"The minimal restriction that guarantees that the unpredictability of ordering will not affect the behavior of a program is then that, for any possible serialization order for the operations performed by a program, any two consecutive operations of the program that are not causally related must commute with respect to the memory state that precedes the first. (Note that exchanging such commutative operations produces another possible serialization order.)"

– Guy Steele, 1990

Commuting Statements

- Two kinds of statements: Assignments and Tests.
- Execution of an Assignment evaluates an expression and writes the results to a variable. The expression is a composition of functional operators applied to values of variables
- A Test evaluates a predicate on values of variables.
- We consider scalar variables and vector variables. Vectors are arrays of one dimension.
- Memory *state* is a mapping of the set of variable identifiers to the domain of values.
- The result of a Test only affects control flow, the sequence of statement esecution. There is no effect on memory

Definition

Two statements S1 and S1 *commute* if and only if one of the following is true:

- S1 and S2 are both assignments and they write different variables.
- S1 and S2 are both assignments, both write variable v, and the final value of v is independent of the order in which S1 and S2 are performed.
- One statement is a test, the other an assignment and the target variable of the assignment is not read by the test.

Definition

A parallel program P is (final value) *repeatable* if, for each choice of input data, every run of P produces the same result.

Theorem

Let P be a parallel program, in which every pair of concurrent statements commute. Then:

1. *Program P is repeatable.*
2. *A functional program can be constructed that is equivalent to P.*

Note: Steele sketched a proof of (1) in his 1990 paper. Demonstration of (2) has been done for a reduced version of Habanero Java, as we discuss here.

Example in Habanero Java

```
int[] multiplyByVector ( int [ ][ ] A, int [ ] X) {
   int m = A.length;
   int n = X.length;
   finish {
      int [ ] Y = new int [ m ];
      for (int i = 0; i < m; i++) {
         async {
            int sum = 0.;
            finish {
               for (int j = 0; j < n; j++) {
                  async {
                     sum += A [ i ][ j ] * X [ j ];
                  }
               }
            }
            Y [ i ] = sum;
         }
      }
    }
    return Y;
}
```

Habanero Java Syntax

Prog	::=	*FinishStmnt*
FinishStmnt	::=	**finish** { *StmntSeq* }
StmntSeq	::=	*Stmnt* | *Stmnt* ; *StmntSeq*
Stmnt	::=	**finish** { *StmntSeq* }
	|	**async** { *StmntSeq* } *StmntSeq*
	|	**for** *Variable* **from** *Expr* **to** *Expr StmntSeq*
	|	**cond** *Variable* **then** *StmntSeq* **else** *StmntSeq*
	|	*Variable* $\leftarrow$ *Expr*
	|	*VectorVar* [*Variable*] $\leftarrow$ *Expr*
Expr	::=	*SimpleExp* | *VectorVar* [*Variable*]
VectorVar	::=	*VectorId*

Concurrency Graph of an HJ Program

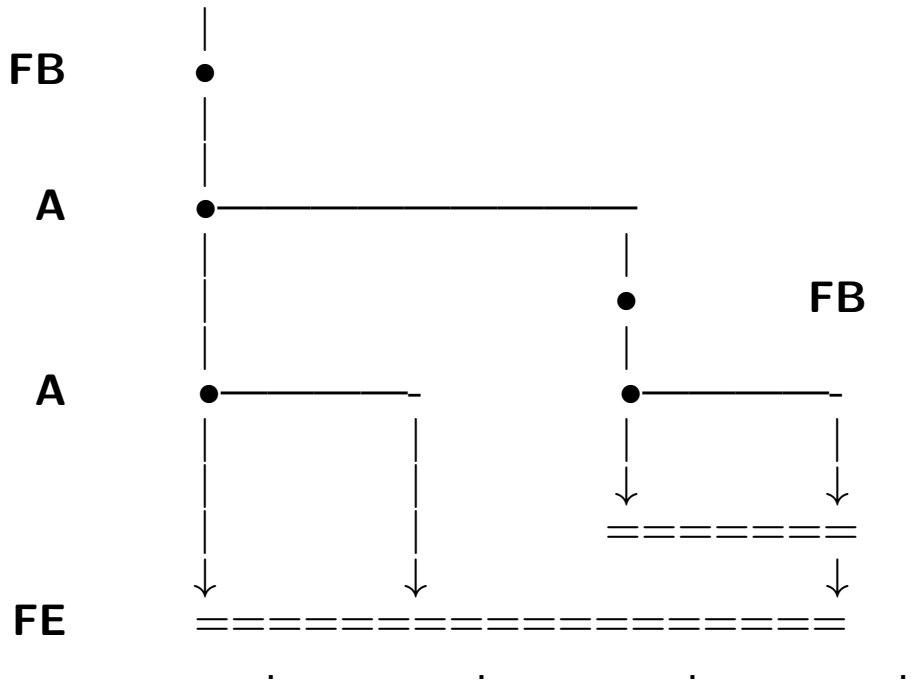
Definition

The *concurrency graph* of an HJ program is a DAG with three node types:

- async nodes (**A**) representing an *AsyncStmnt*
- f-begin nodes (**FB**) representing the start of a *FinishStmnt*
- f-end nodes (**FE**) representing the end of a *FinishStmnt*

The edges of the concurrency graph represent sequences of basic statements: *AssignStmnt*. *CondStmnt*, *WhileStmnt*

The Concurrency Graph of a Finish Statement



Example Program Skeleton

```
finish {
   async {
      finish {
         async { }
      }
   }
   async { }
}
```

Definition

Two statements S1 and S2 of an HJ program P are *concurrent* if the concurrency graph of P contains no directed path from S1 to S2 or from S2 to S1.

Semantics of HJ Finish and Async

$\mathcal{I}$ ([[*Prog*]] , Env) = $\mathcal{I}$ ([[*FinishStmnt*]], Env)

$\mathcal{I}$ ([[*StmntSeq*]] , Env) =
Stmnt ; $\Rightarrow$ $\mathcal{I}$ ([[*Stmnt*]] , Env)
Stmnt ; *StmntSeq* $\Rightarrow$ $\mathcal{I}$ ([[*StmntSeq*]] , $\mathcal{I}$ ([[*Stmnt*]] , Env))

$\mathcal{I}$ ([[*Stmnt*]] , Env) =

finish *StmntSeq* $\Rightarrow$ $\mathcal{I}$ ([[*StmntSeq*]] , Env))

async { $StmntSeq_1$ } $StmntSeq_2$ $\Rightarrow$
CombineEnv ($\mathcal{I}$ ([[$StmntSeq_1$]] , Env), $\mathcal{I}$ ([[$StmntSeq_2$]] , Env))

Variable $\leftarrow$ *Expr* $\Rightarrow$ EnvAppend (Env, $\mathcal{I}$ ([[*Expr*]] , Env))

VectorVar [*Variable*] $\leftarrow$ *Expr* $\Rightarrow$
EnvAppend (Env, $\mathcal{I}$ ([[*VectorVar*]] , Env),
$\mathcal{M}$ ([[*Variable*]] , Env), $\mathcal{I}$ ([[*Expr*]] , Env)))

Combining Environments

Because a *FinishStmnt* will generally include one or more **async** statements, it is necessary to provide a semantic function that combines the effects on FS variables of the several threads begun by **async** statements. This is provided by function *CombineEnv* which combines the effects represented by the environment modifications performed by the several threads. To accomplish this, some information about the variables of the FS is needed; this is given by the map

$$Map : VarId \rightarrow VarClass$$

Thus the *CombineEnv* function has the signature

$$CombineEnv : Env \times map \times Env \times Map \rightarrow Env \times Map$$

The CombineEnv function is defined by the changes made to Env and Map separately for each variable. The rules for some variable of two environments is shown in the following table:

Variable Kinds

- Combining environments exploits properties of accesses of variables performed by threads that produce each transformed environment.
- We distinguish three kinds of accesses to scalar variables and five kinds for vector variables.
- The next two slides list the kinds with comments on their relation to commuting statements.

Scalar Variable Kinds

- **null**: The variable is not accessed by any statement of the threads. The statement will commute with anything!
- **read**: Read-only scalar variables: variables that are read by one or more statements of the thread but are never written by any statement of the thread. Statements with read access commute with other statements with the same access to the variable.
- **update**: The thread includes statements that update the variable using a commutative/associative operator.
- **invalid**: The thread contains a statement (for example a write statement) that will not generally commute with any statement. This also applies if the threads include updates of the variable that do not use the same operator.

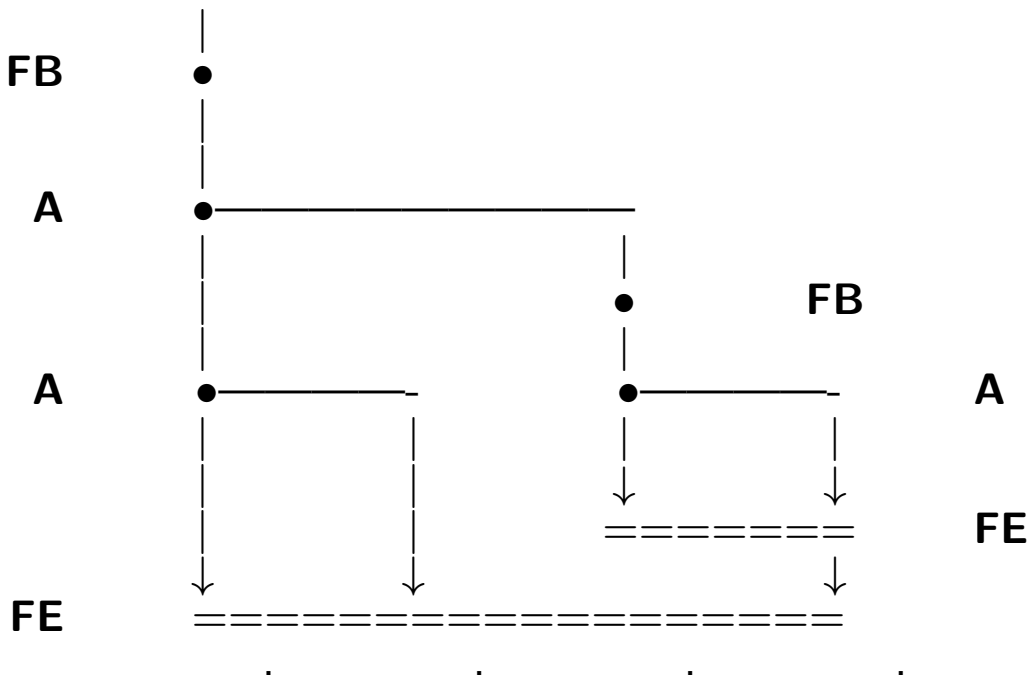
Vector Variable Kinds

- **null**: The variable is not accessed by any statement of the thread. The statement will commute with anything!
- read: Array(vector) The thread includes one or more statements that read either a fixed or indexed element of the vector. Such statements commute with each other.
- **indexed**: All statements of the thread that access the vector, whether Read Write or Update, use a different index value. The set of index values used by statements in the set of threads to which the map applies are recorded in a index list that is part of the variablekind. The common case is where the index of a vector accesss is in a range defined by an afine formula from the index variable of an iteration.
- **update**: Every statement of the threads that access the vector perform an update using the same commutative/associative operator. How the index of the vector element is determined does not matter.
- **invalid**: The threads contain a statement that accesses the vector in a way that will not generally commute with any other statement. For example, a statement (not an update) that writes an element of the vector. Also accesses are invalid if they are updates using different operators.

Kind(1)	Kind(2)		Kind	Value	
null	Kind	$\longrightarrow$	Kind	Value(2)	
Kind	**null**	$\longrightarrow$	Kind	Value(1)	
read	**read**	$\longrightarrow$	**read**	Value(1)	$[=$ Value(2)$]$
invalid	—	$\longrightarrow$	**invalid**	**undef**	
—	**invalid**	$\longrightarrow$	**invalid**	**undef**	
update	**update**	$\longrightarrow$	**update**	$Value(1)\,OpValue(2)$	if $Op(1) = Op(2)$
			invalid	**undef**	otherwise
	Vector Variables Only				
indexed	**indexed**	$\longrightarrow$	**indexed**	$List(1) \cup List(2)$	if $List(1) \cap List(2) = \emptyset$
			invalid	**undef**	otherwise

As a biadic operator on a domain of pairs *CombineEnv* is both commutative and associative. Therefore we are free to apply *CombineEnv* to the collection of threads of an *FinishStmnt* in any convenient order, with assurance that the result will be correct.

Transform: HJ to FJ



Composition of functions:
Each is an Environment transform

1. Each Edge of the CG:
   Sequence of Basic Statements
2. Async Node: Apply CombineEn
3. Contained Finish Statement:
   by recursive application
   of the transform

Functional Java Syntax

Prog	::=	**make** *IdList ExprList*
Expression	::=	*SimpleExpr* | *LetExpr* | *WhileExpr*
	|	*VectorSelect* | *Reduction* | *VectorConstr*
LetExp	::=	**let** *Variable* **in** *Expression*
	|	**let** *VectorVar* **in** *Expression*
WhileExp	::=	**while** *Variable* **with** *IdList* **do** (*ExprList*)
IdList	::=	*Id* | *Id* , *IdList*
VectorSelect	::=	*VectorVar* [*Variable*]
Reduction	::=	**reduce** *Variable* **in** [*Variable* .. *Variable*] **with** *CommAssocOp* { *Expression* }
VectorConstr	::=	**vector** *Variable* **in** [*Variable* .. *Variable*] { *Expression* }
ExprList	::=	*Expression* | *Expression* , *ExprList*
IdList	::=	*Id* | *Id* , *IdList*

Example in Functional Java

The example in Functional Java using **reduce** and **vector**. The **reduce** and **vector** expressions specify an expression to be evaluated for each index in the range.

```
int[] MatrixTimesVector (int [ ][ ] A, int [ ] X) {
    make (Y) {
       let m = VectorLength (A)
       let n = VectorLength (X)
       let Y = vector i in [1 ..  m] {
          let V = A[i] in
          reduce j in [1 ..  n] with plus {
             let op1 = V[j]
             let op2 = X[j]
             in op1 * op2
          }
       }
    }
}
```

Realistic Parallel Programs

Q: How can a program with non-commuting statements be repeatable?

A1: One or both statements of each non-commuting pair is never executed for any choice of input data. That is, the program contains dead code.

A2: The commuting law for statements is satisfied by the non-commuting statements for all values encountered in any computation by the program

Q: Are there realistic programs that are repeatable although they contain non-commuting pair of concurrent statements?

A1: Any ideas??