a Broadcast operation, can be considered as a member of this class of operations:

$$\text{PVector}(n: \text{Int}, \text{value: } T)$$

The other Initialize operations are methods in the PVector class. The Index operation creates a PVector of length $n$ with element values $0, 1, 2, ..., n-1$:

$$\text{Index}(n: \text{Int}): \text{PVector}[\text{Int}]$$

The append operation creates a new PVector from the concatenation of two existing PVectors – the PVector that calls the append method and the PVector specified by the parameter aVec:

$$\text{append}(\text{aVec}: \text{PVector}[T]): \text{PVector}[T]$$

The assign operation copies a source PVector into the destination PVector, which is the one that calls the assign method:

$$\text{assign}(\text{source: PVector}[T]): \text{PVector}[T]$$

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.

The assign operation is unusual among the data parallel operations in our library, in that it modifies an existing PVector object. The only other operation that modifies an existing PVector object is keyed-reduce, which modifies the individual elements of the Target PVector. The other data parallel operation do not modify any already existing PVector object. For example, the map and combine operations create a new PVector object, as illustrated in the following example instruction:

$$C = A.\text{combine}[\text{Int}](\_\_\_+, B)$$

This instruction creates a new PVector object by adding the corresponding elements of PVectors A and B. A reference to this new PVector object is then written into variable C.

### 4.5 Input/Output Operations

The Input/Output operations allow external data values to be pushed into a PVector or extracted from a PVector. One such operation is the PVector constructor that creates a new PVector from the elements of the specified List parameter:
The read operation works in the opposite direction by copying data from the PVector into a specified List:

```scala
read( ): List[T]
```

The get and set operations allow individual data values to be extracted from a specific position of the PVector, or inserted into the PVector, respectively.

Following is a summary of the fifteen primitive operations on PVectors implemented in our data parallel Scala Library:

<table>
<thead>
<tr>
<th>Category</th>
<th>Operations</th>
</tr>
</thead>
<tbody>
<tr>
<td>Map</td>
<td>Map, Combine</td>
</tr>
<tr>
<td>Reduce</td>
<td>Reduce, Scan, Keyed-Reduce</td>
</tr>
<tr>
<td>Permute</td>
<td>Permute, Select</td>
</tr>
<tr>
<td>Initialize</td>
<td>Broadcast, Index, Append, Assign</td>
</tr>
<tr>
<td>Input/Output</td>
<td>List-Input, Read, Get, Set</td>
</tr>
</tbody>
</table>

5. 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 $1/a_i$:

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

The PVectors within the scope of a Where mask must conform to the mask, which in most cases means they must have the same number of elements as the mask. The one exception is keyed-reduce, for which the Target PVector may have a different length than the mask because the masking applies only to the input Data and Index PVectors. The select and append operations are not permitted within the scope of a Where Mask.

Where Masks can also be used to create a data parallel version of looping. In many parallel numerical programs, a while loop iteratively repeats a series of PVector operations until a convergence criterion is achieved. In this case, every virtual processor is engaged in every loop iteration. Now consider a different type of algorithm in which the convergence test is local at each point, so that some virtual processors should stop looping while others continue. This can be accomplished using a Loop Mask with a true value at every position where looping is to continue, and a false value at positions where computation should cease. This allows some virtual processors to be idle, while others continue to execute the loop. The following illustrates the overall structure of the code required to accomplish this using our data parallel Scala library:

```scala
PVector[Boolean] loopMask = ... // Initialize
while(Where.any(loopMask)) {
  ... // series of PVector operations
  loopMask = ... // recompute loop_mask
  Where.end;
}
```

In the above, the loopMask is recomputed each time around the loop. The number of true values in the mask gradually decrease, causing more virtual processors to cease executing the loop. Eventually, the loopMask will be all false values, at which time the entire while loop terminates. This is accomplished with the Where.any method that computes the logical and of the elements in the loopMask. The PVector operations inside the loop body will be executed in the normal way, but only on those elements where the corresponding loopMask element is true.

Fortran 90 [1] does have a Where construct to accompany the array operations, but it is more limited than our Where class. In Fortran 90, the Where may not be nested, and there is no provision for looping using the Where. Thus, our Scala data parallel library expands the utility and applicability of the Where construct, so that a wider range of parallel programs can be easily expressed. Also, in Fortran 90, the compiler is involved in the implementation of the Where construct. We have done it completely with a library, requiring no change to the Scala compiler.

6. 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 volt-
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 initially. 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 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 4.3. The data parallel Jacobi Relaxation algorithm in Scala is shown in Figure 1.

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 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

Figure 1. Jacobi Relaxation
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).

7. Library Implementation

Although our Scala data parallel library is similar to data parallel libraries of other languages, our implementation technique is innovative and introduces some important new ideas, the most notable one being vertical integration to improve performance. The usual way to implement any library function call in Scala or any other language, is to complete the function fully and then return to the caller. In the case of a data parallel operation, this means that the participating parallel threads must all synchronize with each other after they complete their assigned portion of the operation. This synchronization, usually called a barrier, is very time consuming, and introduces a large execution time overhead if it is performed after every data parallel operation.

This problem has been noted by other researchers. The research language Ct [8] under development by Intel uses a technique called vertical integration in the compiler, instead of requiring a barrier after every data parallel operation, successive data parallel operations within each thread are done together without any intervening barriers. However, in the Ct language this vertical integration requires the use of a special compiler with complete knowledge of the data parallel operations. Our novel contribution with the use of a special compiler with complete knowledge of the data parallel operations greatly improves the performance. This synchronization, usually called a barrier, is very time consuming, and introduces a large execution time overhead if it is performed after every data parallel operation.

As illustrated in Figure 2, each Worker has its own instruction queue to receive the sequence of required operations from the class PVector. Each data parallel operation in the User Program requires invocation of a method in the class PVector, which in turn will encode the requested data parallel operation into an instruction, and write this instruction into the queue of each Worker thread. When there is a series of data parallel operations generated from the User Program, the instructions will build up in the Worker instruction queues.

Our implementation uses a very simple form for the instructions with a numeric opcode field and five additional fields for arguments:

<table>
<thead>
<tr>
<th>Instruction Field</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Int</td>
</tr>
<tr>
<td>2</td>
<td>PVector</td>
</tr>
<tr>
<td>3</td>
<td>PVector</td>
</tr>
<tr>
<td>4</td>
<td>function of one parameter</td>
</tr>
<tr>
<td>5</td>
<td>function of two parameters</td>
</tr>
</tbody>
</table>

Each of the fifteen data parallel operations currently included in the library has a specific numeric opcode ranging from 1 to 15, which is placed in Instruction Field 1. To produce an output, the data parallel operations do not modify the input PVector, but rather create a new PVector and return it to the caller. The initial allocation of the output PVector is done in the class PVector. Then a re-

\[
\begin{align*}
\text{T1} &= \text{A.map(} \_*2.0 \text{)} \quad \text{// } \text{T1} = 2\times\text{A} \\
\text{T2} &= \text{T1.combine(} +_\text{, } \text{B) } \quad \text{// } \text{T2} = \text{T1} + \text{B} \\
\text{D} &= \text{T2.combine(} /_\text{, } \text{C) } \quad \text{// } \text{D} = \text{T2/C}
\end{align*}
\]

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 vertically integrated within each Worker Thread without any intervening barriers. This vertical integration of data parallel operations greatly improves the performance.

7.1 Instruction Queue

As illustrated in Figure 2, each Worker has its own instruction queue to receive the sequence of required operations from the class PVector. Each data parallel operation in the User Program requires invocation of a method in the class PVector, which in turn will encode the requested data parallel operation into an instruction, and write this instruction into the queue of each Worker thread. When there is a series of data parallel operations generated from the User Program, the instructions will build up in the Worker instruction queues.

Our implementation uses a very simple form for the instructions with a numeric opcode field and five additional fields for arguments:

<table>
<thead>
<tr>
<th>Instruction Field</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Int</td>
</tr>
<tr>
<td>2</td>
<td>PVector</td>
</tr>
<tr>
<td>3</td>
<td>PVector</td>
</tr>
<tr>
<td>4</td>
<td>function of one parameter</td>
</tr>
<tr>
<td>5</td>
<td>function of two parameters</td>
</tr>
</tbody>
</table>

Each of the fifteen data parallel operations currently included in the library has a specific numeric opcode ranging from 1 to 15, which is placed in Instruction Field 1. To produce an output, the data parallel operations do not modify the input PVector, but rather create a new PVector and return it to the caller. The initial allocation of the output PVector is done in the class PVector. Then a re-
ference to the empty output `PVector` is passed in Field 2 to the Workers, which write the data into it. Field 3 is used for a reference to the input `PVector`. Some data parallel operations, such as `keyed-reduce`, also require a reference to a third `PVector`, which is placed in Field 4. Fields 5 and 6 are used for references to user-defined functions supplied as parameters to many of the data parallel operations, such as `reduce` and `combine`.

These simple instructions represent a kind of data parallel machine language with fifteen opcodes. Each instruction is conveniently represented in Scala as a `tuple`. The instruction queue of each Worker Thread is implemented by the library class `LinkedBlockingQueue()` as found in the library `java.util.concurrent`. The writing of the queue from the Master Thread and the reading of the queue from the Worker Thread is done in parallel. Therefore, proper synchronization is needed to make the queue work properly. This is all handled within the library code of `LinkedBlockingQueue()`.

![Parallel Implementation of PVector Operations](image)

**Figure 2.** Parallel Implementation of PVector Operations
With this implementation, each Worker Thread is free to execute its instructions at its own speed, working on its assigned block of the PVector. Thus, the required vertical integration of the data parallel operations results naturally from this implementation. However, some of the operations in our library may cause Worker Threads to cross block boundaries into portions of the PVector assigned to other Workers. For these operations, a barrier synchronization among the Workers may be required either before or after the operation.

7.2 Barrier Synchronization

As briefly explained in the previous section, the ability to do vertical integration of the data parallel PVector operations originates from the fact that 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.

Consider a simple example of the following sequence of data parallel operations in a User program:

\[
T_1 = A\text{.map}(x \rightarrow x*2.0) \quad // \quad T_1 = 2*A \\
T_2 = T_1\text{.permute(Index)} \quad // \quad \text{permute} T_1
\]

The first operation creates a new PVector \( T_1 \) by multiplying every element of PVector \( A \) by 2.0. Then the second operation permutes (reorders) the elements of \( T_1 \) to create a new PVector \( T_2 \). This is illustrated in Figure 3. Assume there are three Worker Threads and the PVector size is 300. As shown in the Figure, each Worker is assigned a distinct block of 100 elements from PVector \( A \) and \( T_1 \). Therefore, the multiplication by 2.0 can proceed independently in each Worker.

The next step for each Worker is to perform the \textit{permute} operation using the PVector PVec- tor \text{Index}. The Index specifies which element of PVector \( T_1 \) will be copied into the output PVector \( T_2 \) (see discussion of \textit{permute} in section 4.3). In Figure 3, only one specific element of Index is shown: a value of 150 in the block assigned to Worker 0. This will select the value 216.6 from element 150 of PVector \( T_1 \) and write it into PVector \( T_2 \), as illustrated in the Figure. However, this value 216.6 must be written into \( T_1 \) by Worker 1 during the previous data parallel operation (multiply \( A \) by 2.0).

If Worker 0 is faster than Worker 1, it may try to retrieve element 150 from \( T_1 \) before Worker 1 writes, causing the old value to retrieved. To prevent this type of data race error, the first operation (multiply \( A \) by 2.0) must be completed before the \textit{permute} operation begins. Thus, a barrier synchronization of all the Worker Threads is required between the operations, as shown in Figure 3.

Any data parallel operation that may cause a Worker to cross block boundaries, requires some kind of barrier synchronization of the Workers. If the boundary crossing occurs in one of the input PVectors to an operation, then a barrier is required before the operation begins, as is the case with the \textit{permute} operation illustrated in Figure 3. If the boundary crossing occurs in the output PVector of an operation, then a barrier is required at the end of the operation.

It is necessary to carefully examine the detailed implementation of each data parallel operation to see where barriers are required. For each of the fifteen data parallel operations in our Scala library, the following table summarizes whether block boundary crossing occurs on the input and/or output PVector.

<table>
<thead>
<tr>
<th>Operation</th>
<th>Input Block Crossing</th>
<th>Output Block Crossing</th>
</tr>
</thead>
<tbody>
<tr>
<td>map</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>combine</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>reduce</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>scan</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>keyed-reduce</td>
<td>no</td>
<td>yes</td>
</tr>
<tr>
<td>permute</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>select</td>
<td>no</td>
<td>yes</td>
</tr>
<tr>
<td>broadcast</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>index</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>append</td>
<td>no</td>
<td>yes</td>
</tr>
<tr>
<td>assign</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>list-input</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>read</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>get</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>set</td>
<td>no</td>
<td>no</td>
</tr>
</tbody>
</table>

7.3 Master Thread Synchronization

One additional consideration is synchronization between the Master and Worker Threads. The Master Thread contains the User program and class PVector (see Figure 2). When an operation is requested by the User program, the class PVector sends an instruction to each Worker Thread, then returns to the User program before the operation is actually completed by the Workers. The User program continues to execute and may request additional data parallel operations. Thus, the User program can progress far ahead of the actual implementation of the data parallel operations by the Workers.

The question then arises: are there any circumstances under which the User program must wait for the Workers to complete an operation? As long as the output of the data parallel operation goes into another PVector, then the User program does not have to wait for completion. However, if the result of an operation comes out of the PVector space and into an ordinary program variable, then the User program must wait for completion of the operation before moving on execution of the next program instruction.
For example, consider the following instructions found at the end of the do-while loop in the Jacobi Relaxation program shown in Figure 1:

```scala
done = Done.reduce(_&&_)
} while(!done)
```

The Boolean PVector `Done` is reduced using the logical `and` operation to a single Boolean value that is assigned to Boolean variable `done`, which is used to determine whether to do another loop iteration. Clearly, the User program must wait until this data parallel reduction operation is complete before moving on the next program instruction (the `while` instruction). This is a situation where the data from a PVector is coming out into an ordinary program variable that can be used for something other than the fifteen data parallel library operations. Therefore, the User program must wait for the result.

Two other data parallel operations (besides `reduce`) also have this requirement: `read` and `get`. Operation `read` copies a whole PVector into an ordinary List variable. The operations `reduce` and `read` require a synchronization between the User program and the Worker Threads at the completion of the operation by the Workers. This is accomplished in our library implementation by a barrier that includes the Master Thread (see Figure 2) and all of the Worker Threads. Operation `get` copies a single element from a PVector to an ordinary variable, so only the Worker that has this element in its assigned block of the PVector must synchronize with the Master Thread – no global barrier among all the Workers is needed.

As seen in the table of operations above, five of the fifteen data parallel operations requires a barrier either before or after the operation. Three additional operations require a barrier with the Master Thread. The remaining seven operations require no barrier at all. These barriers do restrict the potential vertical integration somewhat, but most programs still have ample opportunity for vertical integration of the data parallel operations to improve performance.

### 8. Library Performance

We have not yet done extensive performance testing of our data parallel Scala library. However, preliminary tests are very encouraging and show significant speedups even us-
ing ordinary desktop business computers. In this section we will describe the detailed performance results for the Jacobi Relaxation program presented in section 6.

The Jacobi Relaxation has an input parameter \( n \) which determines the data size. Recall that the data structure for Jacobi Relaxation is an \( n \) by \( n \) array of double precision floating point values. Each row of the array becomes a separate element in the \( P\text{Vector} \), which therefore has a total of \( n \) elements. The User program also has control over the number of threads by invoking the library function \( \text{PV.setNumThreads()} \).

Figure 4 shows the raw performance data for three different data sizes: \( n = 500, 1000, 1500 \). Remember that the total data size is proportional to \( n^2 \). For each of the three values of \( n \), the number of Worker Threads used by the data parallel library implementation was varied from 1 to 4. Tolerance .01 is used for convergence. The computer used for testing has an Intel i7 Quad-Core processor (Intel DX58SO Motherboard) and 8 GB memory. Since the processor only has four cores, we would not expect any performance improvements by adding additional threads beyond four.

The raw performance data in Figure 4 does show significant performance improvements as the number of threads is increased from 1 to 4. One thread means that there is essentially no parallel computing – a single Worker Thread works on the whole \( P\text{Vector} \) in a sequential manner. To act as a baseline for evaluating the parallel performance characteristics, we also created a completely sequential implementation of the library functions with no Worker Threads at all: each library function is implemented as a simple sequential \( for \) loop in the \( P\text{Vector class} \). This sequential version execution time is then divided by the parallel execution time to compute the speedup. The following table shows the speedup achieved on the Jacobi Relaxation program for four different data sizes:

<table>
<thead>
<tr>
<th>Threads</th>
<th>100</th>
<th>500</th>
<th>1000</th>
<th>1500</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>1.4</td>
<td>1.5</td>
<td>1.6</td>
<td>1.5</td>
</tr>
<tr>
<td>3</td>
<td>1.5</td>
<td>1.8</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>1.6</td>
<td>2.1</td>
<td>2.4</td>
<td>2.2</td>
</tr>
</tbody>
</table>
9. Future Research

Our goal in this research project was to create a self-sufficient Scala class library that allows data parallel programming in Scala. In this paper, we present an overview of the Parallel Vector (PVector) data structure used in this class library, and the fifteen fundamental data parallel operations on PVectors included in the library. A data parallel program for Jacobi Relaxation is presented along with performance results for a quad-core processor. The innovative Where-Mask feature of our library is also briefly described. This is not found in other data parallel libraries.

A novel implementation technique is also described, which allows vertical integration of the data parallel operations with no intervening barrier synchronization. To our knowledge, this is the first time this technique has ever been used in a library. In our future research, we plan to further develop this technique, including some mathematical analysis to understand its properties more completely. We would like to develop and mathematically verify a set of simple rules to determine when and where barrier synchronization is required between data parallel operations.

One important issue for future research is determinacy. The introduction of the paper discussed one of the fundamental problems of using standard multi-threading to write parallel programs: data races on shared data may lead to a nondeterministic output of the program. Data parallel programming using a library like the one we have developed can solve this problem, provided that the user-defined functions passed to the data parallel operations are “pure” functions, i.e. side-effect free.

To users of our library, we suggest that the user-defined functions be pure functions. This is particularly easy in the Scala language, which is designed to support a functional programming style. Because functions in Scala are objects and easily passed as parameters (in contrast to Java), this requirement is much more easily met in a Scala program. However, with the library-only approach, there is no way to force the user to pass only pure functions to the library. And therefore, we cannot guarantee determinacy.

In future research, we plan to explore the possibility of modifying the Scala compiler to support data parallel operations. We hope that the modified compiler can check the pure function requirement, and thus be able to guarantee determinate parallel computation. Also, the compiler approach should allow more comprehensive vertical integration of the data parallel operations, including out-of-order execution to improve performance.

References


