

# Verifying Parallel Programs with MPI-SPIN

## Part 4: Numerical Computation

Stephen F. Siegel

Department of Computer and Information Sciences  
University of Delaware

EuroPVM/MPI 2007  
Paris, France  
30 September 2007

# Overview

1. Goal: prove that program computes correct result
2. Symbolic execution
  - Performing symbolic arithmetic in MPI-SPIN
3. Functional equivalence
4. Three types of numerical equivalence
5. Diffusion revisited
6. Dealing with branches: the path condition

# Goal: prove that program computes correct result

- what does it mean to say program computes correct result?

## Goal: prove that program computes correct result

- what does it mean to say program computes correct result?
- verification requires **specification**
  - definition of *correct*

## Goal: prove that program computes correct result

- what does it mean to say program computes correct result?
- verification requires **specification**
  - definition of *correct*
- our specification
  - a trusted **sequential** version of program

## Goal: prove that program computes correct result

- what does it mean to say program computes correct result?
- verification requires **specification**
  - definition of *correct*
- our specification
  - a trusted **sequential** version of program
- our method
  - use MPI-SPIN to prove the sequential and parallel program are **functionally equivalent**
    - i.e., produce same output for any given input
  - reduces the problem of verifying the correctness of a **parallel** numerical program to the problem of verifying the correctness of a **sequential** numerical program
  - uses symbolic execution to model floating-point computation

## How do we model floating-point computation?

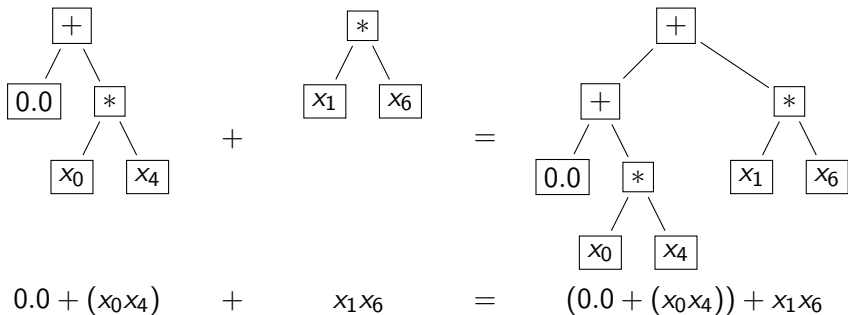
- one double-precision floating-point variable has  $2^{64}$  possible states
- abstraction?

## How do we model floating-point computation?

- one double-precision floating-point variable has  $2^{64}$  possible states
- abstraction?

**Input:** symbolic constants  $x_0, x_1, \dots$

**Output:** symbolic expressions in the  $x_i$





# Performing symbolic arithmetic in MPI-SPIN

- type
  - `MPI_Symbolic`

# Performing symbolic arithmetic in MPI-SPIN

- type
  - `MPI_Symbolic`
- constants of type `MPI_Symbolic`
  1. `SYM_ZERO`: zero (0)
  2. `SYM_ONE`: one (1)
  3. `SYM_FALSE`: the boolean value `false`
  4. `SYM_TRUE`: the boolean value `true`

# Performing symbolic arithmetic in MPI-SPIN

- type
  - `MPI_Symbolic`
- constants of type `MPI_Symbolic`
  1. `SYM_ZERO`: zero (0)
  2. `SYM_ONE`: one (1)
  3. `SYM_FALSE`: the boolean value `false`
  4. `SYM_TRUE`: the boolean value `true`
- basic functions
  1. `MPI_Symbolic SYM_intConstant(int n)`
    - returns symbolic expression with single node containing  $n$
    - e.g., `4`
  2. `MPI_Symbolic SYM_symbolicConstant(int i)`
    - returns symbolic expression with single node containing  $x_i$
    - e.g., `x4`
    - usually used to model input
    - also used to represent floating-point constants ( $\pi$ ,  $e$ , ...)

## Forming new symbolic expressions from old

- the following return an `MPI_Symbolic` of **numeric** type
  1. `SYM_add(MPI_Symbolic x, MPI_Symbolic y)`
  2. `SYM_subtract(MPI_Symbolic x, MPI_Symbolic y)`
  3. `SYM_multiply(MPI_Symbolic x, MPI_Symbolic y)`
  4. `SYM_divide(MPI_Symbolic x, MPI_Symbolic y)`
  5. `SYM_sqrt(MPI_Symbolic x)`
  6. `SYM_abs(MPI_Symbolic x)`
  7. `SYM_if(MPI_Symbolic b, MPI_Symbolic x, MPI_Symbolic y)`
    - `(b ? x : y)`

## Forming new symbolic expressions from old

- the following return an `MPI_Symbolic` of `boolean` type
  1. `SYM_equals(MPI_Symbolic x, MPI_Symbolic y)`
    - `/* x == y */`
  2. `SYM_nequals(MPI_Symbolic x, MPI_Symbolic y)`
    - `/* x != y */`
  3. `SYM_lessThan(MPI_Symbolic x, MPI_Symbolic y)`
  4. `SYM_greaterThan(MPI_Symbolic x, MPI_Symbolic y)`
  5. `SYM_lessThanOrEqualTo(MPI_Symbolic x, MPI_Symbolic y)`
  6. `SYM_greaterThanOrEqualTo(MPI_Symbolic x, MPI_Symbolic y)`
  7. `SYM_conjunct(MPI_Symbolic p, MPI_Symbolic q)`
    - `/* p && q */`
  8. `SYM_negate(MPI_Symbolic p)`
    - `/* !p */`

# How does MPI-SPIN represent symbolic expressions?

## Problem

- symbolic expressions can get big
- there can be millions of states
- storing all symbolic expressions in every state would quickly consume all memory

# How does MPI-SPIN represent symbolic expressions?

## Problem

- symbolic expressions can get big
- there can be millions of states
- storing all symbolic expressions in every state would quickly consume all memory

## Solution: Value numbering

- place all symbolic expressions in a shared **expression table**
  - every expression has a unique **ID number**

# How does MPI-SPIN represent symbolic expressions?

## Problem

- symbolic expressions can get big
- there can be millions of states
- storing all symbolic expressions in every state would quickly consume all memory

## Solution: Value numbering

- place all symbolic expressions in a shared **expression table**
  - every expression has a unique **ID number**
- in the model...
  - replace all floating-point values with ID numbers



# How does MPI-SPIN represent symbolic expressions?

## Problem

- symbolic expressions can get big
- there can be millions of states
- storing all symbolic expressions in every state would quickly consume all memory

## Solution: Value numbering

- place all symbolic expressions in a shared **expression table**
  - every expression has a unique **ID number**
- in the model...
  - replace all floating-point values with ID numbers
  - replace all floating-point operations with symbolic operations
    - to evaluate  $x + y$ :
    - is  $x + y$  already in the table?
    - if **yes**, return its ID number
    - if **no**, create new table entry and return new ID number

# Symbolic expression table: example

$i$	$e_i$	interpretation
-----	-------	----------------

## Symbolic expression table: example

$i$	$e_i$	interpretation
0	(L, 0.0)	0.0
1	(L, 1.0)	1.0

## Symbolic expression table: example

$i$	$e_i$	interpretation
0	(L, 0.0)	0.0
1	(L, 1.0)	1.0
2	(X, 0)	$x_0$
3	(X, 1)	$x_1$
4	(X, 2)	$x_2$
5	(X, 3)	$x_3$
6	(X, 4)	$x_4$
7	(X, 5)	$x_5$
8	(X, 6)	$x_6$
9	(X, 7)	$x_7$

## Symbolic expression table: example

$i$	$e_i$	interpretation
0	(L, 0.0)	0.0
1	(L, 1.0)	1.0
2	(X, 0)	$x_0$
3	(X, 1)	$x_1$
4	(X, 2)	$x_2$
5	(X, 3)	$x_3$
6	(X, 4)	$x_4$
7	(X, 5)	$x_5$
8	(X, 6)	$x_6$
9	(X, 7)	$x_7$
10	(* , 2, 6)	$x_0x_4$

## Symbolic expression table: example

$i$	$e_i$	interpretation
0	(L, 0.0)	0.0
1	(L, 1.0)	1.0
2	(X, 0)	$x_0$
3	(X, 1)	$x_1$
4	(X, 2)	$x_2$
5	(X, 3)	$x_3$
6	(X, 4)	$x_4$
7	(X, 5)	$x_5$
8	(X, 6)	$x_6$
9	(X, 7)	$x_7$
10	(*, 2, 6)	$x_0x_4$
11	(+, 0, 10)	$0.0 + x_0x_4$

## Symbolic expression table: example

$i$	$e_i$	interpretation
0	(L, 0.0)	0.0
1	(L, 1.0)	1.0
2	(X, 0)	$x_0$
3	(X, 1)	$x_1$
4	(X, 2)	$x_2$
5	(X, 3)	$x_3$
6	(X, 4)	$x_4$
7	(X, 5)	$x_5$
8	(X, 6)	$x_6$
9	(X, 7)	$x_7$
10	(*, 2, 6)	$x_0 x_4$
11	(+, 0, 10)	$0.0 + x_0 x_4$
12	(*, 3, 8)	$x_1 x_6$

## Symbolic expression table: example

$i$	$e_i$	interpretation
0	(L, 0.0)	0.0
1	(L, 1.0)	1.0
2	(X, 0)	$x_0$
3	(X, 1)	$x_1$
4	(X, 2)	$x_2$
5	(X, 3)	$x_3$
6	(X, 4)	$x_4$
7	(X, 5)	$x_5$
8	(X, 6)	$x_6$
9	(X, 7)	$x_7$
10	(*, 2, 6)	$x_0x_4$
11	(+, 0, 10)	$0.0 + x_0x_4$
12	(*, 3, 8)	$x_1x_6$

$i$	$e_i$	interpretation
13	(+, 11, 12)	$(0.0 + x_0x_4) + x_1x_6$



## Symbolic expression table: example

$i$	$e_i$	interpretation
0	(L, 0.0)	0.0
1	(L, 1.0)	1.0
2	(X, 0)	$x_0$
3	(X, 1)	$x_1$
4	(X, 2)	$x_2$
5	(X, 3)	$x_3$
6	(X, 4)	$x_4$
7	(X, 5)	$x_5$
8	(X, 6)	$x_6$
9	(X, 7)	$x_7$
10	(*, 2, 6)	$x_0x_4$
11	(+, 0, 10)	$0.0 + x_0x_4$
12	(*, 3, 8)	$x_1x_6$

$i$	$e_i$	interpretation
13	(+, 11, 12)	$(0.0 + x_0x_4) + x_1x_6$
14	(*, 2, 7)	$x_0x_5$

## Symbolic expression table: example

$i$	$e_i$	interpretation
0	(L, 0.0)	0.0
1	(L, 1.0)	1.0
2	(X, 0)	$x_0$
3	(X, 1)	$x_1$
4	(X, 2)	$x_2$
5	(X, 3)	$x_3$
6	(X, 4)	$x_4$
7	(X, 5)	$x_5$
8	(X, 6)	$x_6$
9	(X, 7)	$x_7$
10	(*, 2, 6)	$x_0x_4$
11	(+, 0, 10)	$0.0 + x_0x_4$
12	(*, 3, 8)	$x_1x_6$

$i$	$e_i$	interpretation
13	(+, 11, 12)	$(0.0 + x_0x_4) + x_1x_6$
14	(*, 2, 7)	$x_0x_5$
15	(+, 0, 14)	$0.0 + x_0x_5$

## Symbolic expression table: example

$i$	$e_i$	interpretation
0	(L, 0.0)	0.0
1	(L, 1.0)	1.0
2	(X, 0)	$x_0$
3	(X, 1)	$x_1$
4	(X, 2)	$x_2$
5	(X, 3)	$x_3$
6	(X, 4)	$x_4$
7	(X, 5)	$x_5$
8	(X, 6)	$x_6$
9	(X, 7)	$x_7$
10	(*, 2, 6)	$x_0x_4$
11	(+, 0, 10)	$0.0 + x_0x_4$
12	(*, 3, 8)	$x_1x_6$

$i$	$e_i$	interpretation
13	(+, 11, 12)	$(0.0 + x_0x_4) + x_1x_6$
14	(*, 2, 7)	$x_0x_5$
15	(+, 0, 14)	$0.0 + x_0x_5$
16	(*, 3, 9)	$x_1x_7$

## Symbolic expression table: example

$i$	$e_i$	interpretation
0	(L, 0.0)	0.0
1	(L, 1.0)	1.0
2	(X, 0)	$x_0$
3	(X, 1)	$x_1$
4	(X, 2)	$x_2$
5	(X, 3)	$x_3$
6	(X, 4)	$x_4$
7	(X, 5)	$x_5$
8	(X, 6)	$x_6$
9	(X, 7)	$x_7$
10	(*, 2, 6)	$x_0x_4$
11	(+, 0, 10)	$0.0 + x_0x_4$
12	(*, 3, 8)	$x_1x_6$

$i$	$e_i$	interpretation
13	(+, 11, 12)	$(0.0 + x_0x_4) + x_1x_6$
14	(*, 2, 7)	$x_0x_5$
15	(+, 0, 14)	$0.0 + x_0x_5$
16	(*, 3, 9)	$x_1x_7$
17	(+, 15, 16)	$(0.0 + x_0x_5) + x_1x_7$

## Symbolic expression table: example

$i$	$e_i$	interpretation
0	(L, 0.0)	0.0
1	(L, 1.0)	1.0
2	(X, 0)	$x_0$
3	(X, 1)	$x_1$
4	(X, 2)	$x_2$
5	(X, 3)	$x_3$
6	(X, 4)	$x_4$
7	(X, 5)	$x_5$
8	(X, 6)	$x_6$
9	(X, 7)	$x_7$
10	(*, 2, 6)	$x_0x_4$
11	(+, 0, 10)	$0.0 + x_0x_4$
12	(*, 3, 8)	$x_1x_6$

$i$	$e_i$	interpretation
13	(+, 11, 12)	$(0.0 + x_0x_4) + x_1x_6$
14	(*, 2, 7)	$x_0x_5$
15	(+, 0, 14)	$0.0 + x_0x_5$
16	(*, 3, 9)	$x_1x_7$
17	(+, 15, 16)	$(0.0 + x_0x_5) + x_1x_7$
18	(*, 4, 6)	$x_2x_4$

## Symbolic expression table: example

$i$	$e_i$	interpretation
0	(L, 0.0)	0.0
1	(L, 1.0)	1.0
2	(X, 0)	$x_0$
3	(X, 1)	$x_1$
4	(X, 2)	$x_2$
5	(X, 3)	$x_3$
6	(X, 4)	$x_4$
7	(X, 5)	$x_5$
8	(X, 6)	$x_6$
9	(X, 7)	$x_7$
10	(*, 2, 6)	$x_0x_4$
11	(+, 0, 10)	$0.0 + x_0x_4$
12	(*, 3, 8)	$x_1x_6$

$i$	$e_i$	interpretation
13	(+, 11, 12)	$(0.0 + x_0x_4) + x_1x_6$
14	(*, 2, 7)	$x_0x_5$
15	(+, 0, 14)	$0.0 + x_0x_5$
16	(*, 3, 9)	$x_1x_7$
17	(+, 15, 16)	$(0.0 + x_0x_5) + x_1x_7$
18	(*, 4, 6)	$x_2x_4$
19	(+, 0, 12)	$0.0 + x_2x_4$

## Symbolic expression table: example

$i$	$e_i$	interpretation
0	(L, 0.0)	0.0
1	(L, 1.0)	1.0
2	(X, 0)	$x_0$
3	(X, 1)	$x_1$
4	(X, 2)	$x_2$
5	(X, 3)	$x_3$
6	(X, 4)	$x_4$
7	(X, 5)	$x_5$
8	(X, 6)	$x_6$
9	(X, 7)	$x_7$
10	(*, 2, 6)	$x_0x_4$
11	(+, 0, 10)	$0.0 + x_0x_4$
12	(*, 3, 8)	$x_1x_6$

$i$	$e_i$	interpretation
13	(+, 11, 12)	$(0.0 + x_0x_4) + x_1x_6$
14	(*, 2, 7)	$x_0x_5$
15	(+, 0, 14)	$0.0 + x_0x_5$
16	(*, 3, 9)	$x_1x_7$
17	(+, 15, 16)	$(0.0 + x_0x_5) + x_1x_7$
18	(*, 4, 6)	$x_2x_4$
19	(+, 0, 12)	$0.0 + x_2x_4$
20	(*, 5, 8)	$x_3x_6$

## Symbolic expression table: example

$i$	$e_i$	interpretation
0	(L, 0.0)	0.0
1	(L, 1.0)	1.0
2	(X, 0)	$x_0$
3	(X, 1)	$x_1$
4	(X, 2)	$x_2$
5	(X, 3)	$x_3$
6	(X, 4)	$x_4$
7	(X, 5)	$x_5$
8	(X, 6)	$x_6$
9	(X, 7)	$x_7$
10	(*, 2, 6)	$x_0x_4$
11	(+, 0, 10)	$0.0 + x_0x_4$
12	(*, 3, 8)	$x_1x_6$

$i$	$e_i$	interpretation
13	(+, 11, 12)	$(0.0 + x_0x_4) + x_1x_6$
14	(*, 2, 7)	$x_0x_5$
15	(+, 0, 14)	$0.0 + x_0x_5$
16	(*, 3, 9)	$x_1x_7$
17	(+, 15, 16)	$(0.0 + x_0x_5) + x_1x_7$
18	(*, 4, 6)	$x_2x_4$
19	(+, 0, 12)	$0.0 + x_2x_4$
20	(*, 5, 8)	$x_3x_6$
21	(+, 19, 20)	$(0.0 + x_2x_4) + x_3x_6$



## Symbolic expression table: example

$i$	$e_i$	interpretation
0	(L, 0.0)	0.0
1	(L, 1.0)	1.0
2	(X, 0)	$x_0$
3	(X, 1)	$x_1$
4	(X, 2)	$x_2$
5	(X, 3)	$x_3$
6	(X, 4)	$x_4$
7	(X, 5)	$x_5$
8	(X, 6)	$x_6$
9	(X, 7)	$x_7$
10	(*, 2, 6)	$x_0x_4$
11	(+, 0, 10)	$0.0 + x_0x_4$
12	(*, 3, 8)	$x_1x_6$

$i$	$e_i$	interpretation
13	(+, 11, 12)	$(0.0 + x_0x_4) + x_1x_6$
14	(*, 2, 7)	$x_0x_5$
15	(+, 0, 14)	$0.0 + x_0x_5$
16	(*, 3, 9)	$x_1x_7$
17	(+, 15, 16)	$(0.0 + x_0x_5) + x_1x_7$
18	(*, 4, 6)	$x_2x_4$
19	(+, 0, 12)	$0.0 + x_2x_4$
20	(*, 5, 8)	$x_3x_6$
21	(+, 19, 20)	$(0.0 + x_2x_4) + x_3x_6$
22	(*, 4, 7)	$x_2x_5$

## Symbolic expression table: example

$i$	$e_i$	interpretation
0	(L, 0.0)	0.0
1	(L, 1.0)	1.0
2	(X, 0)	$x_0$
3	(X, 1)	$x_1$
4	(X, 2)	$x_2$
5	(X, 3)	$x_3$
6	(X, 4)	$x_4$
7	(X, 5)	$x_5$
8	(X, 6)	$x_6$
9	(X, 7)	$x_7$
10	(*, 2, 6)	$x_0x_4$
11	(+, 0, 10)	$0.0 + x_0x_4$
12	(*, 3, 8)	$x_1x_6$

$i$	$e_i$	interpretation
13	(+, 11, 12)	$(0.0 + x_0x_4) + x_1x_6$
14	(*, 2, 7)	$x_0x_5$
15	(+, 0, 14)	$0.0 + x_0x_5$
16	(*, 3, 9)	$x_1x_7$
17	(+, 15, 16)	$(0.0 + x_0x_5) + x_1x_7$
18	(*, 4, 6)	$x_2x_4$
19	(+, 0, 12)	$0.0 + x_2x_4$
20	(*, 5, 8)	$x_3x_6$
21	(+, 19, 20)	$(0.0 + x_2x_4) + x_3x_6$
22	(*, 4, 7)	$x_2x_5$
23	(+, 0, 22)	$0.0 + x_2x_5$

## Symbolic expression table: example

$i$	$e_i$	interpretation
0	(L, 0.0)	0.0
1	(L, 1.0)	1.0
2	(X, 0)	$x_0$
3	(X, 1)	$x_1$
4	(X, 2)	$x_2$
5	(X, 3)	$x_3$
6	(X, 4)	$x_4$
7	(X, 5)	$x_5$
8	(X, 6)	$x_6$
9	(X, 7)	$x_7$
10	(*, 2, 6)	$x_0x_4$
11	(+, 0, 10)	$0.0 + x_0x_4$
12	(*, 3, 8)	$x_1x_6$

$i$	$e_i$	interpretation
13	(+, 11, 12)	$(0.0 + x_0x_4) + x_1x_6$
14	(*, 2, 7)	$x_0x_5$
15	(+, 0, 14)	$0.0 + x_0x_5$
16	(*, 3, 9)	$x_1x_7$
17	(+, 15, 16)	$(0.0 + x_0x_5) + x_1x_7$
18	(*, 4, 6)	$x_2x_4$
19	(+, 0, 12)	$0.0 + x_2x_4$
20	(*, 5, 8)	$x_3x_6$
21	(+, 19, 20)	$(0.0 + x_2x_4) + x_3x_6$
22	(*, 4, 7)	$x_2x_5$
23	(+, 0, 22)	$0.0 + x_2x_5$
24	(*, 5, 9)	$x_3x_7$

## Symbolic expression table: example

$i$	$e_i$	interpretation
0	(L, 0.0)	0.0
1	(L, 1.0)	1.0
2	(X, 0)	$x_0$
3	(X, 1)	$x_1$
4	(X, 2)	$x_2$
5	(X, 3)	$x_3$
6	(X, 4)	$x_4$
7	(X, 5)	$x_5$
8	(X, 6)	$x_6$
9	(X, 7)	$x_7$
10	(*, 2, 6)	$x_0x_4$
11	(+, 0, 10)	$0.0 + x_0x_4$
12	(*, 3, 8)	$x_1x_6$

$i$	$e_i$	interpretation
13	(+, 11, 12)	$(0.0 + x_0x_4) + x_1x_6$
14	(*, 2, 7)	$x_0x_5$
15	(+, 0, 14)	$0.0 + x_0x_5$
16	(*, 3, 9)	$x_1x_7$
17	(+, 15, 16)	$(0.0 + x_0x_5) + x_1x_7$
18	(*, 4, 6)	$x_2x_4$
19	(+, 0, 12)	$0.0 + x_2x_4$
20	(*, 5, 8)	$x_3x_6$
21	(+, 19, 20)	$(0.0 + x_2x_4) + x_3x_6$
22	(*, 4, 7)	$x_2x_5$
23	(+, 0, 22)	$0.0 + x_2x_5$
24	(*, 5, 9)	$x_3x_7$
25	(+, 23, 24)	$(0.0 + x_2x_5) + x_3x_7$

# Functional equivalence

- **Goal**
  - prove sequential and parallel programs are functionally equivalent

# Functional equivalence

- **Goal**
  - prove sequential and parallel programs are functionally equivalent
- **Method**
  1. construct symbolic model  $M_{\text{seq}}$  of sequential program
    - input:  $\mathbf{x} = (x_1, \dots, x_n)$ , output:  $\mathbf{y}$

# Functional equivalence

- **Goal**
  - prove sequential and parallel programs are functionally equivalent
- **Method**
  1. construct symbolic model  $M_{\text{seq}}$  of sequential program
    - input:  $\mathbf{x} = (x_1, \dots, x_n)$ , output:  $\mathbf{y}$
  2. construct symbolic model  $M_{\text{par}}$  of parallel program
    - input:  $\mathbf{x} = (x_1, \dots, x_n)$ , output:  $\mathbf{y}'$
    - using same symbolic table

# Functional equivalence

- **Goal**
  - prove sequential and parallel programs are functionally equivalent
- **Method**
  1. construct symbolic model  $M_{\text{seq}}$  of sequential program
    - input:  $\mathbf{x} = (x_1, \dots, x_n)$ , output:  $\mathbf{y}$
  2. construct symbolic model  $M_{\text{par}}$  of parallel program
    - input:  $\mathbf{x} = (x_1, \dots, x_n)$ , output:  $\mathbf{y}'$
    - using same symbolic table
  3. create composite model  $M$ :
    - $M_{\text{seq}}; M_{\text{par}}; \text{assert}(\mathbf{y} = \mathbf{y}')$ ;



# Functional equivalence

- **Goal**
  - prove sequential and parallel programs are functionally equivalent
- **Method**
  1. construct symbolic model  $M_{\text{seq}}$  of sequential program
    - input:  $\mathbf{x} = (x_1, \dots, x_n)$ , output:  $\mathbf{y}$
  2. construct symbolic model  $M_{\text{par}}$  of parallel program
    - input:  $\mathbf{x} = (x_1, \dots, x_n)$ , output:  $\mathbf{y}'$
    - using same symbolic table
  3. create composite model  $M$ :
    - $M_{\text{seq}}; M_{\text{par}}; \text{assert}(\mathbf{y} = \mathbf{y}')$ ;
  4. use MPI-SPIN to verify the assertion in  $M$  can never be violated

# Functional equivalence

- **Goal**
  - prove sequential and parallel programs are functionally equivalent
- **Method**
  1. construct symbolic model  $M_{\text{seq}}$  of sequential program
    - input:  $\mathbf{x} = (x_1, \dots, x_n)$ , output:  $\mathbf{y}$
  2. construct symbolic model  $M_{\text{par}}$  of parallel program
    - input:  $\mathbf{x} = (x_1, \dots, x_n)$ , output:  $\mathbf{y}'$
    - using same symbolic table
  3. create composite model  $M$ :
    - $M_{\text{seq}}; M_{\text{par}}; \text{assert}(\mathbf{y} = \mathbf{y}')$ ;
  4. use MPI-SPIN to verify the assertion in  $M$  can never be violated

MPI-SPIN returns either

- **Yes**: the property holds, or

# Functional equivalence

- **Goal**
  - prove sequential and parallel programs are functionally equivalent
- **Method**
  1. construct symbolic model  $M_{\text{seq}}$  of sequential program
    - input:  $\mathbf{x} = (x_1, \dots, x_n)$ , output:  $\mathbf{y}$
  2. construct symbolic model  $M_{\text{par}}$  of parallel program
    - input:  $\mathbf{x} = (x_1, \dots, x_n)$ , output:  $\mathbf{y}'$
    - using same symbolic table
  3. create composite model  $M$ :
    - $M_{\text{seq}}; M_{\text{par}}; \text{assert}(\mathbf{y} = \mathbf{y}')$ ;
  4. use MPI-SPIN to verify the assertion in  $M$  can never be violated

MPI-SPIN returns either

- **Yes**: the property holds, or
- **No + counterexample**:
  - a trace through  $M_{\text{seq}}$
  - a trace through  $M_{\text{par}}$
  - the values of  $\mathbf{y}$ , and  $\mathbf{y}'$

## Numerical Issues

**Problem:** distinct symbolic expressions should be considered “**equivalent**” in some cases

**Example:** **real equivalence**

- $((x_3 + x_1) + x_2) + x_0$  and  $((x_0 + x_1) + x_2) + x_3$

## Numerical Issues

**Problem:** distinct symbolic expressions should be considered “**equivalent**” in some cases

**Example:** **real equivalence**

- $((x_3 + x_1) + x_2) + x_0$  and  $((x_0 + x_1) + x_2) + x_3$
- if computer arithmetic were real arithmetic then evaluating these expressions would yield identical results for any values of  $x_0, x_1, \dots$

# Numerical Issues

**Problem:** distinct symbolic expressions should be considered “**equivalent**” in some cases

**Example:** **real equivalence**

- $((x_3 + x_1) + x_2) + x_0$  and  $((x_0 + x_1) + x_2) + x_3$
- if computer arithmetic were real arithmetic then evaluating these expressions would yield identical results for any values of  $x_0, x_1, \dots$
- computer arithmetic is not real arithmetic
  - e.g., floating-point addition can never be associative
  - **What every computer scientist should know about floating-point arithmetic**
    - **David Goldberg**
    - **ACM Computing Surveys 23(1), 1991**

## Numerical Issues, cont.

- in some cases, knowing the sequential and parallel programs are “**real equivalent**” is good enough
- if testing yields slightly different results. . .
  - . . .but you know programs are real equivalent
  - then you know the **only reason** results differ is due to vagaries of floating-point arithmetic
  - and **not** to some error in your parallel program

## Three equivalence relations

- MPI-SPIN supports three different **equivalence relations** on the set of symbolic expressions
  0. Herbrand
  1. IEEE
  2. Real



## Three equivalence relations

- MPI-SPIN supports three different **equivalence relations** on the set of symbolic expressions
  0. Herbrand
  1. IEEE
  2. Real
- user specifies which numeric mode to use at command-line
  - `ms -sym=0 ...` for Herbrand, etc.

## Three equivalence relations

- MPI-SPIN supports three different **equivalence relations** on the set of symbolic expressions
  0. Herbrand
  1. IEEE
  2. Real
- user specifies which numeric mode to use at command-line
  - `ms -sym=0 ...` for Herbrand, etc.
- when a new expression is formed it is reduced to a **canonical form** before being inserted into table
  - goal is for each equivalence class to have at most one representative in table

## Three equivalence relations

- MPI-SPIN supports three different **equivalence relations** on the set of symbolic expressions
  0. Herbrand
  1. IEEE
  2. Real
- user specifies which numeric mode to use at command-line
  - `ms -sym=0 ...` for Herbrand, etc.
- when a new expression is formed it is reduced to a **canonical form** before being inserted into table
  - goal is for each equivalence class to have at most one representative in table
  - this is not always achievable with 100% precision
  - estimation is always **conservative**
    - if MPI-SPIN says two expressions are equivalent then they are equivalent
    - if MPI-SPIN says they are **not** equivalent then they **might be** equivalent

## Herbrand equivalence

- two symbolic expressions are **Herbrand equivalent** iff they are identical

## Herbrand equivalence

- two symbolic expressions are **Herbrand equivalent** iff they are identical
- numeric operations are treated as **uninterpreted functions**
- example:  $x$  and  $x + 0$  are not Herbrand equivalent

## Herbrand equivalence

- two symbolic expressions are **Herbrand equivalent** iff they are identical
- numeric operations are treated as **uninterpreted functions**
- example:  $x$  and  $x + 0$  are not Herbrand equivalent
- Herbrand equivalence is the **strongest** form of equivalence
  - if two programs are Herbrand equivalent then they will produce the same result no matter how numeric operations are implemented
    - as long as the numeric operations are deterministic functions!
- in many complex examples, sequential and parallel versions are Herbrand equivalent
  - complexity lies elsewhere (e.g., in distribution of data, coordination of processes)

## IEEE equivalence

- two expressions are **IEEE equivalent** if one can be reduced to the other using the following identities:
  - $x + y = y + x$
  - $x + 0 = x = 0 + x$
  - $x - x = 0$
  - $xy = yx$
  - $1x = x = x1$
  - $x/x = 1$  (if  $x \neq 0$ )
  - $\vdots$

## IEEE equivalence

- two expressions are **IEEE equivalent** if one can be reduced to the other using the following identities:
  - $x + y = y + x$
  - $x + 0 = x = 0 + x$
  - $x - x = 0$
  - $xy = yx$
  - $1x = x = x1$
  - $x/x = 1$  (if  $x \neq 0$ )
  - $\vdots$
- rationale
  - all of these identities are guaranteed to hold on any platform conforming to the IEEE-754 standard



## IEEE equivalence

- two expressions are **IEEE equivalent** if one can be reduced to the other using the following identities:
  - $x + y = y + x$
  - $x + 0 = x = 0 + x$
  - $x - x = 0$
  - $xy = yx$
  - $1x = x = x1$
  - $x/x = 1$  (if  $x \neq 0$ )
  - $\vdots$
- rationale
  - all of these identities are guaranteed to hold on any platform conforming to the IEEE-754 standard
- IEEE equivalence is **weaker** than Herbrand equivalence
- two IEEE equivalent programs are guaranteed to produce the exact same results if executed on an IEEE-754-compliant platform
  - but results may differ on non-compliant platforms

## Real equivalence

- two expressions are **real equivalent** if one can be reduced to the other using any of the field identities
  - all of the IEEE identities
  - associativity of addition and multiplication
  - $x(1/x) = 1$
  - $\vdots$

## Real equivalence

- two expressions are **real equivalent** if one can be reduced to the other using any of the field identities
  - all of the IEEE identities
  - associativity of addition and multiplication
  - $x(1/x) = 1$
  - $\vdots$
- real equivalence is **weaker** than IEEE equivalence
  - two real-equivalent programs may produce different results, even when executed on IEEE-compliant platforms

## Real equivalence

- two expressions are **real equivalent** if one can be reduced to the other using any of the field identities
  - all of the IEEE identities
  - associativity of addition and multiplication
  - $x(1/x) = 1$
  - $\vdots$
- real equivalence is **weaker** than IEEE equivalence
  - two real-equivalent programs may produce different results, even when executed on IEEE-compliant platforms
- if arithmetic were infinite-precision, results would be identical

## Real equivalence

- two expressions are **real equivalent** if one can be reduced to the other using any of the field identities
  - all of the IEEE identities
  - associativity of addition and multiplication
  - $x(1/x) = 1$
  - $\vdots$
- real equivalence is **weaker** than IEEE equivalence
  - two real-equivalent programs may produce different results, even when executed on IEEE-compliant platforms
- if arithmetic were infinite-precision, results would be identical
- sometime real equivalence is the best that can be expected
  - suppose program uses `MPI_Reduce` or `MPI_Allreduce`
  - reduction operations is floating-point addition

## Real equivalence

- two expressions are **real equivalent** if one can be reduced to the other using any of the field identities
  - all of the IEEE identities
  - associativity of addition and multiplication
  - $x(1/x) = 1$
  - $\vdots$
- real equivalence is **weaker** than IEEE equivalence
  - two real-equivalent programs may produce different results, even when executed on IEEE-compliant platforms
- if arithmetic were infinite-precision, results would be identical
- sometime real equivalence is the best that can be expected
  - suppose program uses `MPI_Reduce` or `MPI_Allreduce`
  - reduction operations is floating-point addition
  - **MPI Standard** permits MPI implementation to perform additions in any order
    - order used on one execution could be different than order used on another execution
  - prevents any possibility of IEEE equivalence

## Numerical model of diffusion2d

Composite model:

- `diffusion/diffusion_sym.prom`
- `diffusion/diffusion_sym.c`

## The path correspondence problem

- the programs may contain branches on expressions that involve the symbolic variables
  - **if** ( $x_0 \neq 0$ ) **{...}** **else** **{...}**



## The path correspondence problem

- the programs may contain branches on expressions that involve the symbolic variables
  - **if** ( $x_0 \neq 0$ ) **{...}** **else** **{...}**
- only want to compare the result of an execution path in the parallel program to the result of a *corresponding* path in the sequential program

## Path conditions and domains

- enumerate all paths through the sequential program
  - keeping track of the **path condition** for each path

$$\mathbf{y} = \begin{cases} f_1(\mathbf{x}) & \text{if } p_1(\mathbf{x}) \\ f_2(\mathbf{x}) & \text{if } p_2(\mathbf{x}) \\ \vdots & \vdots \\ f_n(\mathbf{x}) & \text{if } p_n(\mathbf{x}) \end{cases}$$

## Path conditions and domains

- enumerate all paths through the sequential program
  - keeping track of the **path condition** for each path

$$\mathbf{y} = \begin{cases} f_1(\mathbf{x}) & \text{if } p_1(\mathbf{x}) \\ f_2(\mathbf{x}) & \text{if } p_2(\mathbf{x}) \\ \vdots & \vdots \\ f_n(\mathbf{x}) & \text{if } p_n(\mathbf{x}) \end{cases}$$

- each  $p_i$  determines a **path domain**  $D_i = \{\mathbf{x} \mid p_i(\mathbf{x})\}$
- $D_i \cap D_j = \emptyset$  if  $i \neq j$
- $\cup_i D_i$  is the whole input space

## Path conditions and domains

- enumerate all paths through the sequential program
  - keeping track of the **path condition** for each path

$$\mathbf{y} = \begin{cases} f_1(\mathbf{x}) & \text{if } p_1(\mathbf{x}) \\ f_2(\mathbf{x}) & \text{if } p_2(\mathbf{x}) \\ \vdots & \vdots \\ f_n(\mathbf{x}) & \text{if } p_n(\mathbf{x}) \end{cases}$$

- each  $p_i$  determines a **path domain**  $D_i = \{\mathbf{x} \mid p_i(\mathbf{x})\}$
- $D_i \cap D_j = \emptyset$  if  $i \neq j$
- $\cup_i D_i$  is the whole input space

### Solution to path correspondence problem:

1. discover path conditions/domains automatically
2. for each domain  $D_i$ : compare symbolic results of sequential and parallel programs for all inputs in  $D_i$

## Modeling conditional statements

To model the statement **if** ( $x_0 \neq 0$ )  $\{ \dots \}$  **else**  $\{ \dots \}$

```

p ← true; /* path condition */
⋮
b ← μ(p, x₀ ≠ 0);
if (b = -1) {
  if (choose()) {
    b ← 1; p ← p ∧ (x₀ ≠ 0);
  } else {
    b ← 0; p ← p ∧ (x₀ = 0);
  }
}
if (b = 1) { ... } else { ... }

```

$$\mu(p, q) = \begin{cases} 1 & \text{if } p \Rightarrow q \\ 0 & \text{if } p \Rightarrow \neg q \\ -1 & \text{if don't know} \end{cases}$$

for boolean-valued symbolic expressions  $p, q$

## The method: incorporating path condition

1. construct symbolic model  $M_{\text{seq}}$  of sequential program
  - input:  $\mathbf{x}$ , output:  $\mathbf{y}$ , path condition:  $p$
2. construct symbolic model  $M_{\text{par}}$  of parallel program
  - input:  $\mathbf{x}$ , output:  $\mathbf{y}'$ , path condition:  $p$
  - using same symbolic table
3. create composite model  $M$ :
  - $p \leftarrow \text{true}; M_{\text{seq}}; M_{\text{par}}; \text{assert}(\mathbf{y} = \mathbf{y}')$ ;
4. use model checker to verify the assertion in  $M$  can never be violated

## The method: incorporating path condition

1. construct symbolic model  $M_{\text{seq}}$  of sequential program
  - input:  $\mathbf{x}$ , output:  $\mathbf{y}$ , path condition:  $p$
2. construct symbolic model  $M_{\text{par}}$  of parallel program
  - input:  $\mathbf{x}$ , output:  $\mathbf{y}'$ , path condition:  $p$
  - using same symbolic table
3. create composite model  $M$ :
  - $p \leftarrow \text{true}; M_{\text{seq}}; M_{\text{par}}; \text{assert}(\mathbf{y} = \mathbf{y}')$ ;
4. use model checker to verify the assertion in  $M$  can never be violated

The model checker returns either

- Yes: the property holds, or

## The method: incorporating path condition

1. construct symbolic model  $M_{\text{seq}}$  of sequential program
  - input:  $\mathbf{x}$ , output:  $\mathbf{y}$ , path condition:  $p$
2. construct symbolic model  $M_{\text{par}}$  of parallel program
  - input:  $\mathbf{x}$ , output:  $\mathbf{y}'$ , path condition:  $p$
  - using same symbolic table
3. create composite model  $M$ :
  - $p \leftarrow \text{true}; M_{\text{seq}}; M_{\text{par}}; \text{assert}(\mathbf{y} = \mathbf{y}')$ ;
4. use model checker to verify the assertion in  $M$  can never be violated

The model checker returns either

- Yes: the property holds, or
- No + counterexample:
  - a trace through  $M_{\text{seq}}$
  - a trace through  $M_{\text{par}}$
  - the values of  $p$ ,  $\mathbf{y}$ , and  $\mathbf{y}'$



## Example: Gaussian elimination

- Step 1** Locate the leftmost column of  $A$  that does not consist entirely of zeros, if one exists. The top nonzero entry of this column is the pivot.
- Step 2** Interchange the top row with the pivot row, if necessary, so that the entry at the top of the column found in Step 1 is nonzero.
- Step 3** Divide the top row by pivot in order to introduce a leading 1.
- Step 4** Add suitable multiples of the top row to all other rows so that all entries above and below the leading 1 become zero. Repeat.

## Gaussian elimination

transforms a matrix to its reduced row-echelon form:

$$\mathbf{x} = \begin{pmatrix} x_0 & x_1 \\ x_2 & x_3 \end{pmatrix} \rightarrow \mathbf{y} = \begin{pmatrix} y_0 & y_1 \\ y_2 & y_3 \end{pmatrix}$$

## Gaussian elimination

transforms a matrix to its reduced row-echelon form:

$$\mathbf{x} = \begin{pmatrix} x_0 & x_1 \\ x_2 & x_3 \end{pmatrix} \rightarrow \mathbf{y} = \begin{pmatrix} y_0 & y_1 \\ y_2 & y_3 \end{pmatrix}$$

$$\mathbf{y} = \begin{cases} \begin{pmatrix} 0 & 0 \\ 0 & 0 \end{pmatrix} & \text{if } x_0 = 0 \wedge x_2 = 0 \wedge x_1 = 0 \wedge x_3 = 0 \\ \begin{pmatrix} 0 & 1 \\ 0 & 0 \end{pmatrix} & \text{if } x_0 = 0 \wedge x_2 = 0 \wedge x_1 = 0 \wedge x_3 \neq 0 \\ \begin{pmatrix} 0 & 1 \\ 0 & 0 \end{pmatrix} & \text{if } x_0 = 0 \wedge x_2 = 0 \wedge x_1 \neq 0 \\ \begin{pmatrix} 1 & x_3/x_2 \\ 0 & 0 \end{pmatrix} & \text{if } x_0 = 0 \wedge x_2 \neq 0 \wedge x_1 = 0 \\ \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} & \text{if } x_0 = 0 \wedge x_2 \neq 0 \wedge x_1 \neq 0 \\ \begin{pmatrix} 1 & x_1/x_0 \\ 0 & 0 \end{pmatrix} & \text{if } x_0 \neq 0 \wedge x_3 - x_2(x_1/x_0) = 0 \\ \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} & \text{if } x_0 \neq 0 \wedge x_3 - x_2(x_1/x_0) \neq 0 \end{cases}$$

# Numerical model of Gaussian Elimination

See

- [mpi-spin/examples/gausselim/source/](#)
- [mpi-spin/examples/gausselim/model/](#)