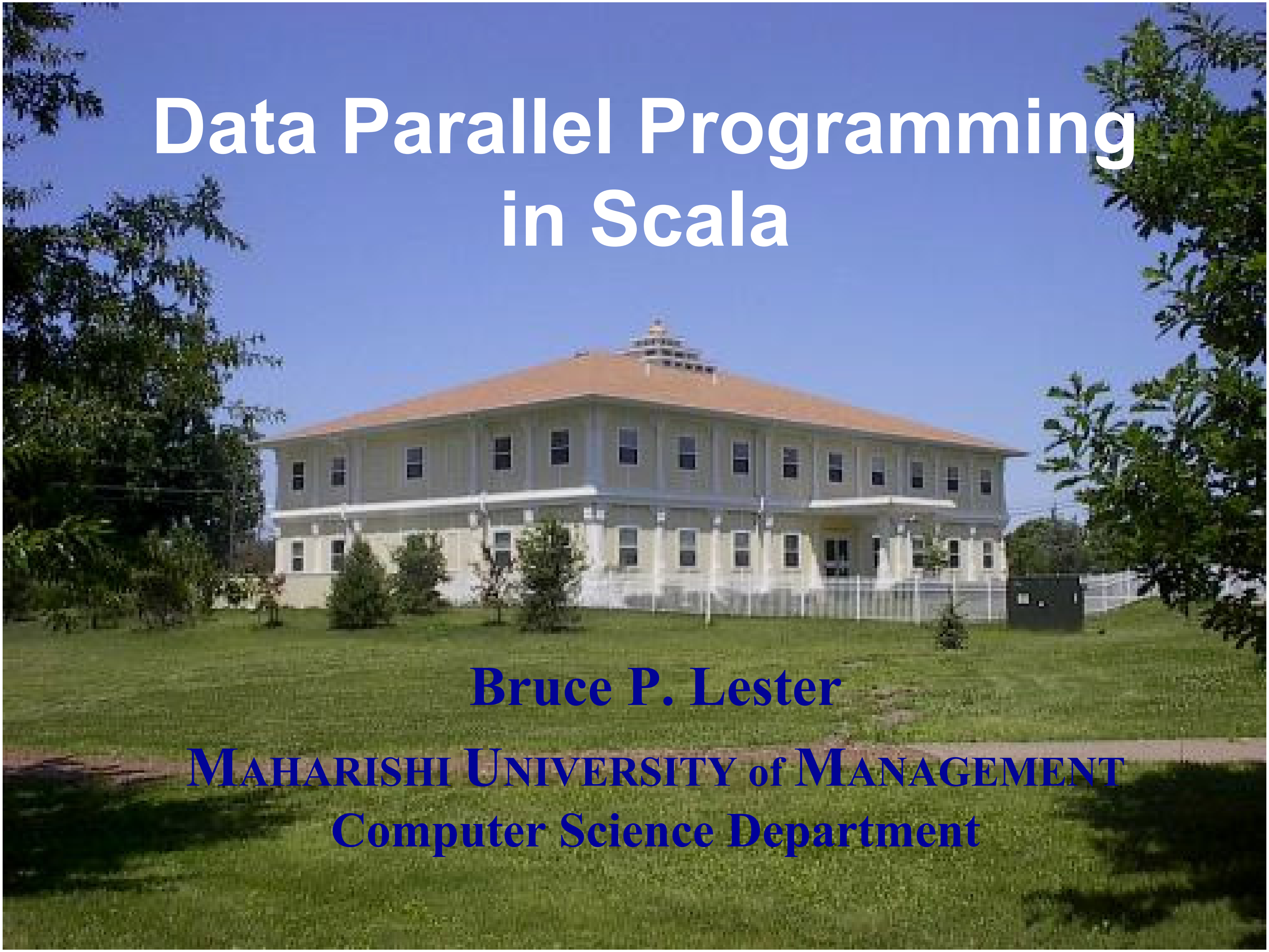
Data Parallel Programming in Scala



Bruce P. Lester

MAHARISHI UNIVERSITY of MANAGEMENT
Computer Science Department

Slide Presentation
Data Parallel Programming in Scala
Bruce P. Lester

Multi-Core Processors

- All the new computer processors have multiple cores.
- The number of cores per processor is expected to double every 2-3 years.
- This offers a potential for enormous performance improvement.
- For this potential to be fully realized, techniques for exposing parallelism in computer programs are needed.

Parallel Programs

- Parallelizing compilers have had some success, but are limited because of sequential constraints found in 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.
- To help the programmer specify parallelism in a program, the programming language must have some special parallel programming features.

Multi-threading

- The predominant approach used so far is multi-threading.
- If the parallel threads modify shared data, then locking is used to provide atomic access.
- The programmer is involved in many of the low-level details of management and synchronization of parallel tasks.

Nondeterministic Programs

- Multi-threaded programs may have data races that create a nondeterministic program.
- Program deadlocks may also occur in a nondeterministic fashion.
- Nondeterminism complicates the software development process, and makes it more difficult to develop reliable software.

Data Parallel Programming

- High-level collection-oriented operations, in which every element of a collection is operated upon in parallel by the same operation.
- One example is the array operations of the language Fortran 90.
- MapReduce operation popularized by Google is another example of a data parallel operation.

Data Parallelism in Scala

- First phase of our research is to implement data parallelism in Scala completely with a new class library.
- Library implements the parallel data structures and the allowable operations on them.
- The paper contains a detailed description of this library, including a sample data parallel Scala program and performance results.

Parallel Vectors (*PVector*)

- Indexed sequence of data items, which bears some resemblance to a one-dimensional array.
- 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].
- Examples:
  – PVector[Double] PVector[String]
  – PVector[Employee]
  – PVector[Array[Double]]

The Data Parallel Library

Operations on PVectors

- 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

Map Operation

map[U](unaryop: (T) => U): PVector[U]

- 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.
- Example:

```
var Mask: new PVector[Boolean]
B = Mask.map( !_ )
```

Combine Operation

combine[U](op: (T,T) => U,
bVec: PVector[T]): PVector[U]

- **Example:**

  val A: PVector[Int](n)

  val B: PVector[Int](n)

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

- **A and B must have same length and component type.**
- **Abstract execution model is a virtual processor operating in parallel at each element of the PVector.**

Advantages of 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.
- Scala allows simple functions to be compactly expressed, such as

  (x, y) => x + y   expressed as: _+_

  (x) => !x   expressed as: !_

Reduce Operation

reduce(binop: (T,T) => T): T

- ***binop*** **is an (associative) user-defined function**
- **Uses *binop* to reduce the input PVector to a single value of type T**
- **Example:**

A = new PVector[Int](aListofIntegers)

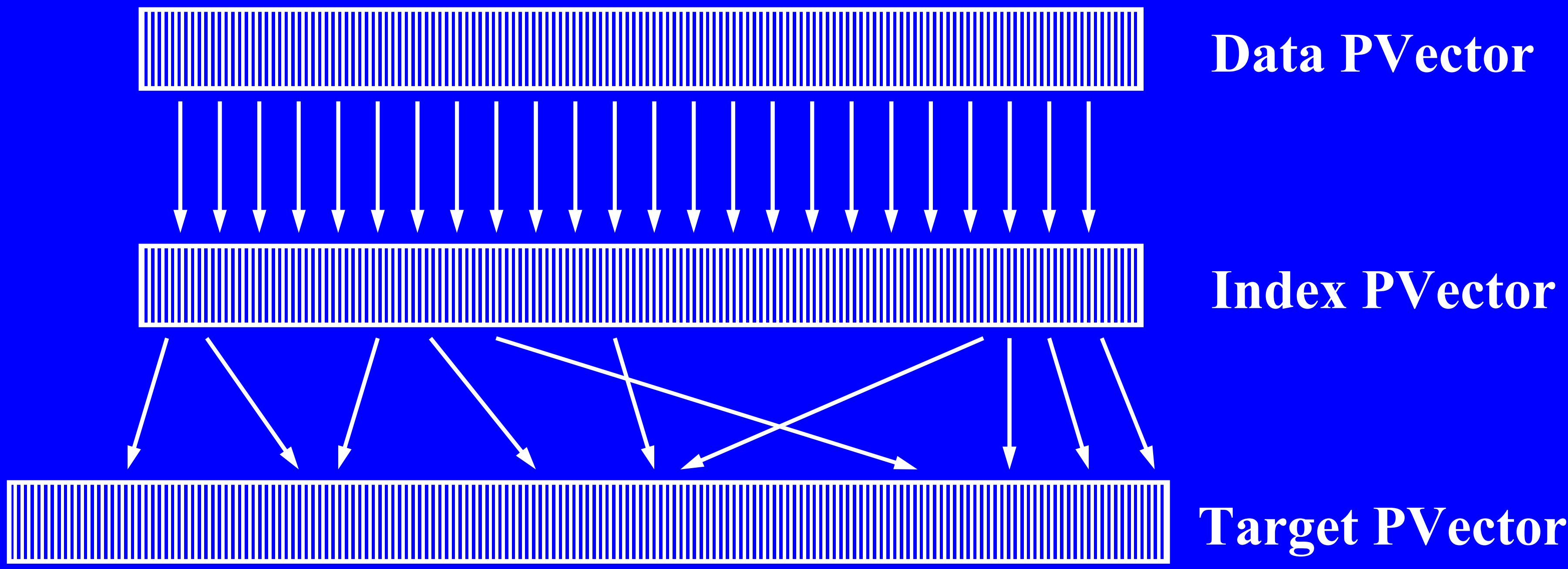
result = A.reduce(_+_)

Scan Operation

scan(binop: (T,T) => T): T

- **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.**

Keyed-Reduce Operation



- User-defined function will combine colliding data values
- Target initially has default values

Permute Operation

permute(Index: PVector[Int]): PVector[T]

Data: [30 5 -2 10]

Index: [3 0 1 2] (index in Data vector)

Output: [10 30 5 -2]

Select Operation

select(Mask: PVector[Bolean]): PVector[T]

Data: [30 5 -2 10 76 -5 20 55]

Mask: [T T F T T F F T]

Result: [30 5 10 76 55]

Initialize Operations

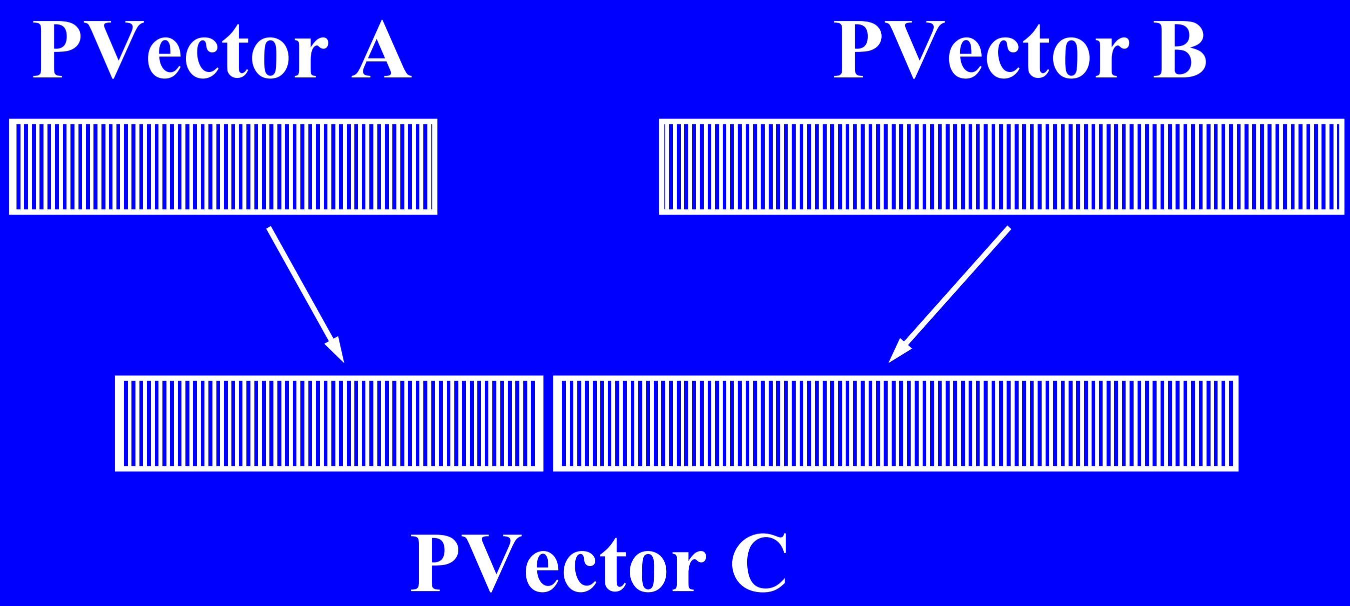
- **Broadcast(*n*, *value*): Creates a new PVector consisting of *n* elements, each of which is a copy of *value*.**

- **Index(*n*): Creates new PVector of with elements 0, 1, 2, …, *n*.**

Append Operation

append(aVec: PVector[T]): PVector[T]

C = A append B



Assign Operation

assign(source: PVector[T]): PVector[T]

B.assign(A)

- **PVectors A and B must conform: same base type and number of elements**
- **Copies elements of A into corresponding elements of B**
- **Different from ordinary assignment (B = A), which copies the object reference in variable A to variable B.**

Immutability

- Generally, the operations do not modify existing PVectors.
- Output of operation is new PVector created from data in existing PVectors.
- Two of the operations do modify an existing PVector:
  - Keyed-reduce
  - Assign
- Therefore, PVectors are *not* immutable.

Input/Output Operations

- Allows external data to be pushed into a PVector or extracted from a PVector.
- PVector constructors allow creation of a PVector with the same elements as an ordinary List or Array object.
- *Read* Operation: A List or Array may be created from the elements of any PVector.

Data Parallel Library

Category	Operations
Map	Map, Combine
Reduce	Reduce, Scan, Keyed-Reduce
Permute	Permute, Select
Initialize	Broadcast, Index, Append, Assign
Input/Output	List-Input, Read, Get, Set

Conditional Execution

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

Where Mask

- *Where Mask* is a Boolean PVector.
- *C*reates a data parallel version of a general purpose *if* statement in ordinary code.
- *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.

Data Parallel Looping

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

- *True* value in *loopMask* causes corresponding virtual processor to continue looping.
- *False* causes virtual processor to terminate its looping.

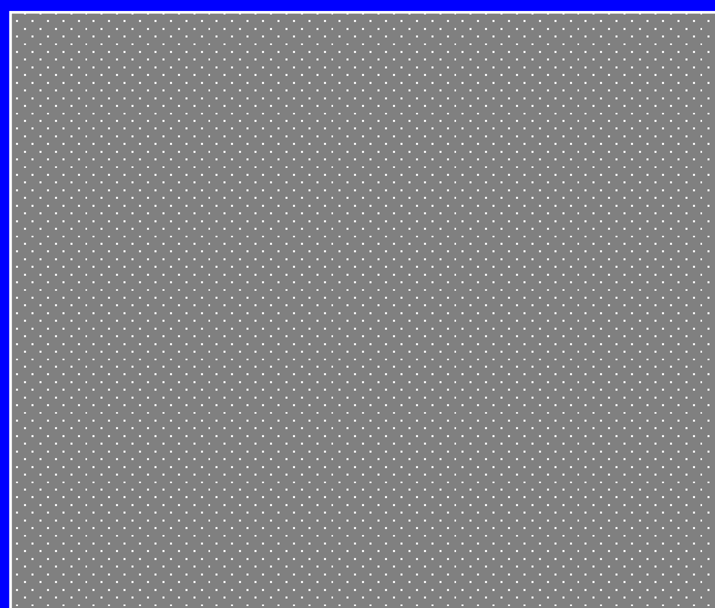
Data Parallel Looping

- **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.**
- **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*.**

Example Program
using
Data Parallel Library

Example Application Program

- Two-dimensional rectangular metal sheet
- Hold voltage fixed at four boundaries
- Want to compute resultant voltage at all internal points

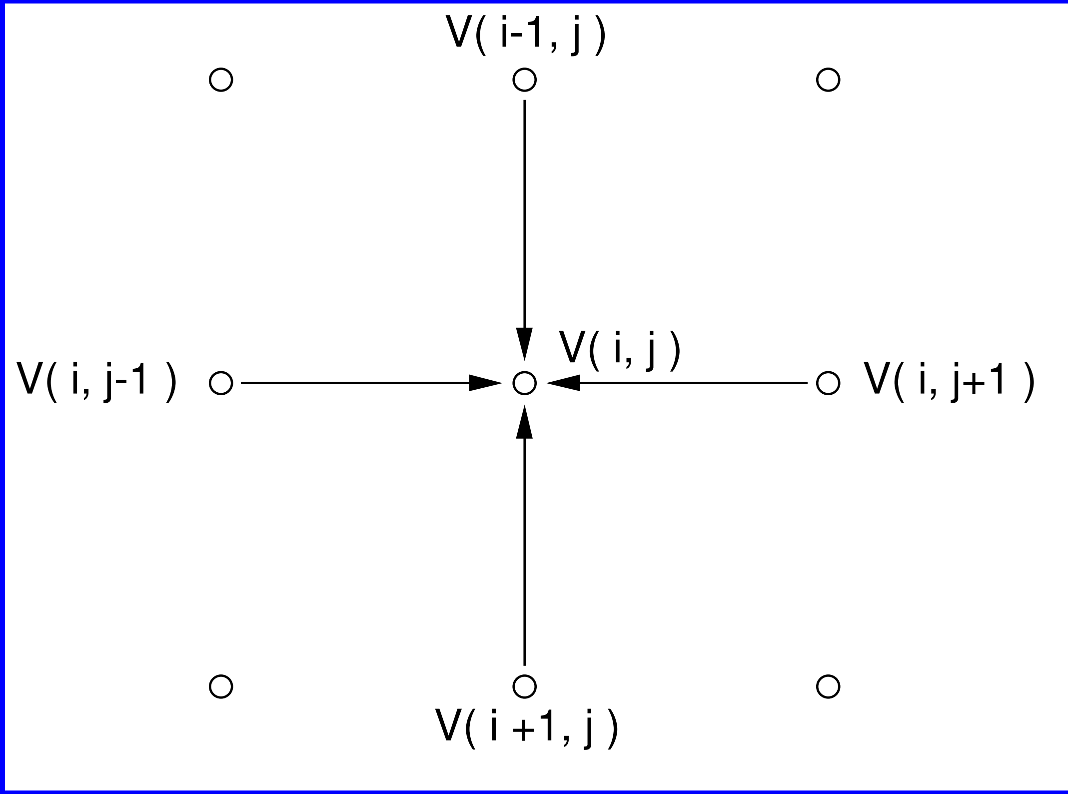


Laplace's Equation

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

Votage distribution *v* can be determined by solving Laplace's Equation in two-dimensions

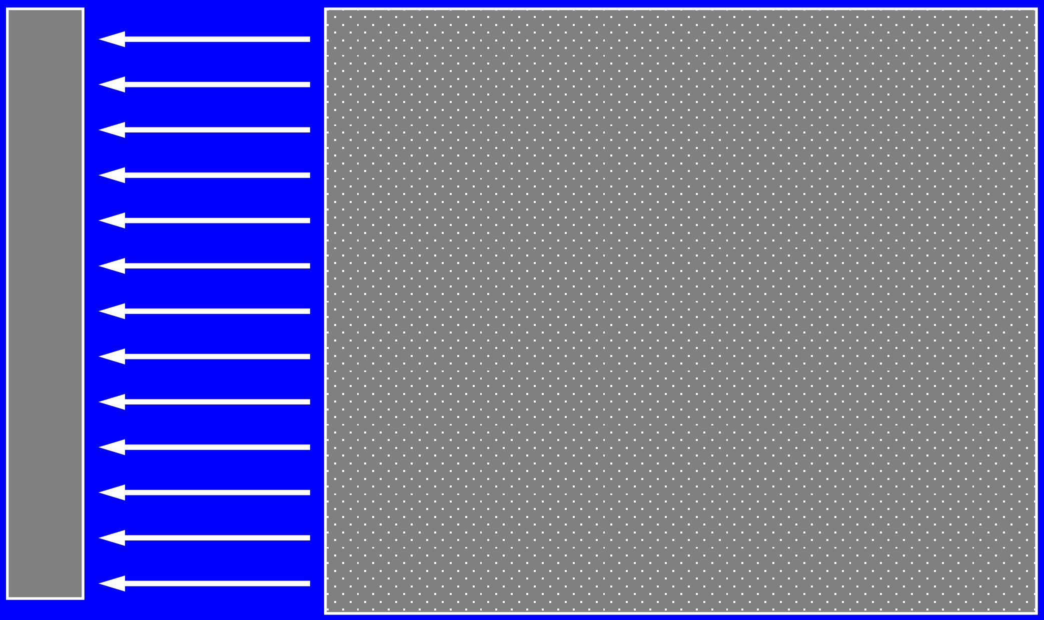
Jacobi Relaxation



- Two-dimensional array of points
- Iteratively recompute the value at each point as the average of the four immediate neighbors

Data Parallel Jacobi Relaxation

- Each row of the two-dimensional array of points is one element of PVector A.
- var A: PVector[Array[Double]]



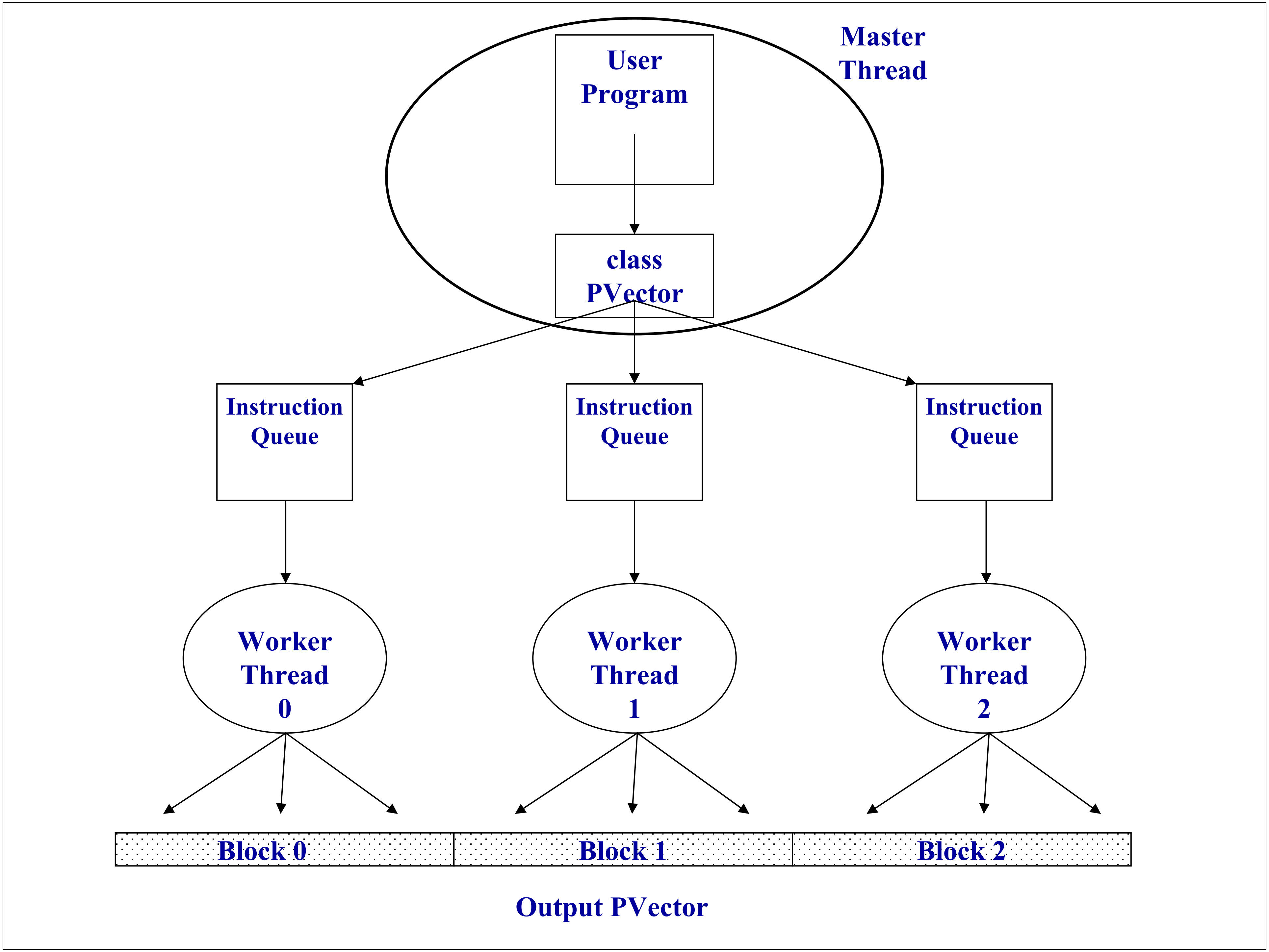
Variable Declaration and Initialization

```
def JacobiRelaxation(n: Int) = {
  val tolerance: Double = 0.0001
    . . .     //  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)
   …  //  continued on next slide
```

Main loop of Parallel Jacobi Relaxation

```
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( _&&_ )     // convergence test
  A.assign( B )
}  while( !done )
Where.end
```

Library Implementation and Performance


Master Thread
User Program
class PVector
Instruction Queue
Instruction Queue
Instruction Queue
Worker Thread 0
Worker Thread 1
Worker Thread 2
Block 0
Block 1
Block 2
Output PVector

Instruction Format

Instruction Field	**Type**
1	Int (Opcode)
2	PVector reference
3	PVector reference
4	PVector reference
5	function of one parameter
6	function of two parameters

Vertical Integration

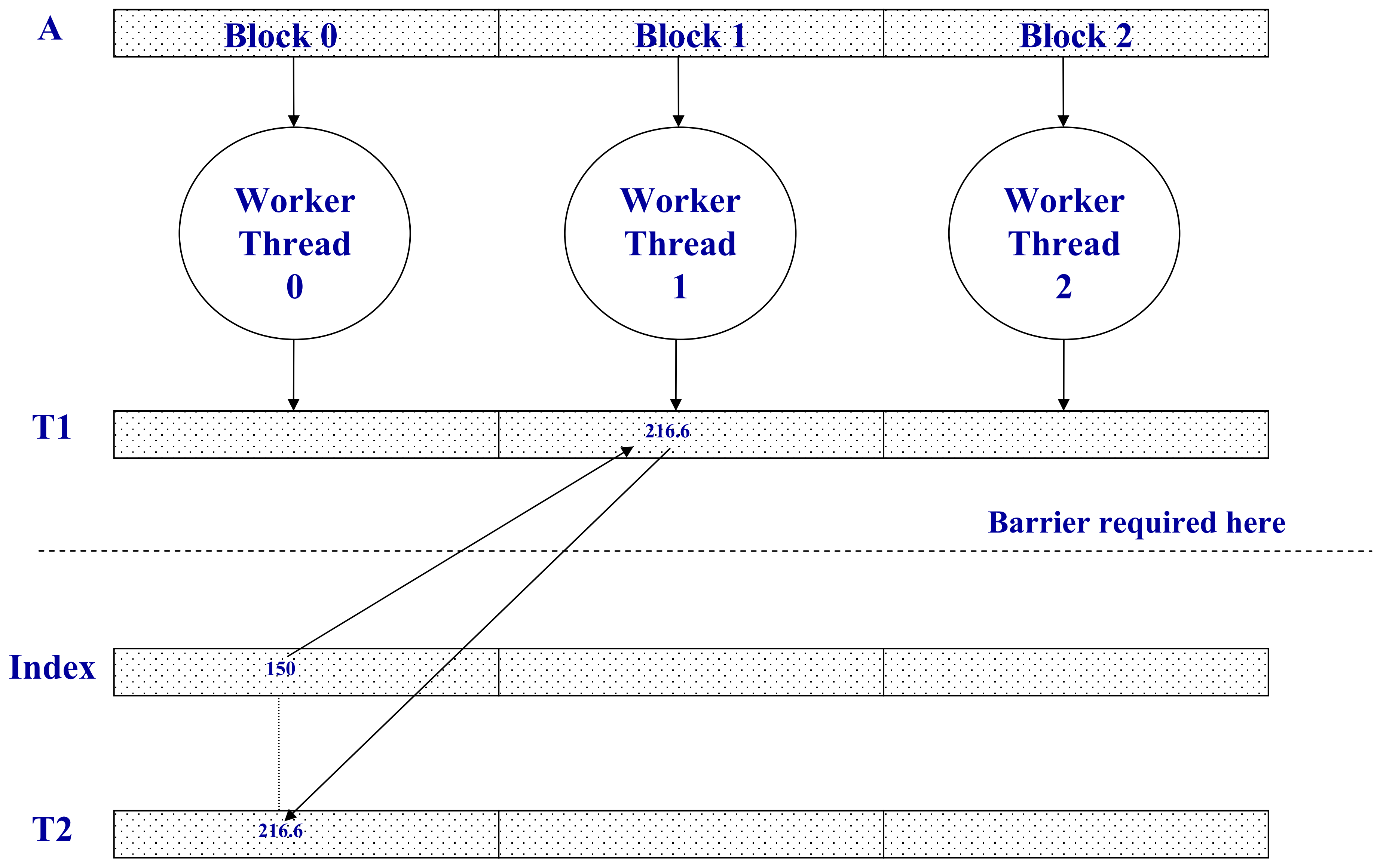
```
T1 = A.map( _*2.0 )                  //  T1 = A*2.0
T2 = T1.combine( _+_ , B )           //  T2 = T1 + B
D  = T2.combine( _/_ , C )           //  D = T2/C
```

- The usual way to implement any library function call is to complete the function fully and then return to the caller.
- Therefore, the participating parallel threads must all participate in a time-comsuming synchronization.
- However, this is not necessary because the intermediate results computed by the Workers do not cross the block boundaries.
- This vertical integration of data parallel operations greatly improves the performance.

Barrier Synchronization

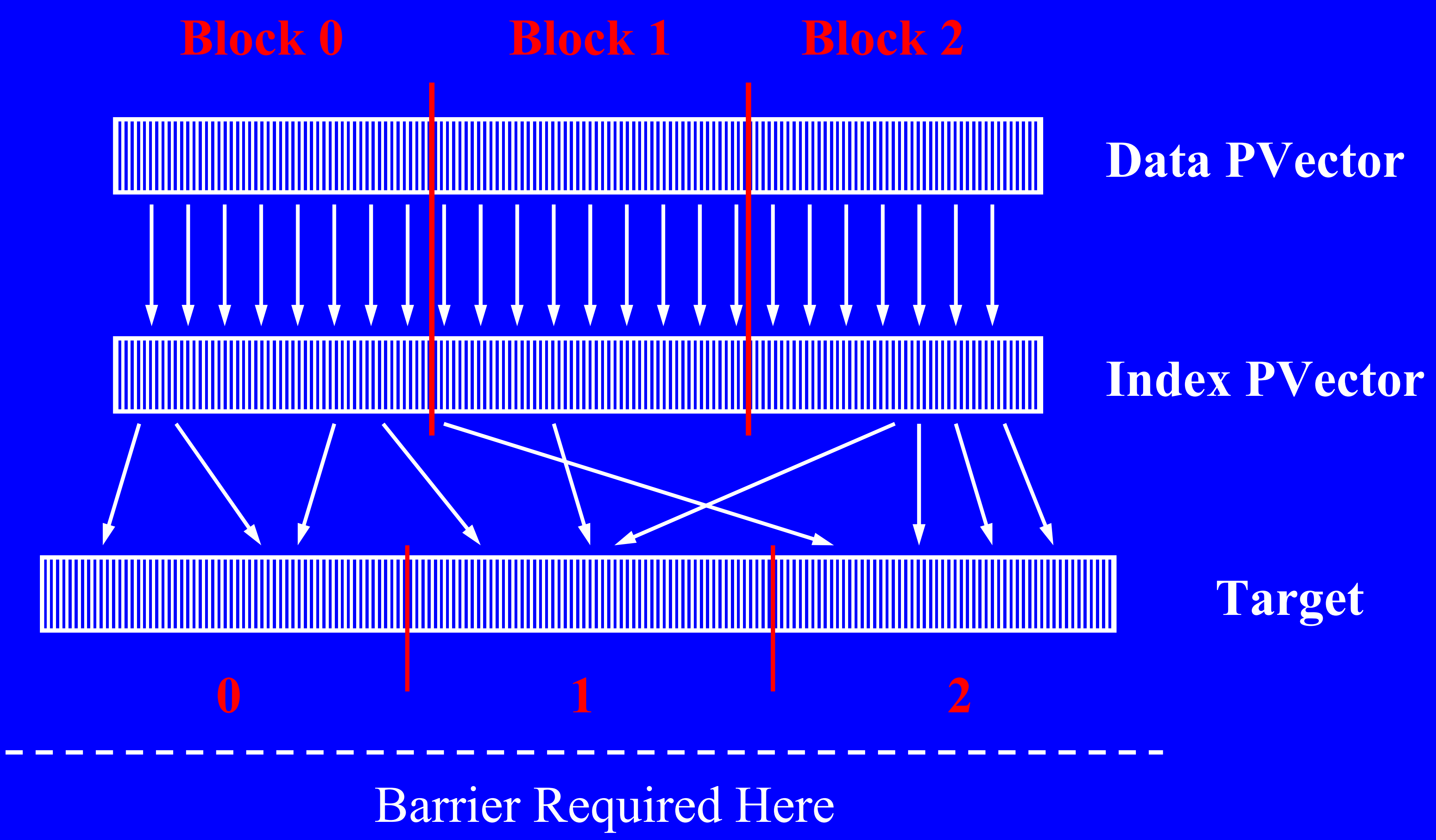
```
T1 = A.map( _*2.0 )            //  T1 = 2*A
T2 = T1.permute(Index)     //   permute T1
```

- There are some library operations that do require the Workers to cross block boundaries .
- If the boundary crossing occurs in one of the input PVectors to an operation, then a barrier is required before the operation begins.
- If the boundary crossing occurs in the output PVector of an operation, then a barrier is required at the end of the operation.



Permute Operation Requires Barrier

Keyed-Reduce Requires Output Barrier

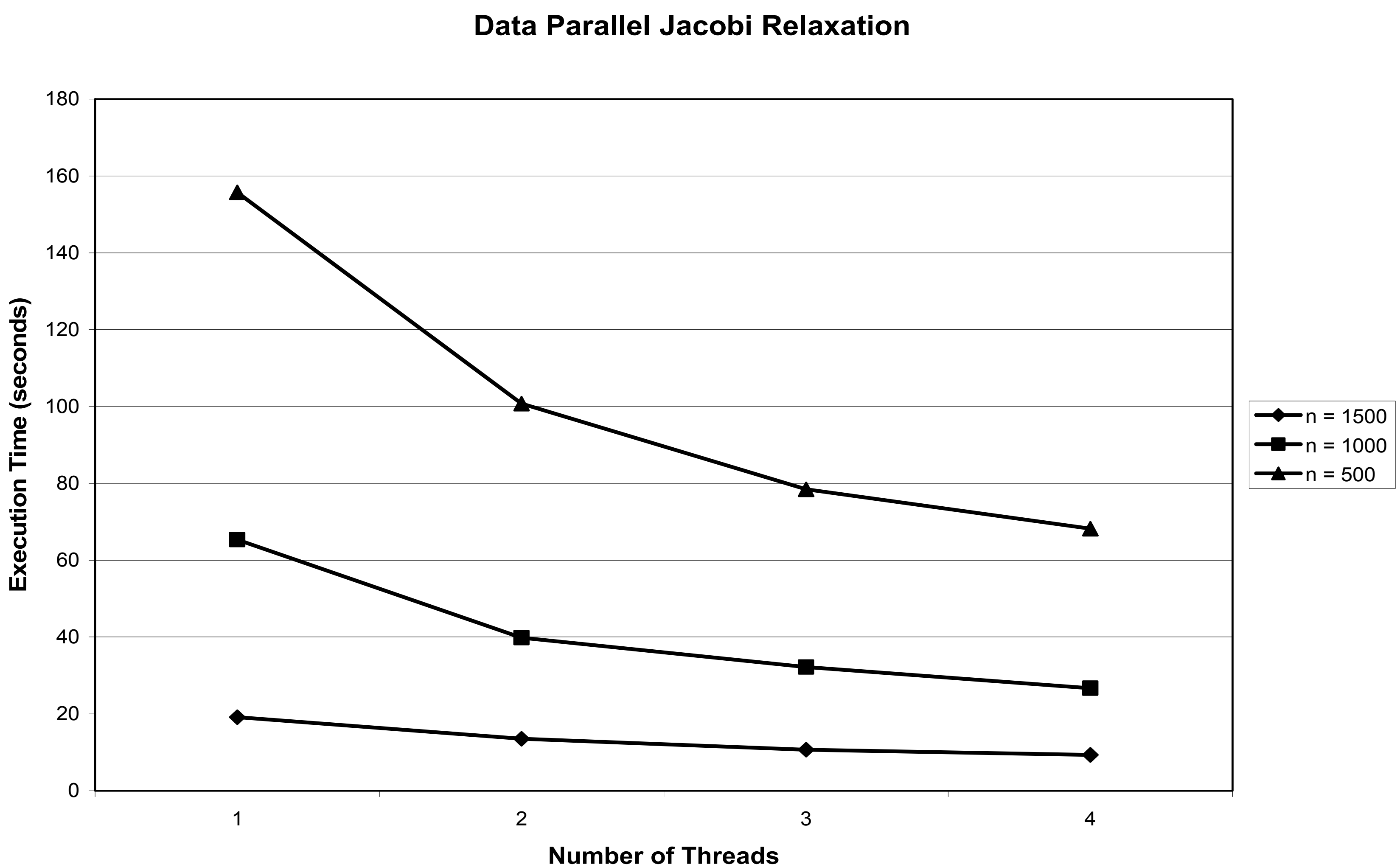


Operation	Input Block Crossing	Output Block Crossing
map	no	no
combine	no	no
reduce	no	no
scan	yes	no
keyed-reduce	no	yes
permute	yes	no
select	no	yes
broadcast	no	no
index	no	no
append	no	yes
assign	no	no
list-input	no	no
read	no	no
get	no	no
set	no	no

Master Thread Synchronization

- 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 can progress far ahead of the implementation of the data parallel operations by the Workers.
- 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.
- Reduce, Read, Get operations require barrier among Workers and Master Thread.

Library Performance



Speedup on Four-Core Processor

		Data Size *n*			
		100	500	1000	1500
Threads	1	1	1	1	1
	2	1.4	1.5	1.6	1.5
	3	1.5	1.8	2	2
	4	1.6	2.1	2.4	2.2

Our Research Contribution

- The operations we use in the data parallel library are found in other libraries.
- Our data parallel library has following novel features:
  - Where Mask
  - Vertical Integration
  - Scala Implementation

Determinacy

- Standard multi-threading for writing parallel programs may result in data races on shared data, which leads to nondeterminism.
- Data parallel programming library can solve this problem, provided that the user-defined functions passed to the data parallel operations are "pure" functions, i.e. side-effect free.
- Using current Scala compiler, there is no way to force the user to pass only pure functions to the library.