Verifying Parallel Programs with MPI-SPIN
Part 1:
Introduction and Tool Demonstration

Stephen F. Siegel

Department of Computer and Information Sciences
University of Delaware

EuroPVM/MPI 2007
Paris, France
30 September 2007

Tutorial Overview

1. Introduction and Tool Demonstration
2. Language Basics
3. Using MPI-Spin
4. Verifying Correctness of Numerical Computation

Tutorial Overview

1. Introduction and Tool Demonstration
    1.1 Problems
    1.2 Model checking
    1.3 Diffusion Demo
    1.4 Strengths and Weaknesses
2. Language Basics
3. Using MPI-SPIN
4. Verifying Correctness of Numerical Computation

The Twin Problems

Compared to sequential programs designed to accomplish similar tasks, parallel programs are more. . .

- complex
- difficult to debug
- difficult to understand
- difficult to port
- difficult to test effectively

The Twin Problems

Compared to sequential programs designed to accomplish similar tasks, parallel programs are more. . .

- complex
- difficult to debug
- difficult to understand
- difficult to port
- difficult to test effectively

$$\Downarrow$$

1. **increased development effort**
2. **decreased confidence in correctness**

Specific problems with parallel programs

- they contain race conditions
- they deadlock
- they behave differently on two executions
    - with same input
    - perhaps even on same platform

Nondeterminism

- definition
  - any aspect of program execution not specified by program code
- primary source of nondeterminism in parallel programs
  - numerous ways actions from different processes can be *interleaved*

Sources of nondeterminism in MPI programs

- numerous ways actions of MPI infrastructure can be interleaved with those of processes
    - has request completed?
- `MPI_ANY_SOURCE`
    - which message to select?
- `MPI_Waitany`
    - which request to complete?
- `MPI_Testany`
- `MPI_Testsome`
- `MPI_Waitsome`

Sources of nondeterminism in MPI programs

- numerous ways actions of MPI infrastructure can be interleaved with those of processes
    - has request completed?
- `MPI_ANY_SOURCE`
    - which message to select?
- `MPI_Waitany`
    - which request to complete?
- `MPI_Testany`
- `MPI_Testsome`
- `MPI_Waitsome`
- `MPI_Send`
    - synchronize or buffer?

The limitations of testing

- lack of coverage
    - only a tiny fraction of inputs can be tested

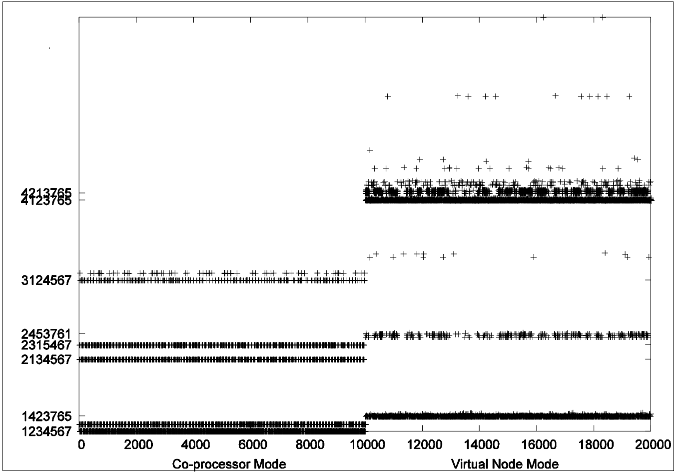
The limitations of testing

- lack of coverage
  - only a tiny fraction of inputs can be tested
- nondeterminism
  - correct result on one execution does not even guarantee correct result on another execution with the same input

The limitations of testing

- lack of coverage
  - only a tiny fraction of inputs can be tested
- nondeterminism
  - correct result on one execution does not even guarantee correct result on another execution with the same input
- problem of oracles
  - in scientific computation, often don't know correct result for a given test input, so can't tell if the observed result is correct

"Bias in Occurrence of Message Orderings"



R. Vuduc, M. Schulz, D. Quinlan, B. de Supinski

Improving distributed memory applications testing by message perturbation

PADTAD'06 (slide from presentation)

Model checking techniques

Three tasks

1. construct a finite-state model of the program
2. formalize correctness properties for the model
3. use automated algorithmic techniques to verify that all executions of the model satisfy the properties

Model checking terminology

- what is a model?
    - a *simplified* or *abstract* version of the program, often written in a *modeling language* for a particular FSV tool
    - abstracts away irrelevant details
    - floating-point variables are usually not used in models

Model checking terminology

- what is a model?
    - a *simplified* or *abstract* version of the program, often written in a *modeling language* for a particular FSV tool
    - abstracts away irrelevant details
    - floating-point variables are usually not used in models
- what is a state of the model?
    - a vector with one component for each variable in the model

Model checking terminology

- what is a model?
    - a *simplified* or *abstract* version of the program, often written in a *modeling language* for a particular FSV tool
    - abstracts away irrelevant details
    - floating-point variables are usually not used in models
- what is a state of the model?
    - a vector with one component for each variable in the model
- what are typical properties of models?
    - freedom from deadlock
    - assertions about the state
        - `assert(x==y*z);`
    - assertions about the order of events (temporal logic)
        - $\Box((\texttt{x==1}) \Rightarrow \Diamond(\texttt{y==1}))$

The reachable state space

- state: a vector s with one component for every variable in the model

The reachable state space

- state: a vector s with one component for every variable in the model
- initial state: the state s_0 for the initial values of the variables

The reachable state space

- state: a vector s with one component for every variable in the model
- initial state: the state s_0 for the initial values of the variables
- next(s): set of all states reachable from s by a single execution step

The reachable state space

- state: a vector s with one component for every variable in the model
- initial state: the state s_0 for the initial values of the variables
- next(s): set of all states reachable from s by a single execution step
- state space: the directed graph with
  - nodes: states
  - edges: $s \longrightarrow t$ iff $t \in \text{next}(s)$

The reachable state space

- state: a vector s with one component for every variable in the model
- initial state: the state s_0 for the initial values of the variables
- $\mathrm{next}(s)$: set of all states reachable from s by a single execution step
- state space: the directed graph with
  - nodes: states
  - edges: $s \longrightarrow t$ iff $t \in \mathrm{next}(s)$
- reachable state space: subgraph G of all states reachable from s_0

The reachable state space

- state: a vector s with one component for every variable in the model
- initial state: the state s_0 for the initial values of the variables
- next(s): set of all states reachable from s by a single execution step
- state space: the directed graph with
  - nodes: states
  - edges: $s \longrightarrow t$ iff $t \in \text{next}(s)$
- reachable state space: subgraph G of all states reachable from s_0
  - can be computed by starting with s_0, computing all next states, computing all next states of those states, ...

The reachable state space

- state: a vector s with one component for every variable in the model
- initial state: the state s_0 for the initial values of the variables
- $\mathrm{next}(s)$: set of all states reachable from s by a single execution step
- state space: the directed graph with
    - nodes: states
    - edges: $s \longrightarrow t$ iff $t \in \mathrm{next}(s)$
- reachable state space: subgraph G of all states reachable from s_0
    - can be computed by starting with s_0, computing all next states, computing all next states of those states, ...
- paths through G correspond to executions of the model

Example: Shared Resource

```
   boolean x;
   proc rw0 {
     while (true) {
0      x := 0;
1      synch();
2      if (x == 0)
3        use_resource();
     }
   }
   proc rw1 {
     while (true) {
0      x := 1;
1      synch();
2      if (x == 1)
3        use_resource();
     }
   }
```

Example: Shared Resource

Property 1: Freedom from deadlock

The program does not deadlock.

```
    boolean x;
    proc rw0 {
      while (true) {
0       x := 0;
1       synch();
2       if (x == 0)
3         use_resource();
      }
    }
    proc rw1 {
      while (true) {
0       x := 1;
1       synch();
2       if (x == 1)
3         use_resource();
      }
    }
```

Example: Shared Resource

Property 1: Freedom from deadlock
The program does not deadlock.

Property 2: Mutual exclusion
It is never the case that both processes use the resource at the same time.

```
  | boolean x;
  | proc rw0 {
  |   while (true) {
0 |     x := 0;
1 |     synch();
2 |     if (x == 0)
3 |       use_resource();
  |   }
  | }
  | proc rw1 {
  |   while (true) {
0 |     x := 1;
1 |     synch();
2 |     if (x == 1)
3 |       use_resource();
  |   }
  | }
```

Example: Shared Resource

Property 1: Freedom from deadlock
The program does not deadlock.

Property 2: Mutual exclusion
It is never the case that both processes use the resource at the same time.

Property 3: Liveness
The resource will eventually be used.

```
   boolean x;
   proc rw0 {
     while (true) {
0      x := 0;
1      synch();
2      if (x == 0)
3        use_resource();
     }
   }
   proc rw1 {
     while (true) {
0      x := 1;
1      synch();
2      if (x == 1)
3        use_resource();
     }
   }
```

Example: Shared Resource

Property 1: Freedom from deadlock
The program does not deadlock.

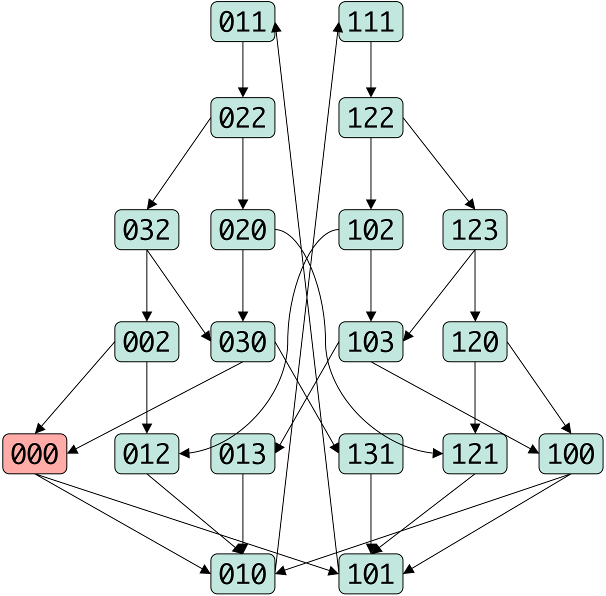
Property 2: Mutual exclusion
It is never the case that both processes use the resource at the same time.

Property 3: Liveness
The resource will eventually be used.

State: $[x, \mathrm{pc}_0, \mathrm{pc}_1]$

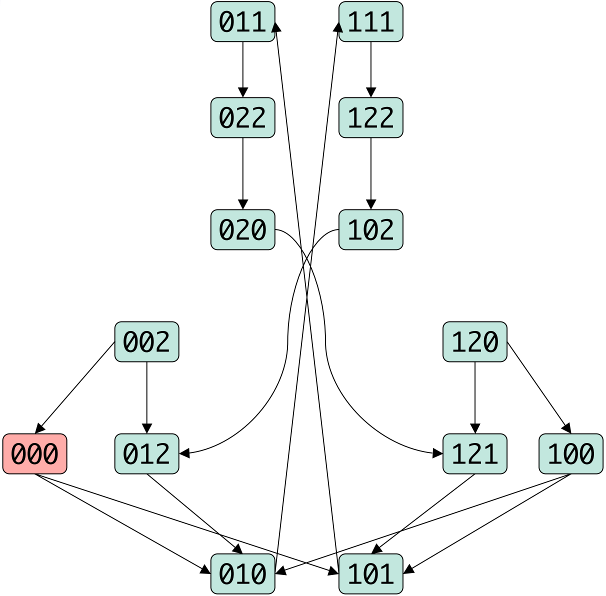
```
  boolean x;
  proc rw0 {
    while (true) {
0     x := 0;
1     synch();
2     if (x == 0)
3       use_resource();
    }
  }
  proc rw1 {
    while (true) {
0     x := 1;
1     synch();
2     if (x == 1)
3       use_resource();
    }
  }
```

Example: Shared Resource



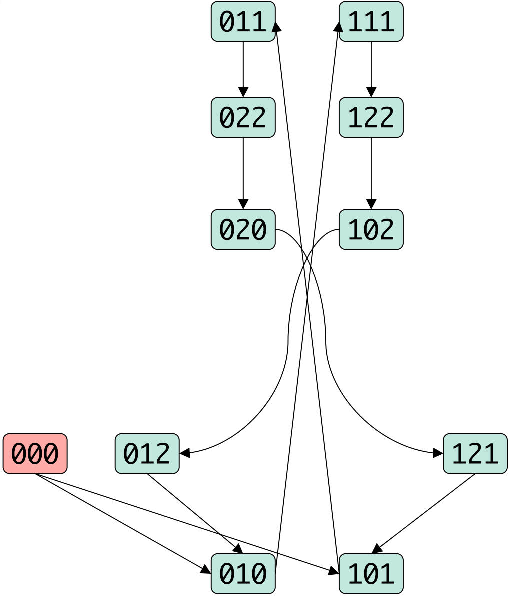
```
   boolean x;
   proc rw0 {
     while (true) {
0      x := 0;
1      synch();
2      if (x == 0)
3        use_resource();
     }
   }
   proc rw1 {
     while (true) {
0      x := 1;
1      synch();
2      if (x == 1)
3        use_resource();
     }
   }
```

Example: Shared Resource



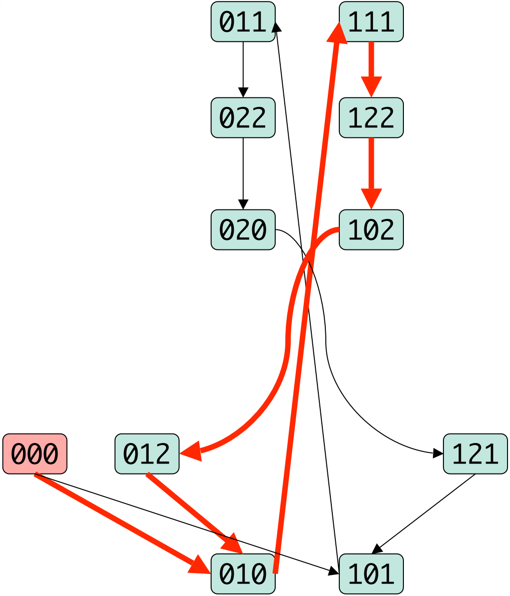
```
  boolean x;
  proc rw0 {
    while (true) {
0     x := 0;
1     synch();
2     if (x == 0)
3       use_resource();
    }
  }
  proc rw1 {
    while (true) {
0     x := 1;
1     synch();
2     if (x == 1)
3       use_resource();
    }
  }
```

Example: Shared Resource



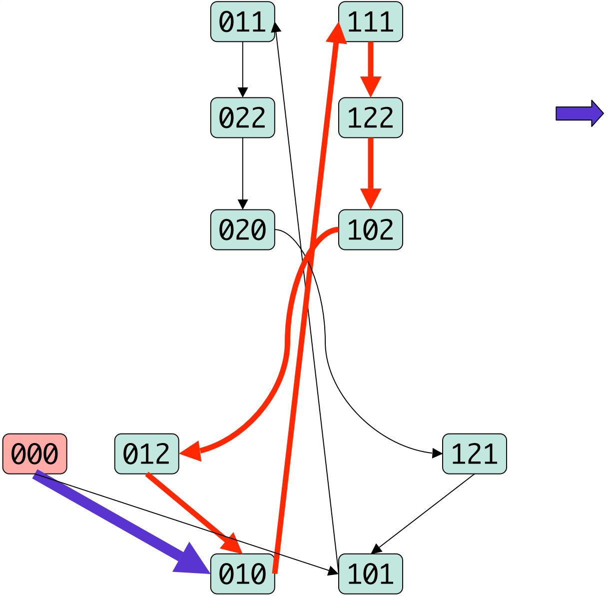
```
    boolean x;
    proc rw0 {
      while (true) {
0       x := 0;
1       synch();
2       if (x == 0)
3         use_resource();
      }
    }
    proc rw1 {
      while (true) {
0       x := 1;
1       synch();
2       if (x == 1)
3         use_resource();
      }
    }
```

Example: Shared Resource



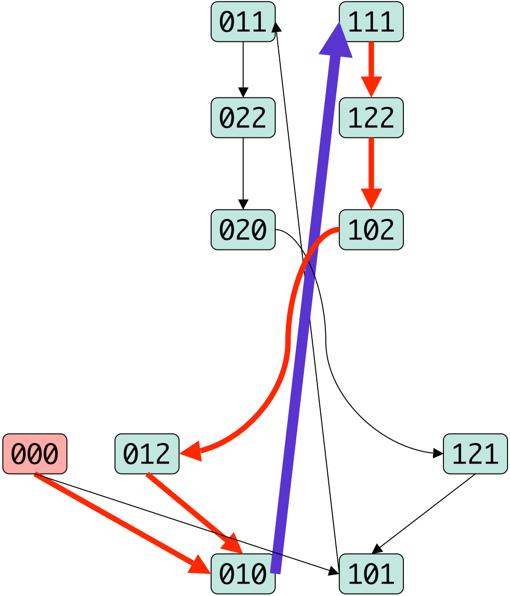
```
    boolean x;
    proc rw0 {
      while (true) {
0       x := 0;
1       synch();
2       if (x == 0)
3         use_resource();
      }
    }
    proc rw1 {
      while (true) {
0       x := 1;
1       synch();
2       if (x == 1)
3         use_resource();
      }
    }
```

Example: Shared Resource



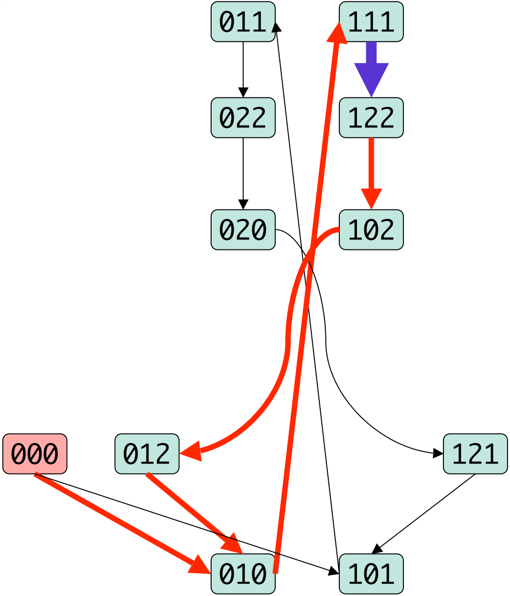
```
  boolean x;
  proc rw0 {
    while (true) {
0     x := 0;
1     synch();
2     if (x == 0)
3       use_resource();
    }
  }
  proc rw1 {
    while (true) {
0     x := 1;
1     synch();
2     if (x == 1)
3       use_resource();
    }
  }
```

Example: Shared Resource



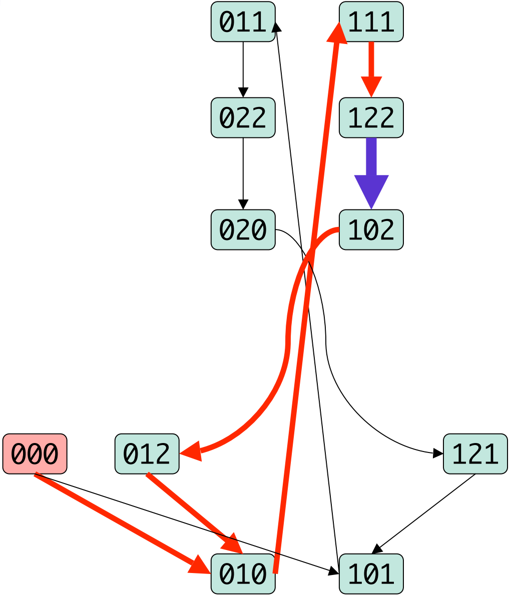
```
    boolean x;
    proc rw0 {
      while (true) {
0       x := 0;
1       synch();
2       if (x == 0)
3         use_resource();
      }
    }
    proc rw1 {
      while (true) {
0       x := 1;
1       synch();
2       if (x == 1)
3         use_resource();
      }
    }
```

Example: Shared Resource



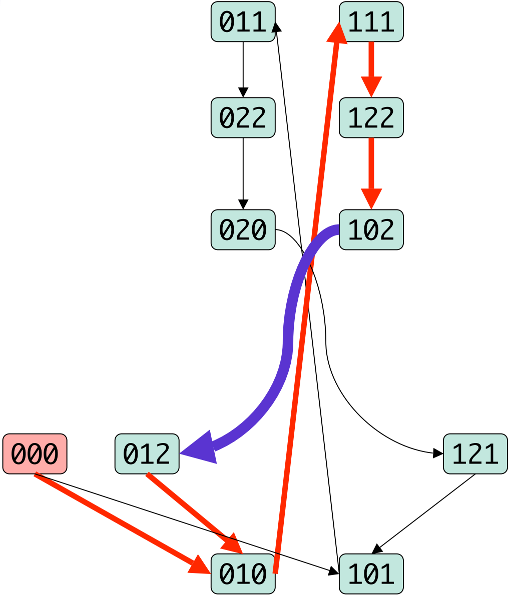
```
    boolean x;
    proc rw0 {
      while (true) {
0       x := 0;
1       synch();
2       if (x == 0)
3         use_resource();
      }
    }
    proc rw1 {
      while (true) {
0       x := 1;
1       synch();
2       if (x == 1)
3         use_resource();
      }
    }
```

Example: Shared Resource



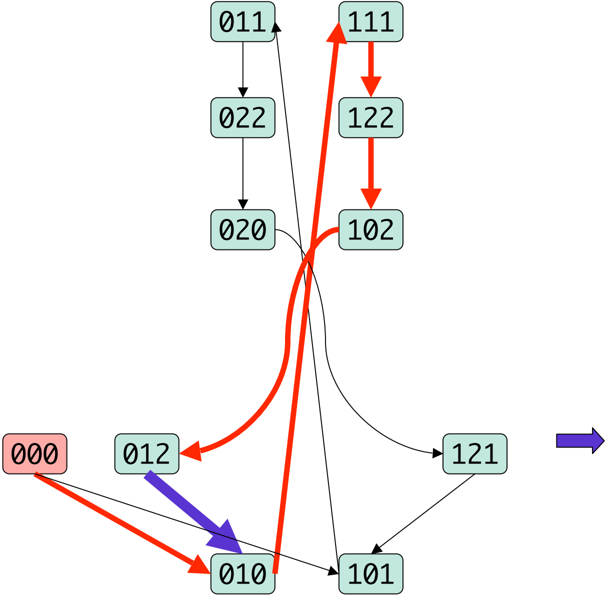
```
    boolean x;
    proc rw0 {
      while (true) {
0       x := 0;
1       synch();
2       if (x == 0)
3         use_resource();
      }
    }
    proc rw1 {
      while (true) {
0       x := 1;
1       synch();
2       if (x == 1)
3         use_resource();
      }
    }
```

Example: Shared Resource



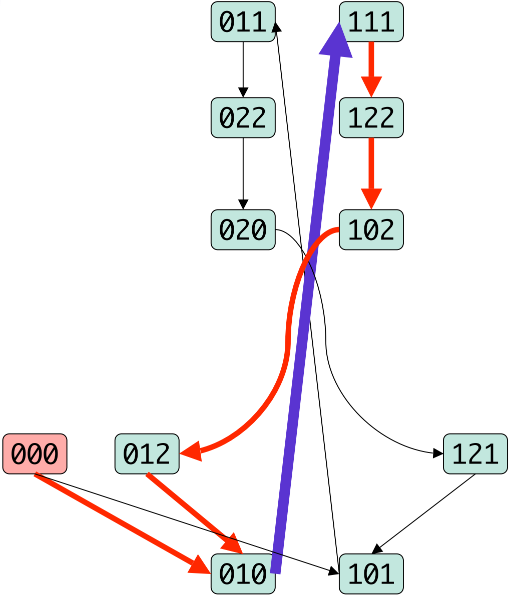
```
  boolean x;
  proc rw0 {
    while (true) {
0     x := 0;
1     synch();
2     if (x == 0)
3       use_resource();
    }
  }
  proc rw1 {
    while (true) {
0     x := 1;
1     synch();
2     if (x == 1)
3       use_resource();
    }
  }
```

Example: Shared Resource



```
    boolean x;
    proc rw0 {
      while (true) {
0       x := 0;
1       synch();
2       if (x == 0)
3         use_resource();
      }
    }
    proc rw1 {
      while (true) {
0       x := 1;
1       synch();
2       if (x == 1)
3         use_resource();
      }
    }
```

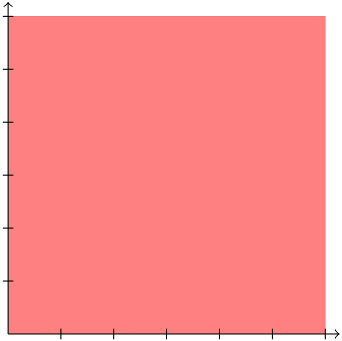
Example: Shared Resource



```
   boolean x;
   proc rw0 {
     while (true) {
0      x := 0;
1      synch();
2      if (x == 0)
3        use_resource();
     }
   }
   proc rw1 {
     while (true) {
0      x := 1;
1      synch();
2      if (x == 1)
3        use_resource();
     }
   }
```

Diffusion2d

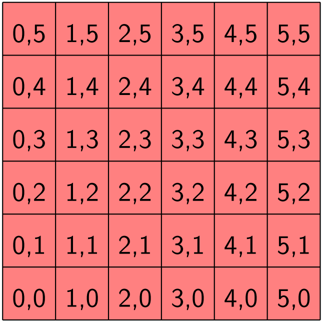
- teacher's solution
  - Andrew Siegel
  - *Applied Parallel Programming*, U. Chicago, Spring 2002
- models evolution of diffusion (heat) equation



$$\frac{\partial u}{\partial t} = D\left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2}\right)$$

Diffusion2d

- teacher's solution
  - Andrew Siegel
  - *Applied Parallel Programming*, U. Chicago, Spring 2002
- models evolution of diffusion (heat) equation

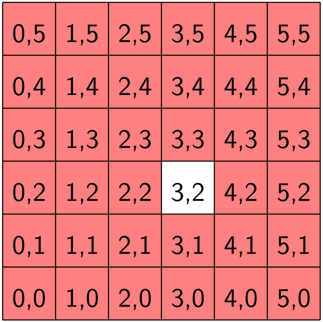


0,5	1,5	2,5	3,5	4,5	5,5
0,4	1,4	2,4	3,4	4,4	5,4
0,3	1,3	2,3	3,3	4,3	5,3
0,2	1,2	2,2	3,2	4,2	5,2
0,1	1,1	2,1	3,1	4,1	5,1
0,0	1,0	2,0	3,0	4,0	5,0

$$\begin{aligned} u^{n+1}(i,j) = u^n(i,j) \\ &+k\big[u^n(i+1,j) + u^n(i-1,j) \\ &+u^n(i,j+1) + u^n(i,j-1) \\ &-4u^n(i,j)\big] \end{aligned}$$

Diffusion2d

- teacher's solution
  - Andrew Siegel
  - *Applied Parallel Programming*, U. Chicago, Spring 2002
- models evolution of diffusion (heat) equation

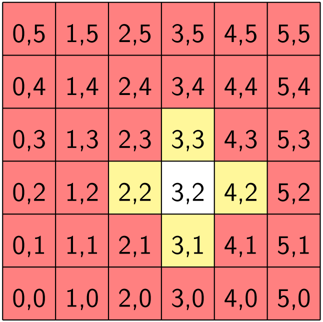


0,5	1,5	2,5	3,5	4,5	5,5
0,4	1,4	2,4	3,4	4,4	5,4
0,3	1,3	2,3	3,3	4,3	5,3
0,2	1,2	2,2	3,2	4,2	5,2
0,1	1,1	2,1	3,1	4,1	5,1
0,0	1,0	2,0	3,0	4,0	5,0

$$
\begin{aligned}
u^{n+1}(i,j) = u^{n}(i,j) & \\
& + k\big[u^{n}(i+1,j) + u^{n}(i-1,j) \\
& + u^{n}(i,j+1) + u^{n}(i,j-1) \\
& - 4u^{n}(i,j)\big]
\end{aligned}
$$

Diffusion2d

- teacher's solution
    - Andrew Siegel
    - *Applied Parallel Programming*, U. Chicago, Spring 2002
- models evolution of diffusion (heat) equation



0,5	1,5	2,5	3,5	4,5	5,5
0,4	1,4	2,4	3,4	4,4	5,4
0,3	1,3	2,3	3,3	4,3	5,3
0,2	1,2	2,2	3,2	4,2	5,2
0,1	1,1	2,1	3,1	4,1	5,1
0,0	1,0	2,0	3,0	4,0	5,0

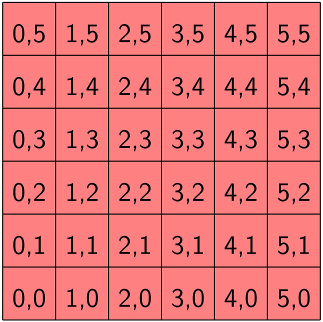
$$
\begin{aligned}
u^{n+1}(i,j) = u^n(i,j)& \\
&+k\big[u^n(i+1,j)+u^n(i-1,j) \\
&+u^n(i,j+1)+u^n(i,j-1) \\
&-4u^n(i,j)\big]
\end{aligned}
$$

Diffusion2d: sequential version

Source code:

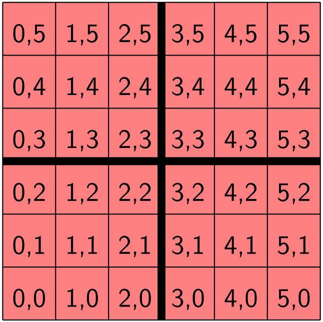
`diffusion/diffusion_seq.c`

Diffusion2d: Parallelization



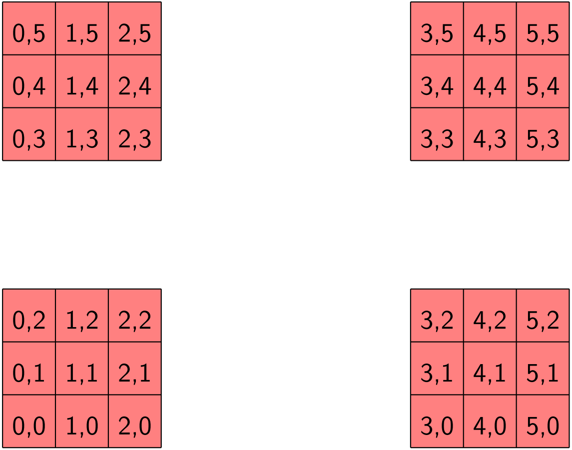
0,5	1,5	2,5	3,5	4,5	5,5
0,4	1,4	2,4	3,4	4,4	5,4
0,3	1,3	2,3	3,3	4,3	5,3
0,2	1,2	2,2	3,2	4,2	5,2
0,1	1,1	2,1	3,1	4,1	5,1
0,0	1,0	2,0	3,0	4,0	5,0

Diffusion2d: Parallelization



0,5	1,5	2,5	3,5	4,5	5,5
0,4	1,4	2,4	3,4	4,4	5,4
0,3	1,3	2,3	3,3	4,3	5,3
0,2	1,2	2,2	3,2	4,2	5,2
0,1	1,1	2,1	3,1	4,1	5,1
0,0	1,0	2,0	3,0	4,0	5,0

Diffusion2d: Distributed Grid



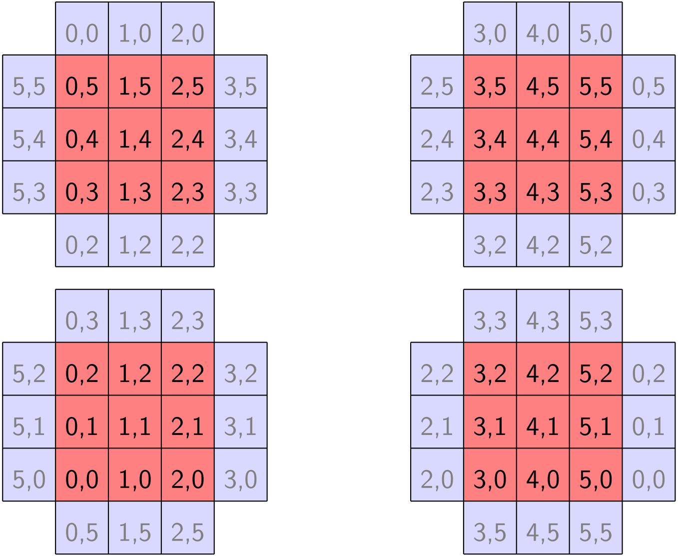
0,5	1,5	2,5
0,4	1,4	2,4
0,3	1,3	2,3

3,5	4,5	5,5
3,4	4,4	5,4
3,3	4,3	5,3

0,2	1,2	2,2
0,1	1,1	2,1
0,0	1,0	2,0

3,2	4,2	5,2
3,1	4,1	5,1
3,0	4,0	5,0

Diffusion2d: Distributed Grid with Ghost Cells



	0,0	1,0	2,0	
5,5	0,5	1,5	2,5	3,5
5,4	0,4	1,4	2,4	3,4
5,3	0,3	1,3	2,3	3,3
	0,2	1,2	2,2	

	3,0	4,0	5,0	
2,5	3,5	4,5	5,5	0,5
2,4	3,4	4,4	5,4	0,4
2,3	3,3	4,3	5,3	0,3
	3,2	4,2	5,2	

	0,3	1,3	2,3	
5,2	0,2	1,2	2,2	3,2
5,1	0,1	1,1	2,1	3,1
5,0	0,0	1,0	2,0	3,0
	0,5	1,5	2,5	

	3,3	4,3	5,3	
2,2	3,2	4,2	5,2	0,2
2,1	3,1	4,1	5,1	0,1
2,0	3,0	4,0	5,0	0,0
	3,5	4,5	5,5	

Diffusion2d: parallel version 1

- source code
  - `diffusion/diffusion_par1.c`

Diffusion2d: parallel version 1

- source code
  - `diffusion/diffusion_par1.c`
- tool demonstration
  - use MPI-SPIN to verify diffusion_par1 is free from deadlock
  - `diffusion/diffusion_dl1.prom`

Diffusion2d: parallel version 2

- `write_frame` version 1
    - proc 0 receives rows in fixed order
    - might block waiting for particular row when data from another proc is available
- optimization: receive data in any order
- use `MPI_ANY_SOURCE`
- insert data into appropriate point in file
    - appropriate point is determined from source field of status object
- `diffusion/diffusion_par2.c`
- `diffusion/diffusion_dl2.prom`

Diffusion2d: parallel version 3

- insert barrier at end of `write_frame`
- `diffusion/diffusion_dl3.prom`

Model checking: strengths

- can prove things about all possible executions of a program
    - all possible inputs
    - all possible interleavings
    - all possible choices available to MPI infrastructure

$$\Downarrow$$

increased confidence in correctness

Model checking: strengths

- can prove things about all possible executions of a program
  - all possible inputs
  - all possible interleavings
  - all possible choices available to MPI infrastructure

$$\Downarrow$$

increased confidence in correctness

- can be (close to) fully automated
- produces a counterexample if property does not hold
  - greatly facilitates debugging

$$\Downarrow$$

decreased development effort

Model checking: limitations

1. the model construction problem
    - the result is only as good as the model
        - model may not accurately reflect some aspect of the program
        - could lead to false confidence

Model checking: limitations

1. the model construction problem
    - the result is only as good as the model
        - model may not accurately reflect some aspect of the program
        - could lead to false confidence
    - but much progress has been made in automatic model extraction
        - Bandera and Bogor (Java)
        - Java PathFinder (Java)
        - Microsoft's SLAM toolset (C)
        - BLAST (C)

Model checking: limitations, cont.

2. state space explosion problem
    - the number of states typically grows exponentially with the number of processes

Model checking: limitations, cont.

2. state space explosion problem
    - the number of states typically grows exponentially with the number of processes
    - but: small scope hypotheses
        - software defects almost always manifest themselves in small configurations
        - very different from the case with testing

Model checking: limitations, cont.

2. state space explosion problem
    - the number of states typically grows exponentially with the number of processes
    - but: small scope hypotheses
        - software defects almost always manifest themselves in small configurations
        - very different from the case with testing
    - methods to combat state explosion
        - partial order reductions (Spin)
        - use of BDDs to represent state space (SMV, NuSMV)
        - symmetry
        - abstraction
        - counterexample-guided refinement

The state of model checking

- wide industial use
    - Intel, Motorola, Microsoft, NEC, . . .
- numerous conferences and workshops
    - SPIN, CAV, . . .
- many tools
- starting to be used for HPC. . .

Model checking for MPI programs

- MPI-Spin (`http://vsl.cis.udel.edu/mpi-spin`)
- Modeling wildcard-free MPI programs for verification
  - Siegel and Avrunin (PPoPP'05)
- Efficient verification of halting properties for MPI programs with wildcard receives
  - Siegel (VMCAI'05)
- Using model checking with symbolic execution to verify parallel numerical programs
  - Siegel, Mirovnova, Avrunin, and Clarke (ISSTA'06)
- Formal verification of programs that use MPI one-sided communication
  - Pervez, Gopalakrishnan, Kirby, Thakur, and Gropp (EuroPVM/MPI'06)
- Practical model checking method for verifying correctness of MPI programs
  - Pervez, Gopalakrishnan, Kirby, Palmer, Thakur, and Gropp (EuroPVM/MPI'07)