Using Model Checking with Symbolic Execution for the Verification of Data-Dependent Properties of MPI-Based Parallel Scientific Software

Anastasia Mironova

Problem

- It is hard to create “correct” parallel programs
  - Concurrency adds complexity and introduces problems such as deadlock
  - Non-determinacy makes testing even less effective
- Model checking techniques have been applied to concurrent systems
  - But focus on patterns of communication instead of correctness of the computation
  - Limited experience with MPI-based programs

Objective

- Verification of MPI-based programs using model checking
  - Freedom from Deadlock
  - Computational correctness

Approach

- Use a model checker to explore all possible executions
- Deadlock detection
  - Modeling MPI functions
  - Abstracting away unnecessary data
- Computational correctness
  - Extend the model checker to create symbolic representations of the executions
  - Compare the sequential program's symbolic representation of the result to the parallel program's representation

Outline

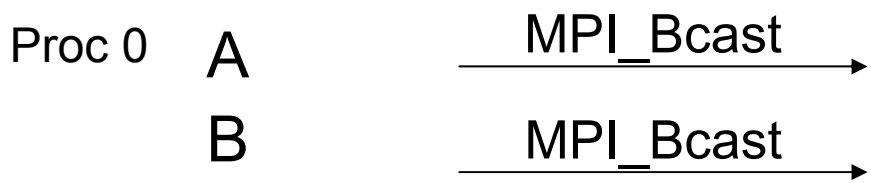
- Background on MPI
- Verification for freedom from deadlock
- Verification using symbolic execution
- Experimental results

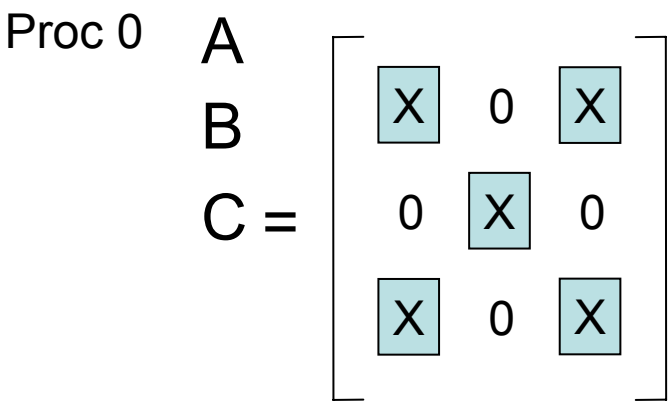
Overview of MPI

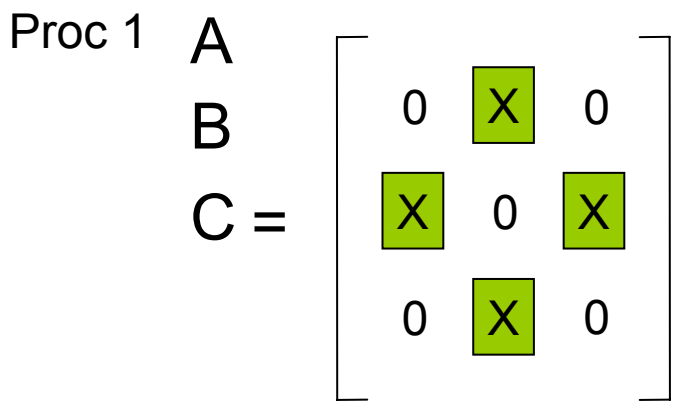
- Origin: result of work of the Committee formed at the Supercomputing ’92 conference
- Significance: widely used in scientific computation
- Communication: message passing in a distributed-memory environment
- Types of operations supported:
  - Blocking/non-blocking
  - Collective

Example MPI Program

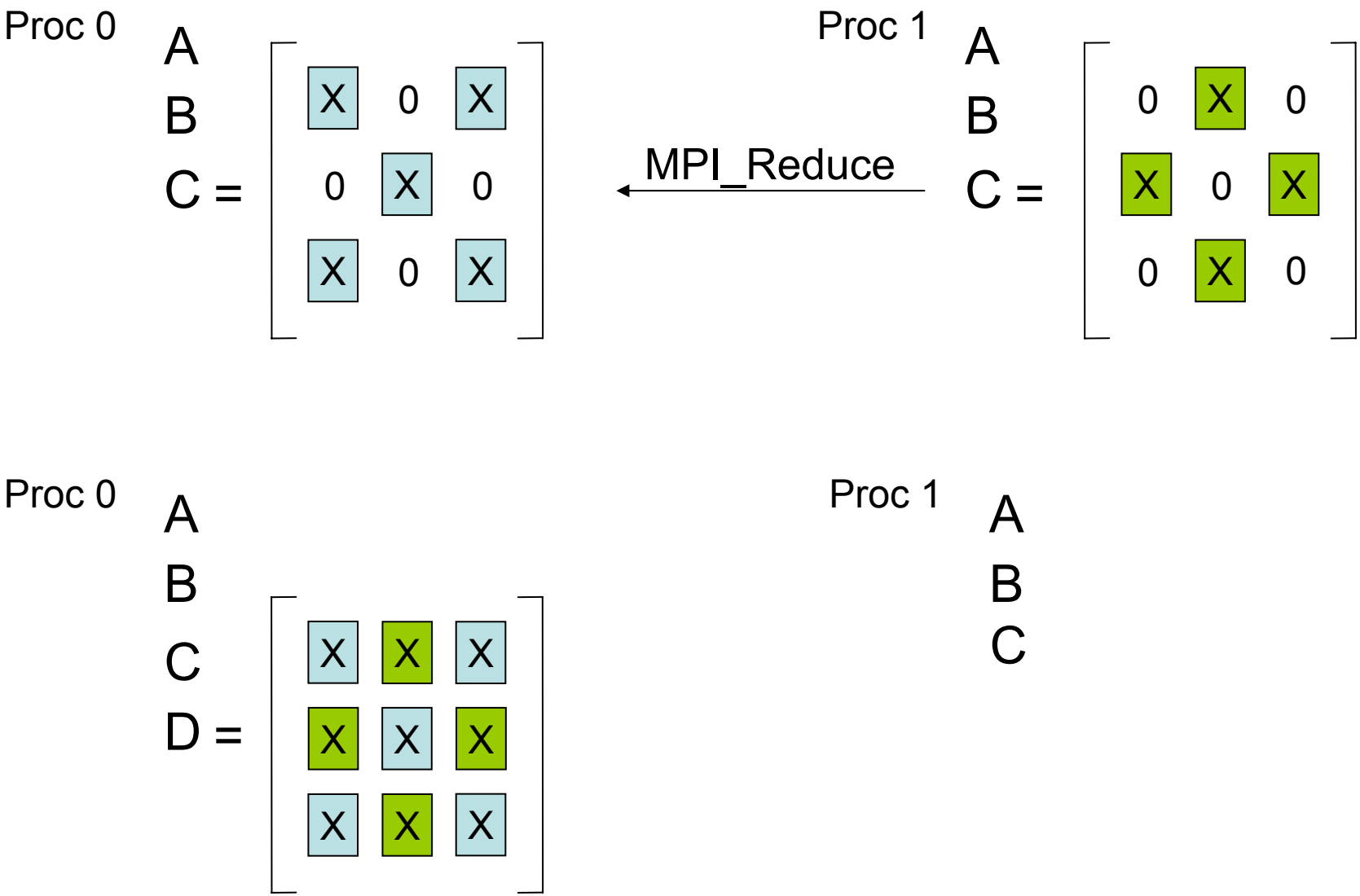
Multiplication of two 3X3 matrices A and B with two processes







Example MPI Program



Example MPI Program: Code

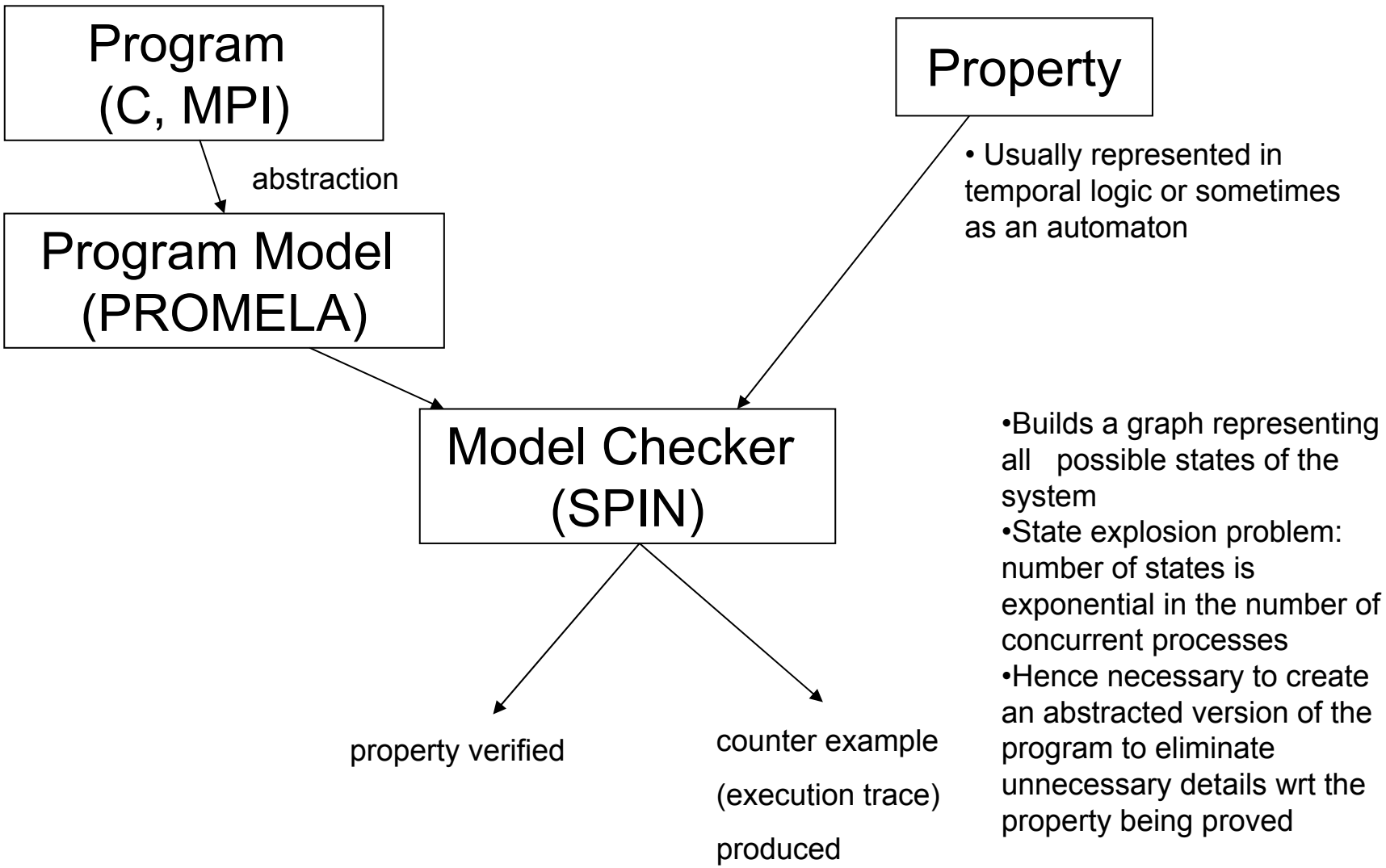
```
double *A, *B;     //two initial matrices
double *C;         //intermediate result
double *D;         //final result, D = AB
/* Read matrix dimensions from file*/
if(rank == 0){
  fscanf(inFile, "%d\n", &n);
  nElts = n*n;
}
/* Broadcast the dimension and the total number of
      elements in each matrix*/
MPI_Bcast(&nElts, 1, MPI_INT, 0, MPI_COMM_WORLD);
MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);
/* Allocate memory for A, B, and C*/
A, B, C = (double *) malloc(nElts * sizeof(double));
/* Root process allocates memory for D and reads
      entries of A and B from file */
if (rank==0){
  D = (double *) malloc(nElts * sizeof(double));
  for(i=0; i<nElts; i++) {
    if (i%n==0) fscanf(inFile, "\n");
    fscanf(inFile, "%lf", &(A, B[i]));
  }
}
```

```
/* Broadcast entries of A and B */
MPI_Bcast(A, nElts, MPI_DOUBLE, 0, MPI_COMM_WORLD);
MPI_Bcast(B, nElts, MPI_DOUBLE, 0, MPI_COMM_WORLD);
/* Computation */
for (i = 0; i<n; i++)
  for (j=0; j<n; j++) {
    C[(i*n)+j]=0;
    if (((i*n)+j)%size==rank) for (k = 0; k < n; k++) C[(i*n)+j] +=
       (double)A[(i*n)+k] * (double)B[(k*n)+j];
  }
}
/* Reduce entries in C to the corresponding entries D on
      the root process*/
for (i = 0; i<nElts; i++)
   MPI_Reduce(&C[i], &D[i], 1, MPI_DOUBLE, MPI_MAX, 0,
      MPI_COMM_WORLD);
```

Verification Process Overview

- Model checking
- Modeling the program
  - MPI functions
  - Deadlock free
  - Computation correctness
- Carrying out the verification

Typical Model Checking Architecture



- Builds a graph representing all possible states of the system
- State explosion problem: number of states is exponential in the number of concurrent processes
- Hence necessary to create an abstracted version of the program to eliminate unnecessary details wrt the property being proved

Example Code

- Matrix Multiplication -

  written by my classmates and myself as one of the assignments for the Programming Languages Course in Computer Science (CS331) at the University of Alaska, Anchorage

- Gauss-Jordan Elimination –

  written at the University of Massachusetts, Amherst under the guidance of Dr. Stephen Siegel

Testing was performed on both of these implementations and based on the results the code was assumed to be correct.

Program Model: MPI

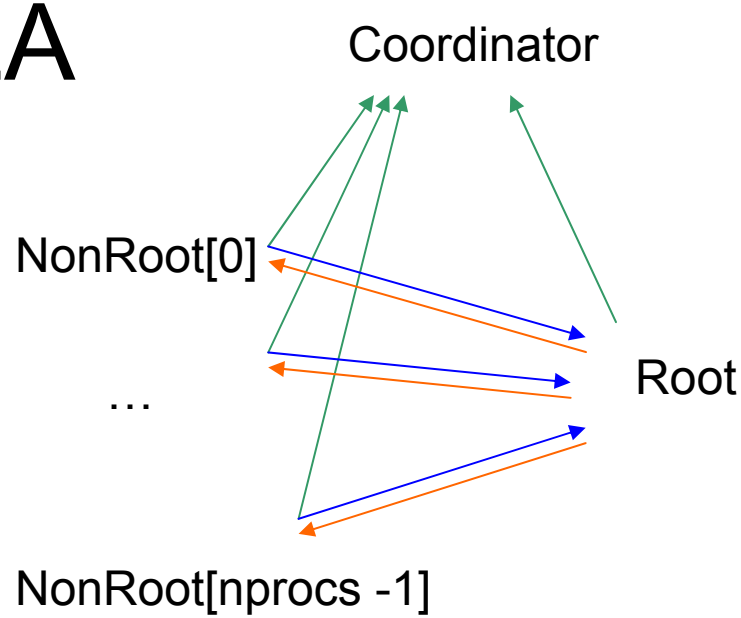
Key Abstractions:

- Processes and communication channels
- Collective MPI functions
- Messages

Processes and Channels in PROMELA

- Definition of Processes:

```
active proc Root{…}
active proc [numprocs - 1] NonRoot{…}
active proc Coordinator{…}
```



- Definition of Channels:

/*Data channels from root to non-root nodes*/

```
chan chan_from_root[nprocs]
```

/*Data Channels from non-root nodes to root */

```
chan chan_to_root[nprocs]
```

/*Collective communication channels from nodes to Coordinator */

```
chan chan_c[nprocs]
```

Modeling MPI Functions

Matrix multiplication example utilizes two MPI functions, both of which are collective:

- **`MPI_Bcast(void *buffer, int count, MPI_Datatype datatype, int root, MPI_Comm comm)`**

  Broadcasts a message from the process with rank **`root`** *to all processes of the group*

- **`MPI_Reduce(void *sendbuf, void *recvbuf, int count, MPI_Datatype datatype, MPI_Op op, int root, MPI_Comm comm)`**

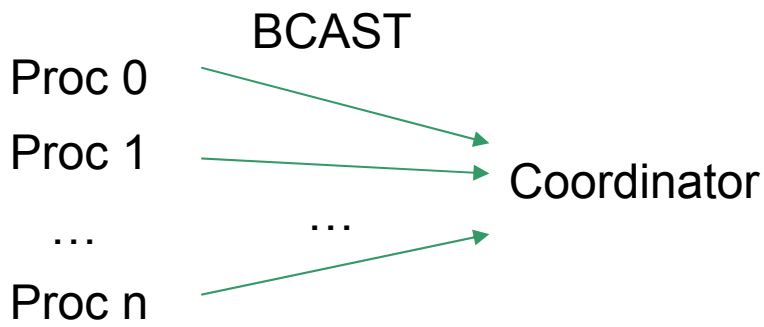
  Combines the elements provided in the input buffer of each process in the group, using the operation **`op`***, and returns the combined value in the output buffer of the process with rank* **`root`**

Based on the description in Siegel and Avrunin, introduce another process for coordinating collective operations…

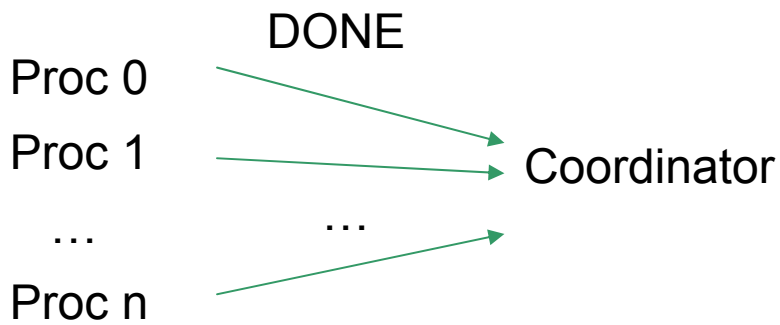
Modeling MPI Functions: Coordinator Process

The Coordinator process is used to model the collective MPI functions MPI_Bcast and MPI_Reduce:

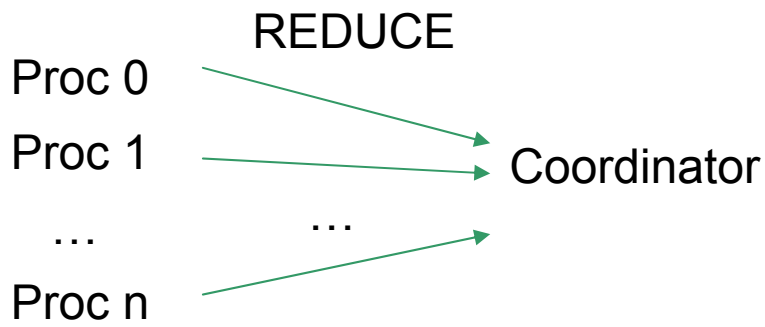
MPI_Bcast



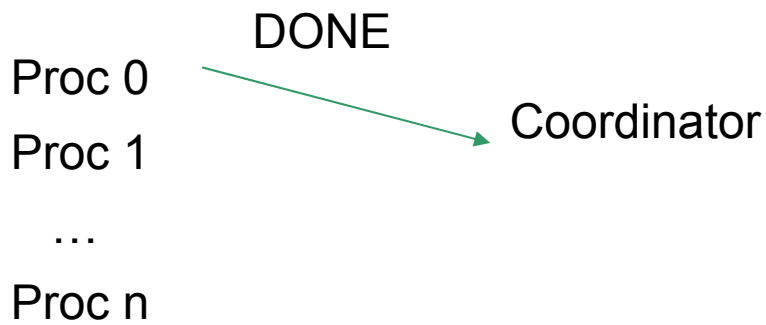
…wait for all processes to receive data from root…



MPI_Reduce



…wait for the root process to collect data from all procs…



Coordinator Process: PROMELA Code

```
active proctype Coordinator() {
  /* Receive initial messages from all processes and confirm that all nodes agree on the type of communication in progress*/
  do
  :: chan_c[0]?im;
    i = 1;
    do
    :: i < nprocs -> chan_c[i]?cm; assert(im == cm);  i++
    :: else -> i = 0; break
    od;
    if
    :: im == BCAST ->
      /* Interpret a broadcast message, receive a DONE message from every node */
      do
      :: i < nprocs -> chan_c[i]?cm; assert (cm == DONE); i++
      :: else -> i = 0; break
      od
    :: im == REDUCE ->
      /* Interpret a reduce message, receive a DONE message from the root process */
         …
    fi
  od
}
```

Messages

Messages between the coordinator and processes participating in the computation are abstracted in the following manner:

```
mtype = {BCAST, REDUCE, DONE},
```

where

- BCAST is a message sent to the Coordinator process by every process participating in the computation at the beginning of all the MPI_Bcast collective routines
- REDUCE is sent to the Coordinator process by every process participating in the computation at the beginning of all the MPI_Reduce collective routines
- DONE is sent to the Coordinator process by processes participating in the computation upon completion of every collective operation as expected by the Coordinator process

Modeling to Check Freedom From Deadlock

- Abstractions to remove details about data

```
mtype = {BCAST, REDUCE, DONE, DATA}
```

Example:

```
MPI_Bcast (0, DATA);
```

- process with rank 0 broadcasting an abstracted data unit to all process participating in the computation

- Synchronization issues

```
channel!message
if
:: 1 -> empty(channel)
:: 1 -> skip
fi
```

Check for Freedom from Deadlock

- Apply SPIN
- No need for special property definition
- Scalability
  - raw
  - utilizing optimizations of SPIN results in extended capabilities

Modeling Computation

- MPI Communication – the same as in the deadlock model
- Add support for symbolic expressions in PROMELA
  - Symbolic expressions

are stored in byte arrays in prefix notation

```
typedef Expn{
  byte length;
  byte array[maxExpnSize]
}
```

$$A = \begin{bmatrix} a_0 & a_1 & a_2 \\ a_3 & a_4 & a_5 \\ a_6 & a_7 & a_8 \end{bmatrix} \quad B = \begin{bmatrix} b_9 & b_{10} & b_{11} \\ b_{12} & b_{13} & b_{14} \\ b_{15} & b_{16} & b_{17} \end{bmatrix}$$

Example:

$a_2 * a_3 + a_4 * a_5$ in infix notation is equivalent to $+*a_2a_3*a_4a_5$ in prefix notation

and to

+	*	2	3	*	4	5

* in the model

* Actually, integer constants represent operators, e.g. `#define 255 PLUS`

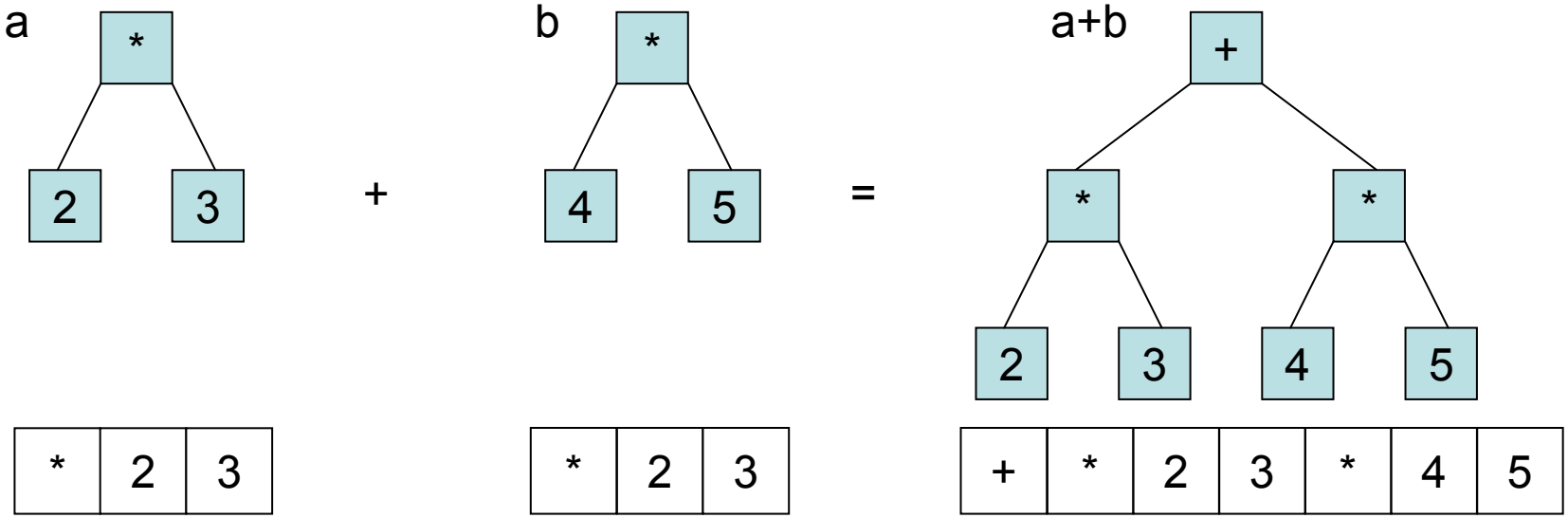
Modeling Computation During Verification

- Add functions to manipulate symbolic expressions

  inline addTo(a, b){…} results in a = a + b

  inline multBy(a, b){...} results in a = a * b

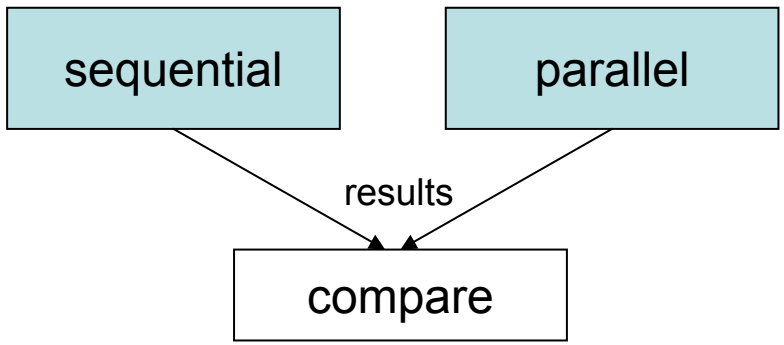
Example:



Validating the Results

Key Idea:

1. Implement both the sequential and parallel versions of the algorithm
2. Apply SPIN and compare the symbolic expressions produced
3. Conclude that the result of the parallel computations is correct if it matches the output of the sequential code.



Matrix multiplication example:

1. Symbolic computation is performed in parallel, generating a matrix of expressions on the root process
2. The root process does the symbolic computations sequentially
3. The root process loops through the two resultant structures checking that the two are the same via a set of assertions

More Interesting Example: Gauss-Jordan Elimination

Common application finding a matrix inverse

$$[\mathrm{A} \quad \mathrm{I}] \equiv \begin{bmatrix} a_{11} & \cdots & a_{1n} & 1 & 0 & \cdots & 0 \\ a_{21} & \cdots & a_{2n} & 0 & 1 & \cdots & 0 \\ \vdots & \ddots & \vdots & \vdots & \vdots & \ddots & \vdots \\ a_{n1} & \cdots & a_{nn} & 0 & 0 & \cdots & 1 \end{bmatrix};$$

where I is the identity matrix, to obtain a matrix of the form

$$\begin{bmatrix} 1 & 0 & \cdots & 0 & b_{11} & \cdots & b_{1n} \\ 0 & 1 & \cdots & 0 & b_{21} & \cdots & b_{2n} \\ \vdots & \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & 1 & b_{n1} & \cdots & b_{nn} \end{bmatrix}.$$

$$\mathrm{B} \equiv \begin{bmatrix} b_{11} & \cdots & b_{1n} \\ b_{21} & \cdots & b_{2n} \\ \vdots & \ddots & \vdots \\ b_{n1} & \cdots & b_{nn} \end{bmatrix}$$

is then the inverse of A if such exists and [A I] is in the reduced row-echelon form.

More Interesting Example: Gauss-Jordan Elimination

Definition: reduced row-echelon form

Properties of a matrix in a reduced row-echelon form:

1. Each row contains only zeros until the first non-zero element, which must be a 1
2. As the rows are followed from top to bottom, the first nonzero number occurs further to the right than in the previous row
3. The entries above and below the first 1 in each row must be all 0

Performing the Gauss-Jordan Elimination algorithm on any matrix results in the equivalent reduced row-echelon form of that matrix

Gauss-Jordan Elimination: Example

Step 1. Locate the leftmost column (vertical line) that does not consist entirely of zeros

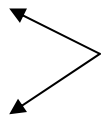
$$\begin{bmatrix} 0 & 0 & -2 & 0 & 7 & 12 \\ 2 & 4 & -10 & 6 & 12 & 28 \\ 2 & 4 & -5 & 6 & -5 & -1 \end{bmatrix}$$

Leftmost nonzero column

Gauss-Jordan Elimination: Example

Step 2. Interchange the top row with another row, if necessary, so that the entry at the top column found in Step 1 is different from zero

$$\begin{bmatrix} 0 & 0 & -2 & 0 & 7 & 12 \\ 2 & 4 & -10 & 6 & 12 & 28 \\ 2 & 4 & -5 & 6 & -5 & -1 \end{bmatrix}$$



Interchange the first and second rows

Gauss-Jordan Elimination: Example

Step 3. If the entry that is now at the top of the column found in Step 1 is *a*, multiply the first row by 1/*a* in order to introduce a leading 1

$$\begin{bmatrix} 1 & 2 & -5 & 3 & 6 & 14 \\ 0 & 0 & -2 & 0 & 7 & 12 \\ 2 & 4 & -5 & 6 & -5 & -1 \end{bmatrix}$$

X 1/2

Gauss-Jordan Elimination: Example

Step 4. Add suitable multiples of the top row to the rows above and below so that all entries below leading 1 become zeros

$$\begin{bmatrix} 1 & 2 & -5 & 3 & 6 & 14 \\ 0 & 0 & -2 & 0 & 7 & 12 \\ 2 & 4 & -5 & 6 & -5 & -1 \end{bmatrix}$$

2 times the first row was added to the third row

Gaussian-Jordan Elimination Algorithm

The above procedure is repeated for every row of the matrix to produce the reduced row-echelon form of the example matrix:

Initial matrix:

$$\begin{bmatrix} 0 & 0 & -2 & 0 & 7 & 12 \\ 2 & 4 & -10 & 6 & 12 & 28 \\ 2 & 4 & -5 & 6 & -5 & -1 \end{bmatrix}$$

Gauss-Jordan Elimination →

Reduced Row-Echelon Form of the initial matrix:

$$\begin{bmatrix} 1 & 2 & 0 & 3 & 0 & 7 \\ 0 & 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & 2 \end{bmatrix}$$

Matrix Multiplication vs. Gauss-Jordan Elimination

Why is it harder to verify the correctness of the Gauss-Jordan Elimination?

- Need to introduce branching when conditions are functions of data
- The property is harder to express since there is no closed formula of the answer, again, consequence of data dependencies

Example of Symbolic Gauss-Jordan Elimination on a 2X2 matrix

$$A = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix}$$

Case 1:

$a_{11} = 0$
$a_{21} = 0$
$a_{12} = 0$
$a_{22} = 0$

$$\begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix}$$

Case 2:

$a_{11} = 0$
$a_{21} = 0$
$a_{12} = 0$
$a_{22} \mathrel{!}= 0$

$$\begin{bmatrix} 0 & 0 \\ 0 & 1 \end{bmatrix}$$

Case 3:

$a_{11} = 0$
$a_{21} = 0$
$a_{12} \mathrel{!}= 0$
a_{22} free

$a_{11} \mathrel{!}= 0$
$a_{22} - a_{21} * a_{12} / a_{11} \mathrel{!}= 0$

$$\begin{bmatrix} 0 & 1 \\ 0 & 0 \end{bmatrix}$$

Case 4:

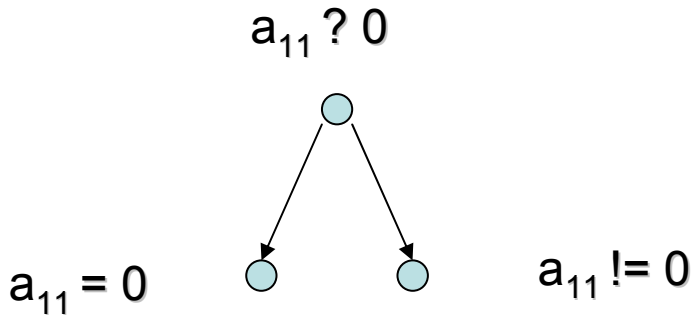
$a_{11} = 0$
$a_{21} \mathrel{!}= 0$
a_{12} free
a_{22} free

$a_{11} \mathrel{!}= 0$
$a_{22} - a_{21} * a_{12} / a_{11} \mathrel{!}= 0$

$$\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$

Dealing with Data Dependency

Exploring Both Branches:



PROMELA Code:

```
if
:: 1 -> /* Assume (a11 == 0) */
    /* Add expression (a11 == 0) to the path conditions table */
    …
:: 1 ->  /* Assume (a11 != 0) */
    /* Add expression (a11 != 0) to the path conditions table */
     …
fi
```

Experimental Evaluation

- Matrix Multiplication Example

  Matrix dimensions: nxn, where n = 2, 3, …, 7

  Number of processes: numprocs = 1, …, 10

- Gauss-Jordan Elimination
  - Sequential

    Matrix sizes: mxn, where m, n = 2, 3, 4
  - Parallel

    Implementation and experimental runs in progress

Measured memory usage

Main issue is the size of the State Vector;

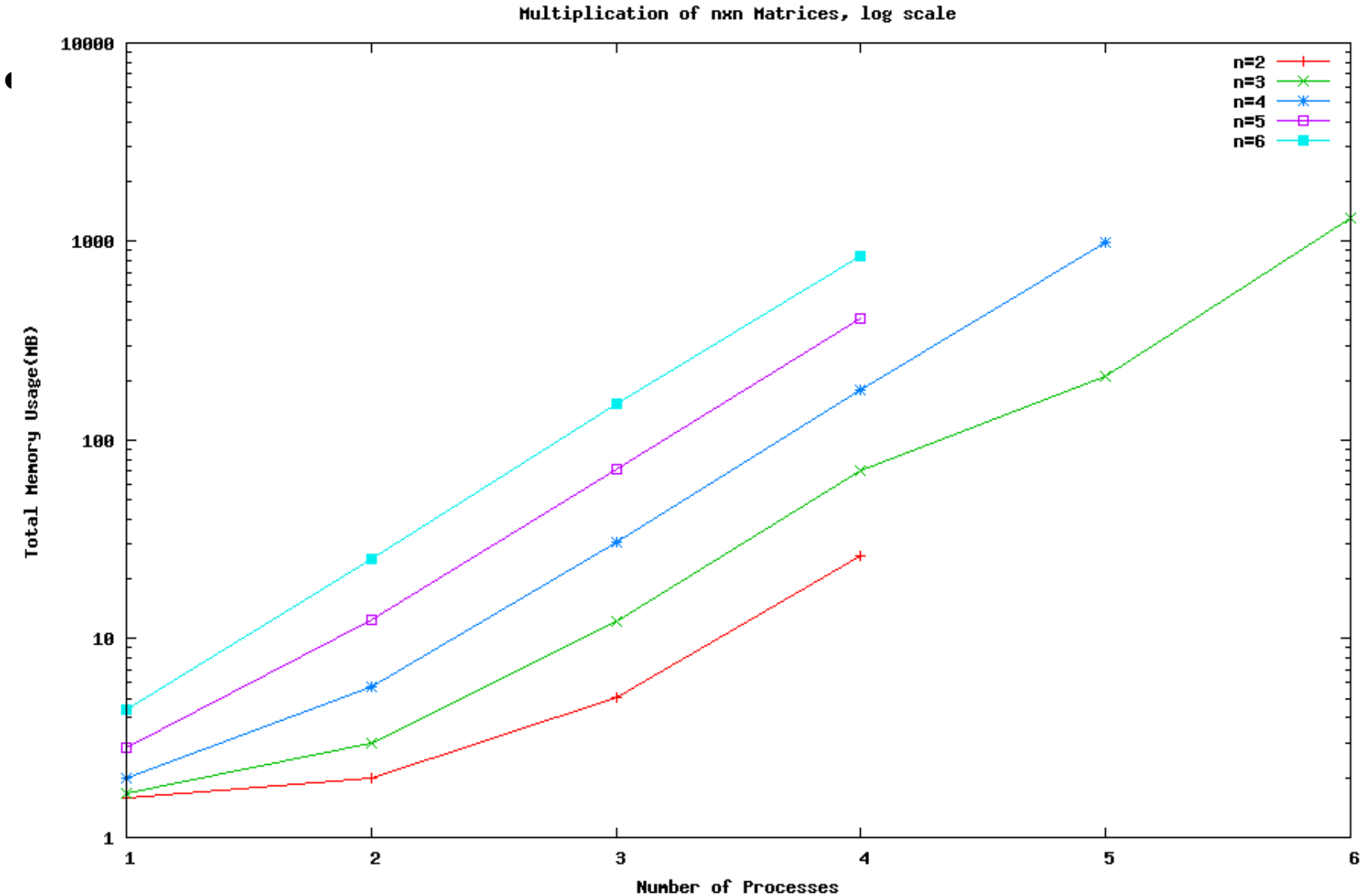
some runs use 85% of the entire machine memory

Distinguishing feature

the common problem of state explosion of

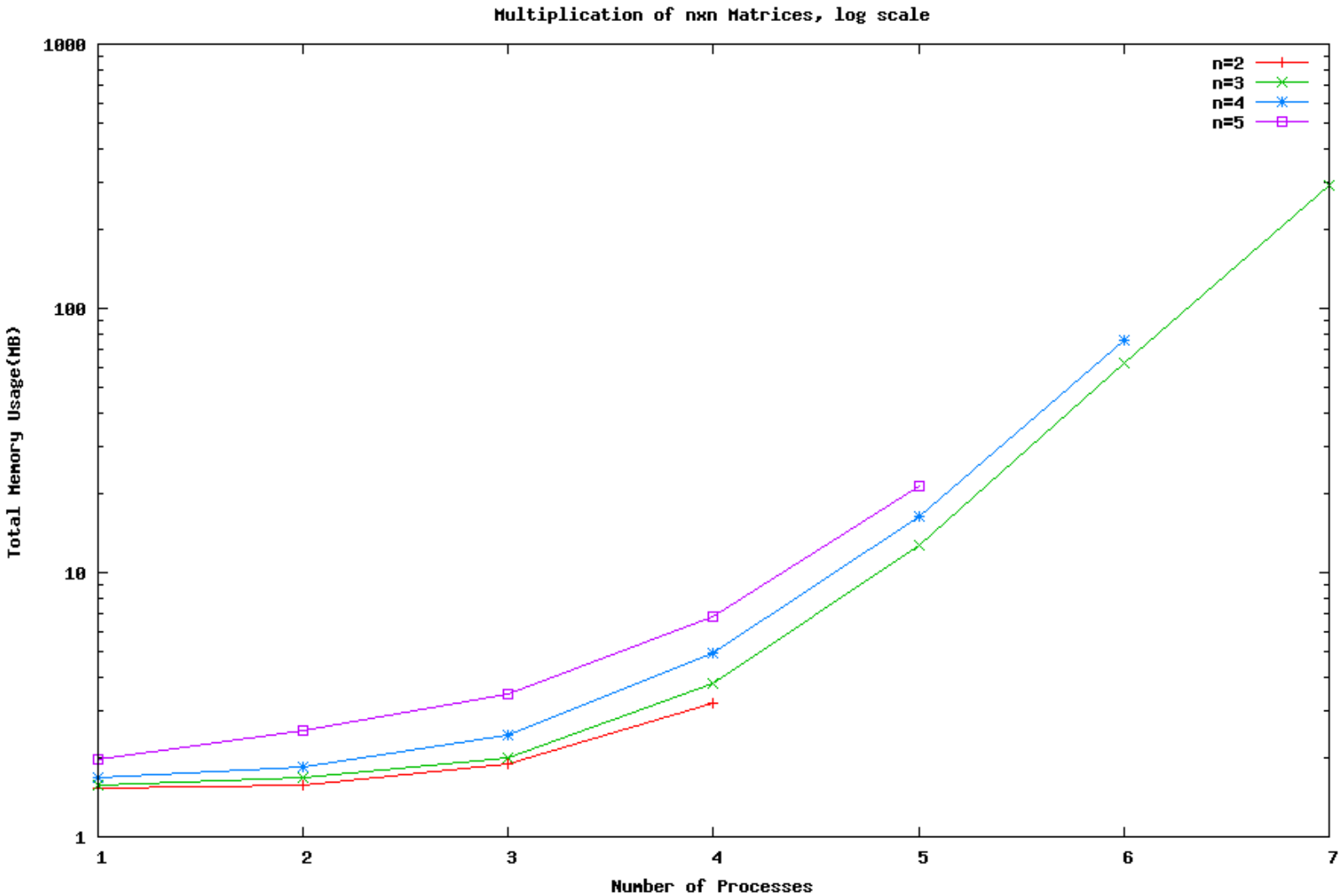
overshadowed by issues associated with the size of the State Vector

Experimental Results: Scalability of the Computation



Experimental Results: Optimization Options in SPIN

-DCOLLAPSE



Future Work

- Improvement of PROMELA models
  - Data structures
  - Incorporating C code
  - Theorem proving packages
- Using a different model checker, e.g. MOVer (MPI-Optimized Verifier)
- Exploring other non-trivial computational examples

Conclusions

- Deadlock
  - demonstrated applicability of abstractions
  - scaled somewhat reasonably – ability to do non-trivial sizes of matrices
- Computational correctness

  - sequential model capable of handling non-trivial sizes of matrices

  - using SPIN to create symbolic expressions and comparing these expressions for the parallel and sequential versions of the algorithm
- Initial experimental results are

  promising!

Gauss-Jordan Elimination: Procedure

Step 1. Locate the leftmost column (vertical line) that does not consist entirely of zeros

$$\begin{bmatrix} 0 & 0 & -2 & 0 & 7 & 12 \\ 2 & 4 & -10 & 6 & 12 & 28 \\ 2 & 4 & -5 & 6 & -5 & -1 \end{bmatrix}$$

Leftmost nonzero column

Gauss-Jordan Elimination: Procedure

Step 2. Interchange the top row with another row, if necessary, so that the entry at the top column found in Step 1 is different from zero

$$\begin{bmatrix} 0 & 0 & -2 & 0 & 7 & 12 \\ 2 & 4 & -10 & 6 & 12 & 28 \\ 2 & 4 & -5 & 6 & -5 & -1 \end{bmatrix}$$

Interchange the first and second rows

Gauss-Jordan Elimination: Procedure

Step 3. If the entry that is now at the top of the column found in Step 1 is *a*, multiply the first row by 1/*a* in order to introduce a leading 1

$$\begin{bmatrix} 1 & 2 & -5 & 3 & 1 & 4 \\ 0 & 0 & -2 & 0 & 7 & 12 \\ 2 & 4 & -5 & 6 & -5 & -1 \end{bmatrix}$$

X 1/2

Gauss-Jordan Elimination: Procedure

Step 4. Add suitable multiples of the top row to the rows above and below so that all entries below leading 1 become zeros

$$\begin{bmatrix} 1 & 2 & -5 & 3 & 6 & 14 \\ 0 & 0 & -2 & 0 & 7 & 12 \\ 2 & 4 & -5 & 6 & -5 & -29 \end{bmatrix}$$

2 times the first row was added to the third row

Background

- Distributed computing

MPI, the Message-Passing Interface (C bindings)

- Symbolic execution of programs with non-trivial arithmetic
  - Matrix multiplication

  Multiplication of two square matrices;
  - Gauss-Jordan elimination

  Computing a reduced row-echelon form of any nxm matrix on n processes
- Model checking

SPIN, Simple PROMELA INterpreter

Matrix Multiplication

Implementation in C using MPI:

- Written as an assignment for CS331 Programming Languages, University of Alaska Anchorage
- Structure and algorithm:
  - Files:

    matrices.txt – contains matrix size and values of all entries in initial matrices

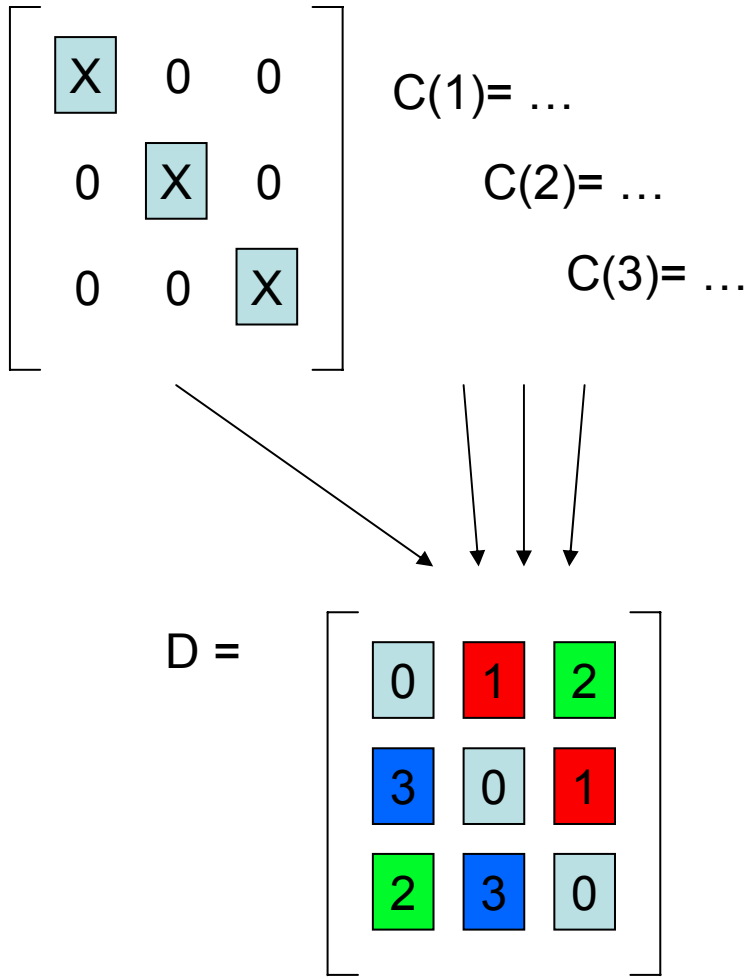
    matrixmul.c – code;
  - Algorithm:
    1. Root process reads the dimensions (n) of the input matrices (A, B) and values of their respective entries
    2. Root process broadcasts n, A, and B to all processes
    3. Each process computes an entry of the resultant matrix represented by a new matrix C if its rank is equal to the remainder of the index of that entry (in row major order)
    4. The final result of the computation is accumulated in a matrix D on the root process by reducing the maximum of the corresponding entries in C from all the processes

Matrixmul.c

```
double *A, *B;     //two initial matrices
double *C;         //intermediate result
double *D;         //final result, D = AB
/* Read matrix dimensions from file*/
if(rank == 0){
  fscanf(inFile, "%d\n", &n);
  nElts = n*n;
}
/* Broadcast the dimension and the total number of elements in each matrix*/
MPI_Bcast(&nElts, 1, MPI_INT, 0, MPI_COMM_WORLD);
MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);
/* Allocate memory for A, B, and C*/
A, B, C = (double *) malloc(nElts * sizeof(double));
/* Root process allocates memory for D and reads entries of A and B from file */
if (rank==0){
  D = (double *) malloc(nElts * sizeof(double));
  for(i=0; i<nElts; i++) {
    if (i%n==0) fscanf(inFile, "\n");
    fscanf(inFile, "%lf", &(A, B[i]));
  }
}
```

Matrixmul.c

```
/* Broadcast entries of A and B */
MPI_Bcast(A, nElts, MPI_DOUBLE, 0,
     MPI_COMM_WORLD);
MPI_Bcast(B, nElts, MPI_DOUBLE, 0,
     MPI_COMM_WORLD);
/* Computation */
for (i = 0; i<n; i++)
  for (j=0; j<n; j++) {
    C[(i*n)+j]=0;
    if (((i*n)+j)%size==rank) for (k = 0; k < n;
      k++) C[(i*n)+j] += (double)A[(i*n)+k] *
      (double)B[(k*n)+j];
  }
}
/* Reduce entries in C to the
     corresponding entries D on the root
     process*/
for (i = 0; i<nElts; i++)
   MPI_Reduce(&C[i], &D[i], 1, MPI_DOUBLE,
     MPI_MAX, 0, MPI_COMM_WORLD);
```



Properties

1. Freedom from deadlock

2. Correctness of computations

Modeling MPI Functions

matrixmul.c utilizes two MPI functions, both of which are collective:

- **`MPI_Bcast`**`(void *buffer, int count, MPI_Datatype datatype, int root, MPI_Comm comm)`

  Broadcasts a message from the process with rank `root` *to all processes of the group, itself included*

- **`MPI_Reduce`**`(void *sendbuf, void * recvbuf, int count, MPI_Datatype datatype, MPI_Op op, int root, MPI_Comm comm)`

  Combines the elements provided in the input buffer of each process in the group, using the operation `op`*, and returns the combined value in the output buffer of the process with rank* `root`

Based on the description in Siegel and Avrunin, ***Modeling MPI Programs for Verification***, introduce another process for coordinating collective operations…

Coordinator Process

```
active proctype Coordinator() {
  /* Receive initial messages from all processes and confirm that all nodes agree on the type
of communication in progress*/
  do
  :: chan_c[0]?im;
    i = 1;
    do
    :: i < nprocs -> chan_c[i]?cm; assert(im == cm);  i++
    :: else -> i = 0; break
    od;
    if
    :: im == BCAST ->
      /* Interpret a broadcast message, receive a DONE message from every non-root node */
      do
      :: i < nprocs -> chan_c[i]?cm; assert (cm == DONE); i++
      :: else -> i = 0; break
      od
    :: im == REDUCE ->
      /* Interpret a reduce message, receive a DONE message from the root process */
         …
    fi
  od
}
```

Abstraction of C Code

Processes in PROMELA:

```
active proc Root{…}
active proc [nprocs - 1] NonRoot{…}
active proc Coordinator{…}
```

/*Data channels from root to non-root nodes*/

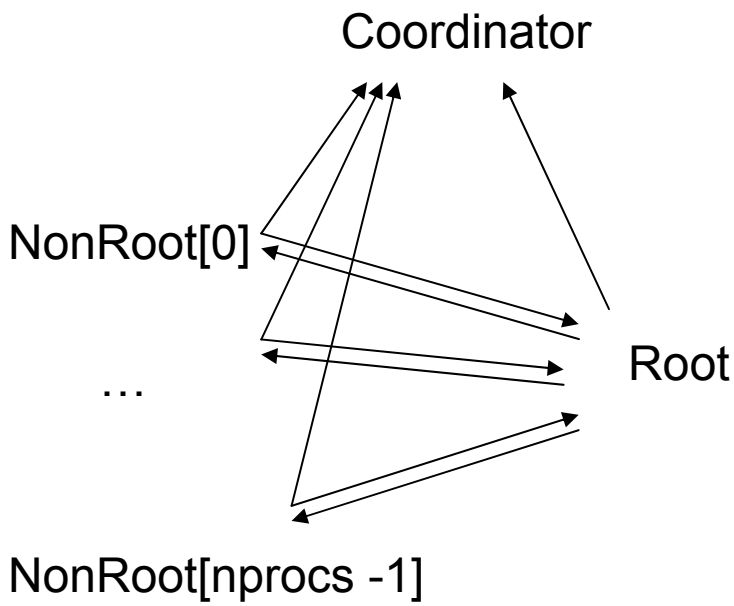
```
chan chan_from_root[nprocs]
```

/*Data Channels from non-root nodes to root */

```
chan chan_to_root[nprocs]
```

/*Collective communication channels from nodes to Coordinator */

```
chan chan_c[nprocs]
```



Deadlock Model

- Message abstraction

```
mtype = {BCAST, REDUCE, DONE, DATA}
```

- Modeling possible need for synchronization of every send operation by implementing a non-deterministic choice

```
if
::1 -> empty(channel)
::1
fi
```

Deadlock Model

```
inline MPI_Bcast(datum, root){
 /* Initiate broadcast */
  chan_c[_pid]!BCAST;
  if
  :: _pid == root ->
    /* Send data to all procs */
    bcastData(datum, root)
  :: else ->
    /* Receive data from root */
    chan_from_root[_pid]?datum
  fi;
 /* Notify coordinator upon completion*/
  chan_c[_pid]!DONE;
  /* Account for a possibility of a blocking
      send */
  if
  ::1 -> empty(chan_c[_pid])
  ::1 ->
  fi
}
```

```
/* Reduce routine executed by the root process*/
inline MPI_ReduceR(root){
  chan_c[0]!REDUCE; /* Initiate reduce */
  …
  do
  :: i < nprocs ->
    if
    :: i != root ->
      /* Receive data from non-root procs */
      chan_to_root[i]?DATA
    :: else ->
    fi;
  …
  od;
  chan_c[_pid]!DONE; /* Notify coordinator upon
completion */
}

/* Reduce routine executed by a non-root
process*/
inline MPI_ReduceNR(root){
  chan_c[_pid]!REDUCE; /* Initiate reduce */
  chan_to_root[_pid]!DATA; /* Send data to root */
  chan_c[_pid]!DONE; /* Notify coordinator upon
completion */
}
```

Deadlock Model

```
/* Root process */
active proctype Root() {
  MPI_Bcast(DATA, 0); /* Broadcast matrix dimension n */
  MPI_Bcast(DATA, 0); /* Broadcast n*n */
  MPI_Bcast(DATA, 0); /* Broadcast A */
  MPI_Bcast(DATA, 0); /* Broadcast B */
 /* Reduce the result */
  do
  :: i < n*n -> MPI_ReduceR(0); i++
  :: else -> i = 0; break
  od
}

/* Non-Root process */
active [nprocs-1] proctype NonRoot() {
  MPI_Bcast(DATA, 0); /* Receive n */
  MPI_Bcast(DATA, 0); /* Receive n*n */
  MPI_Bcast(DATA, 0); /* Receive A */
  MPI_Bcast(DATA, 0); /* Receive B */
  /* Reduce the result to the root node */
  do
  :: i < n*n -> MPI_ReduceNR(0); i++
  :: else -> i = 0; break
  od
}
```

Computation Model

- MPI Communication – the same as in the deadlock model
- Expressions
  - Representation

    Symbolic expressions are stored in byte arrays in prefix notation

```
typedef Expn{
  byte length;
  byte array[maxExpnSize]
}
```

$$A = \begin{bmatrix} a_0 & a_1 & a_2 \\ a_3 & a_4 & a_5 \\ a_6 & a_7 & a_8 \end{bmatrix} \qquad B = \begin{bmatrix} a_9 & a_{10} & a_{11} \\ a_{12} & a_{13} & a_{14} \\ a_{15} & a_{16} & a_{17} \end{bmatrix}$$

  - Symbolic constants represent operators, e.g.

```
#define 255 PLUS
```

Example:

$a_2*a_3 + a_4*a_5$ in infix notation is equivalent to $+*a_2a_3*a_4a_5$ in prefix notation and

to

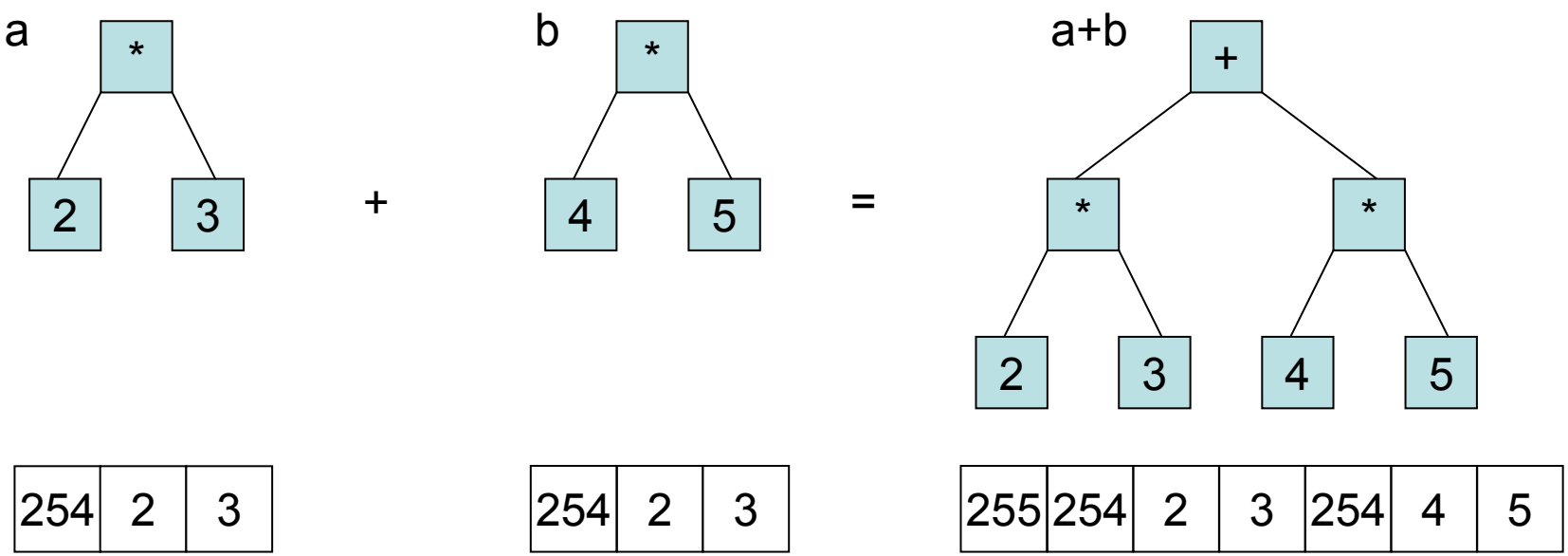
	254	2	3	254	4	5

in the model

Computation Model

- Manipulation of Expressions
  - **inline addTo(a, b){…} results in a = a+b**
  - **inline multBy(a, b){...} results in a = a*b**

Example:



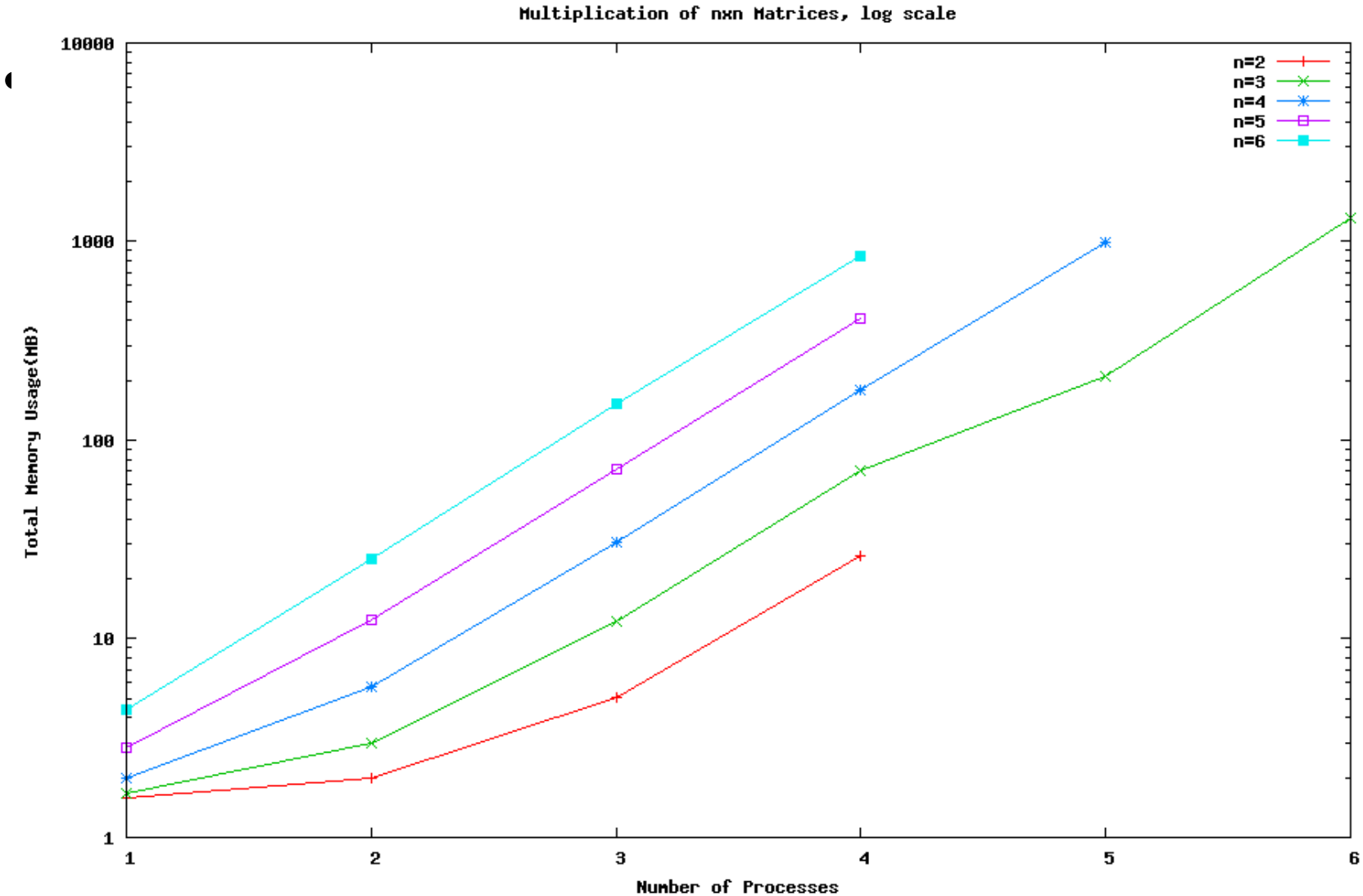
Computation Model

1. Symbolic computation is performed in parallel, generating a matrix of expressions on the root process

2. The root process does the symbolic computations sequentially

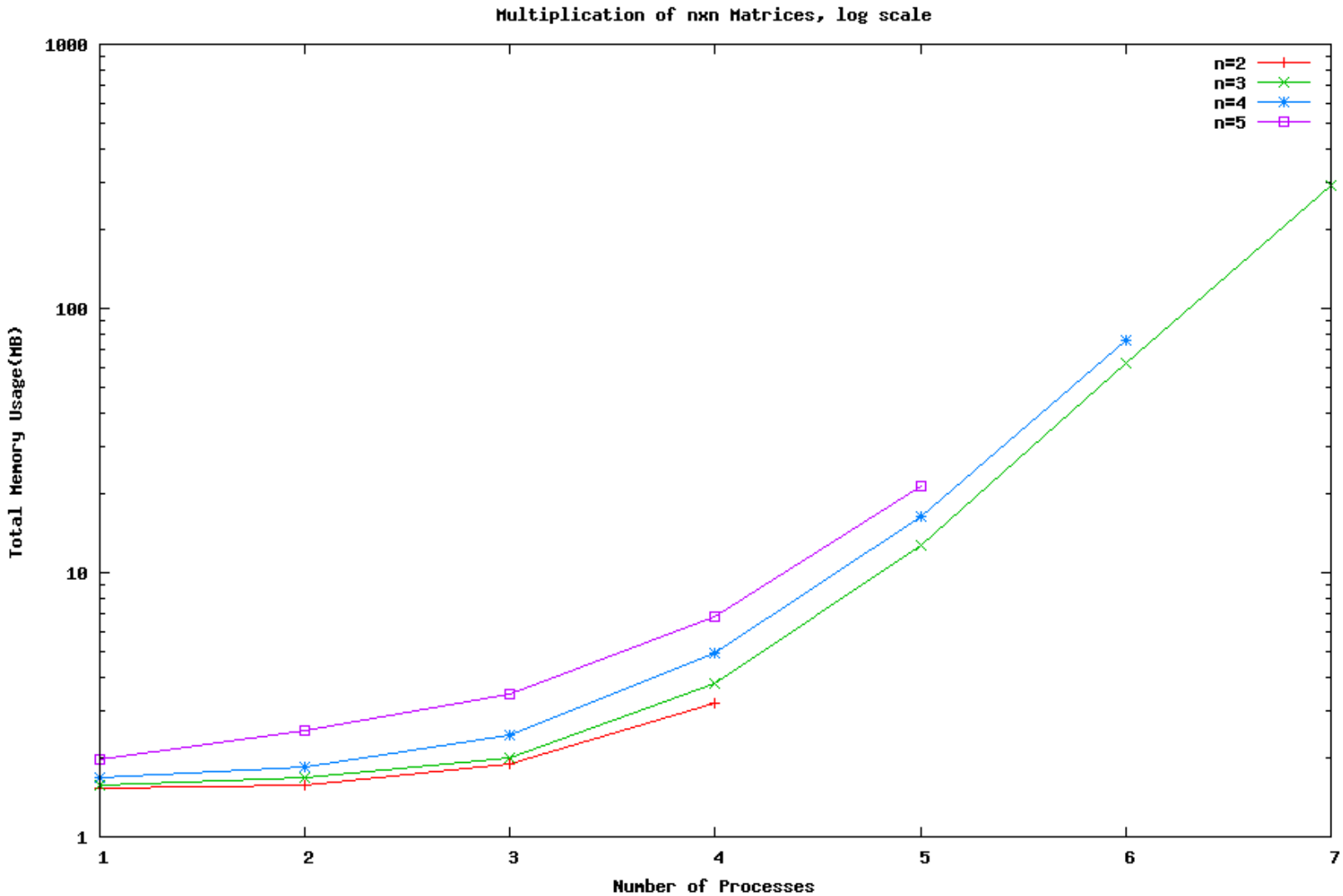
3. The root process loops through the two resultant structures checking that the two are the same via a set of assertions

Scalability: Computation



Optimization options in SPIN

-DCOLLAPSE



Matrixmul summary

- Model 1: Freedom from deadlock

  Verified for 10X10 matrices with 4 processes or 4X4 matrices with 16 processes
- Model 2: Correctness of computation

  Verified that the correct result is produced for all possible executions for 3X3 matrices with 7 processes, …

Gauss-Jordan Elimination

Common application finding a matrix inverse

$$[\mathrm{A} \quad \mathrm{I}] \equiv \begin{bmatrix} a_{11} & \cdots & a_{1n} & 1 & 0 & \cdots & 0 \\ a_{21} & \cdots & a_{2n} & 0 & 1 & \cdots & 0 \\ \vdots & \ddots & \vdots & \vdots & \vdots & \ddots & \vdots \\ a_{n1} & \cdots & a_{nn} & 0 & 0 & \cdots & 1 \end{bmatrix},$$

where I is the identity matrix, to obtain a matrix of the form

$$\begin{bmatrix} 1 & 0 & \cdots & 0 & b_{11} & \cdots & b_{1n} \\ 0 & 1 & \cdots & 0 & b_{21} & \cdots & b_{2n} \\ \vdots & \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & 1 & b_{n1} & \cdots & b_{nn} \end{bmatrix}. \qquad \mathrm{B} \equiv \begin{bmatrix} b_{11} & \cdots & b_{1n} \\ b_{21} & \cdots & b_{2n} \\ \vdots & \ddots & \vdots \\ b_{n1} & \cdots & b_{nn} \end{bmatrix}$$

is then the inverse of A if such exists and [A I] is in the reduced row-echelon form.

Gaussian-Jordan Elimination Algorithm

Repeat for every row of the matrix:

- Locate the leftmost column that does not consist entirely of zeros.
- Interchange the top row with another row, if necessary, so that the entry at the top of the column found in Step 1 in different from zero.
- If the entry that is now at the top of the column found in Step 1 is *a*, multiply the first row by 1/*a* in order to introduce a leading 1.
- Add suitable multiples of the top row to the rows **above and** below so that all entries **above and** below the leading 1 become zero.

Example:

$$\begin{bmatrix} 0 & 0 & -2 & 0 & 7 & 12 \\ 2 & 4 & -10 & 6 & 12 & 28 \\ 2 & 4 & -5 & 6 & -5 & -1 \end{bmatrix} \longrightarrow \begin{bmatrix} 1 & 2 & 0 & 3 & 0 & 7 \\ 0 & 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & 2 \end{bmatrix}$$

Matrix Multiplication vs. Gauss-Jordan Elimination

Why is Gauss-Jordan Elimination harder?

- There is need to introduce branching, where conditions are functions of data
- The property is harder to express since there is no closed formula of the answer, again, consequence of data dependencies

Example of Symbolic Gauss-Jordan Elimination on a 2X2 matrix

$$A = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix}$$

Case 1: $a_{11} = 0$, $a_{21} = 0$, $a_{12} = 0$, $a_{22} = 0$

$$\begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix}$$

Case 2: $a_{11} = 0$, $a_{21} = 0$, $a_{12} = 0$, $a_{22} \mathrel{!}= 0$

$$\begin{bmatrix} 0 & 0 \\ 0 & 1 \end{bmatrix}$$

Case 3: $a_{11} = 0$, $a_{21} = 0$, $a_{12} \mathrel{!}= 0$, a_{22} free

$$\begin{bmatrix} 0 & 1 \\ 0 & 0 \end{bmatrix}$$

Case 4: $a_{11} = 0$, $a_{21} \mathrel{!}= 0$, a_{12} free, a_{22} free

---

$a_{11} \mathrel{!}= 0$, $a_{22} - a_{21} * a_{12} / a_{11} \mathrel{!}= 0$

$$\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$

Case 5: $a_{11} \mathrel{!}= 0$, $a_{22} - a_{21} * a_{12} / a_{11} \mathrel{!}= 0$

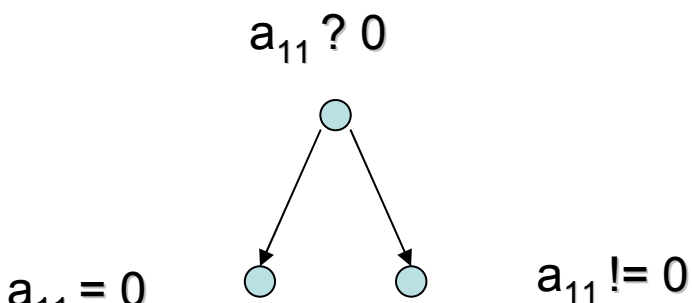
$$\begin{bmatrix} 0 & a_{12}/a_{11} \\ 0 & 0 \end{bmatrix}$$

Approach to Verification

1. Construct both sequential and parallel models
2. Assume that the sequential model algorithm is correct
3. Show that for all input values given to the sequential algorithm, the parallel version will arrive at the same result.

Dealing with branching on data:

The sequential version executes the algorithm first, exploring both possible branches on values of matrix entries, that is



PROMELA:

```
if
:: 1 -> a11 == 0; …
:: 1 -> a11 != 0; …
fi
```

Approach to Verification

Sequential model

1. accumulates a table of path conditions
2. passed on to the parallel code along with the generated expression for the answer

Parallel model

1. uses the path conditions of the table generated by sequential model and checks that the symbolic expressions are the same as the ones obtained sequentially

Gauss-Jordan Elimination Work in Progress

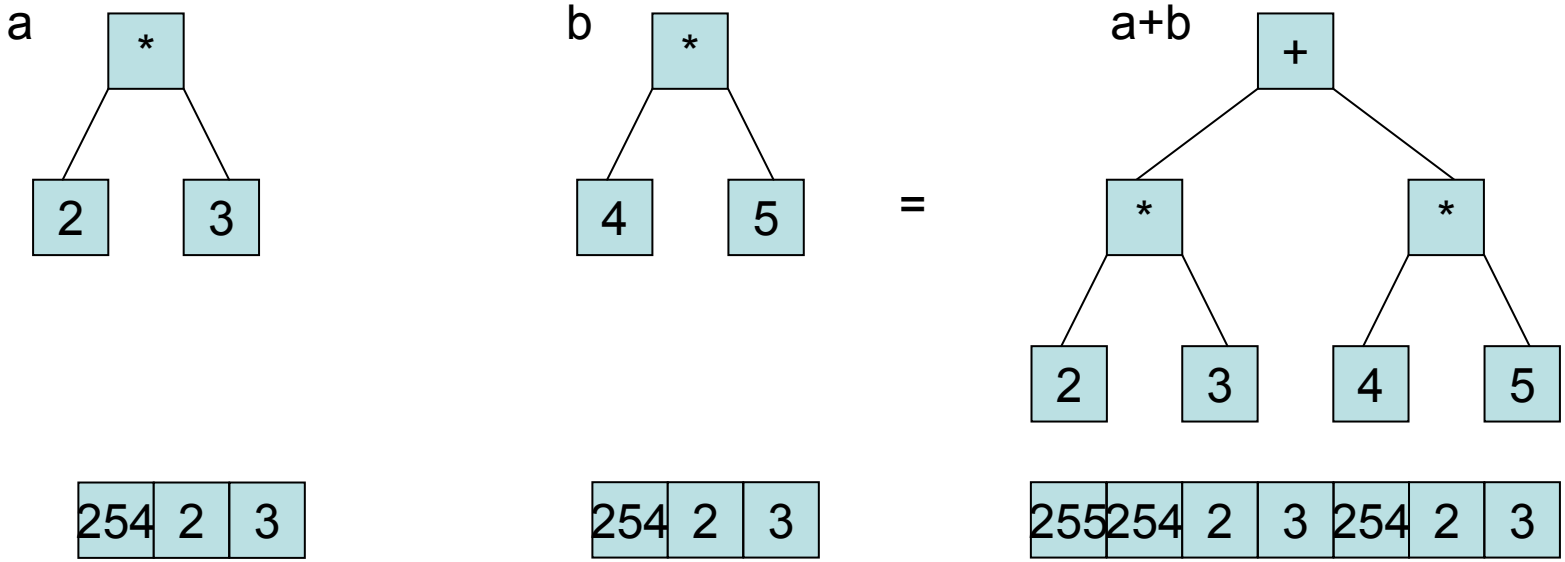
Implemented so far:

- Sequential version in C
- Parallel version in C using MPI
- Sequential version of the SPIN/PROMELA model that explores all possible executions and accumulates a table of expressions of path conditions

Remaining to implement:

- Parallel version of the SPIN/PROMELA model which takes the input data, table of path conditions, and sequentially computed resultant matrix

The End


a
*
2
3
254
2
3
b
*
4
5
254
2
3
=
a+b
+
*
*
2
3
4
5
255
254
2
3
254
2
3

Matrix Multiplication

Definition:

The product C(nxm) of two matrices A(nxp) and B(pxm), is defined by

$$c_{ik} = a_{ij} b_{jk}$$

where *j* is summed over for all possible values of *i* and *k*.

$$\begin{bmatrix} c_{11} & c_{12} & \cdots & c_{1p} \\ c_{21} & c_{22} & \cdots & c_{2p} \\ \vdots & \vdots & \ddots & \vdots \\ c_{n1} & c_{n2} & \cdots & c_{np} \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1m} \\ a_{21} & a_{22} & \cdots & a_{2m} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nm} \end{bmatrix} \begin{bmatrix} b_{11} & b_{12} & \cdots & b_{1p} \\ b_{21} & b_{22} & \cdots & b_{2p} \\ \vdots & \vdots & \ddots & \vdots \\ b_{m1} & b_{m2} & \cdots & b_{mp} \end{bmatrix},$$

Computation PROMELA Code

```
typedef Expression{ /* Symbolic expression in
prefix notation */
  byte length;
  byte array[entrySize]
}
typedef SArray{/*Original matrices. integer
symbols as entries*/
  byte length;
  byte array[matrixSizeSquared]
}
typedef matrix{ /* Resultant matrix of
expressions */
  Expression array[matrixSize2]
}
```

```
/* Root process */
active proctype Root() {
  …
/* D now has the result computed in parallel */
/* Compute the result locally, store in matrix C */
    i = 0;
    j = 0;
    do
    :: i < n ->
      do
      :: j < n ->
      k = 0;
          do
          :: k < n ->
```

```
            mult(A.array[i*n + k], B.array[k*n + j], tmp);
            addTo(C.array[i*n + j], tmp);
            k++
          :: else -> k = 0; break
          od; j++
        ::else -> j = 0; break
        od; i++
      :: else -> i = 0; break
      od;
      /* Verify that D and C are the same */
      i = 0;
      do
      :: i < nElts ->
        assert(D.array[i].length == C.array[i].length);
        ii = 0;
        do
        :: ii < entrySize -> assert(D.array[i].array[ii] == C.array[i].array[ii]);
          ii++
        :: else ii = 0; break
        od;
        i++
      :: else -> i = 0; break
      od;
  }
```

Typical Model Checking Architecture