Challenge Problems in Verification of MPI Programs

Stephen F. Siegel

Verified Software Laboratory
University of Delaware, USA

FRIDA 2024: the 11th Workshop on Formal Reasoning in Distributed Algorithms
International Conference on Computer Aided Verification (CAV 2024)
Montreal, Canada
July 23, 2024

Outline

1. Problem Statement
2. A Brief Tutorial on MPI
3. Challenge 1 (Deterministic): Parametric Verification of Ghost Cell Exchanges
4. Challenge 2 (Nondeterministic): Verification of Manager-Worker Codes

Message Passing Interface: Background

- MPI = "Message Passing Interface"
- a standardized library for writing message-passing parallel programs
  - in C, C++, or Fortran
- **MPI: A Message Passing Interface Standard**
  - v1.0 (1994), . . . , v4.1 (2023)
  - `https://www.mpi-forum.org`
- universal in scientific/HPC computing
  - weather prediction, climate change
  - design of aircraft, engines, buildings
  - genome sequencing
  - prediction of protein structure
  - certifying nuclear reactor safety
  - monitoring/simulation of nuclear weapons
  - prediction of seismic activity

Univ. Delaware Verified Software Lab
Aurora supercomputer
Argonne National Laboratory, USA

MPI Program Model

MPI Program Model

- an MPI program consists of multiple processes

MPI Program Model

- an MPI program consists of multiple processes
- each process has its own memory (no shared memory)

MPI Program Model

- an MPI program consists of multiple processes
- each process has its own memory (no shared memory)
- think of each process as a program running on its own computer
- the computers can have different architectures
- the programs do not even have to be written in the same language

MPI Program Model

- an MPI program consists of multiple processes
- each process has its own memory (no shared memory)
- think of each process as a program running on its own computer
- the computers can have different architectures
- the programs do not even have to be written in the same language
- however, in most cases:
  - programmer writes one generic program
  - compiles this
  - at run-time, specifies number of processes

MPI Program Model

- an MPI program consists of multiple processes
- each process has its own memory (no shared memory)
- think of each process as a program running on its own computer
- the computers can have different architectures
- the programs do not even have to be written in the same language
- however, in most cases:
  - programmer writes one generic program
  - compiles this
  - at run-time, specifies number of processes
  - run-time system
    - instantiates that number of processes
    - distributes them where they need to go

MPI Program Model

- an MPI program consists of multiple processes
- each process has its own memory (no shared memory)
- think of each process as a program running on its own computer
- the computers can have different architectures
- the programs do not even have to be written in the same language
- however, in most cases:
  - programmer writes one generic program
  - compiles this
  - at run-time, specifies number of processes
  - run-time system
    - instantiates that number of processes
    - distributes them where they need to go
  - a process can obtain its unique ID ("rank")
    - by branching on rank, each process can execute different code

Problem Statement

- Goal: a mechanized way to verify the functional correctness of C/MPI programs

Problem Statement

- Goal: a mechanized way to verify the functional correctness of C/MPI programs
  - This implies a way to specify such programs. Two approaches:
    1. contracts [see my CAV talk tomorrow]
    2. functional equivalence with sequential program

Problem Statement

- Goal: a mechanized way to verify the functional correctness of C/MPI programs
  - This implies a way to specify such programs. Two approaches:
    1. contracts [see my CAV talk tomorrow]
    2. functional equivalence with sequential program
- What I can do now: small scope verification
  - using model checking and symbolic execution techniques
  - place (small) bounds on `nprocs`, input sizes
  - verify assertions, deadlock-freedom, functional equivalence
  - verify a function conforms to its contracts
  - tools: MPI-Spin, TASS, CIVL Model Checker — `https://civl.dev`

Problem Statement

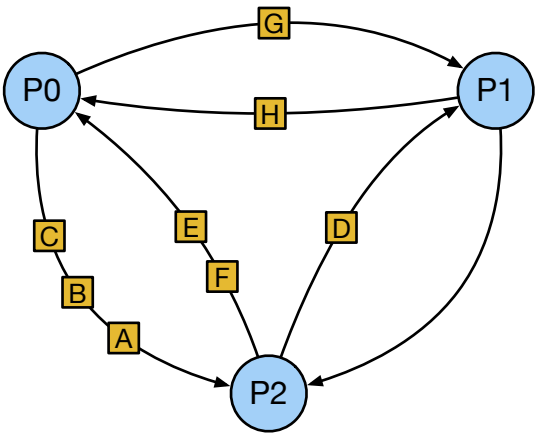
- Goal: a mechanized way to verify the functional correctness of C/MPI programs
  - This implies a way to specify such programs. Two approaches:
    1. contracts [see my CAV talk tomorrow]
    2. functional equivalence with sequential program
- What I can do now: small scope verification
  - using model checking and symbolic execution techniques
  - place (small) bounds on `nprocs`, input sizes
  - verify assertions, deadlock-freedom, functional equivalence
  - verify a function conforms to its contracts
  - tools: MPI-Spin, TASS, CIVL Model Checker — `https://civl.dev`
- What I really want
  - verification for arbitrary number of processes
  - with minimal manual effort (annotations, hints, proof assistant interactions...)

Hello, world

```
#include<stdio.h>
#include<mpi.h>
int main() {
  int rank, nprocs;
  MPI_Init(NULL, NULL);
  MPI_Comm_size(MPI_COMM_WORLD, &nprocs); // get the number of procs
  MPI_Comm_rank(MPI_COMM_WORLD, &rank); // get this proc's PID
  printf("Hello from MPI process %d of %d!\n", rank, nprocs);
  fflush(stdout);
  MPI_Finalize();
}
```

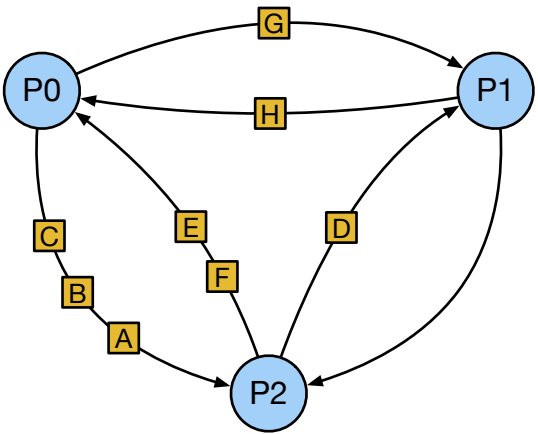
```
> mpicc -o hello.exec hello.c
> mpiexec -n 4 hello.exec
Hello from MPI process 0 of 4!
Hello from MPI process 3 of 4!
Hello from MPI process 1 of 4!
Hello from MPI process 2 of 4!
>
```

Message channels: conceptual framework



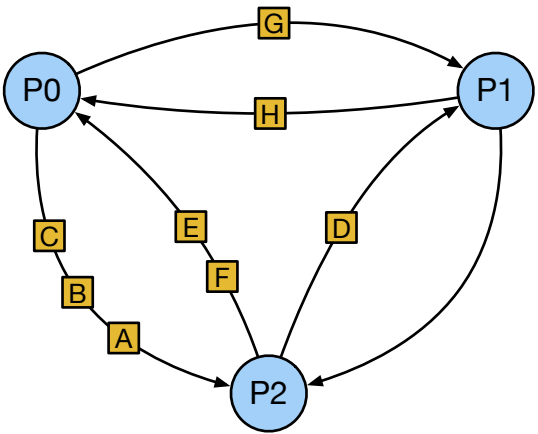
- the state of a communicator with 3 procs

Message channels: conceptual framework



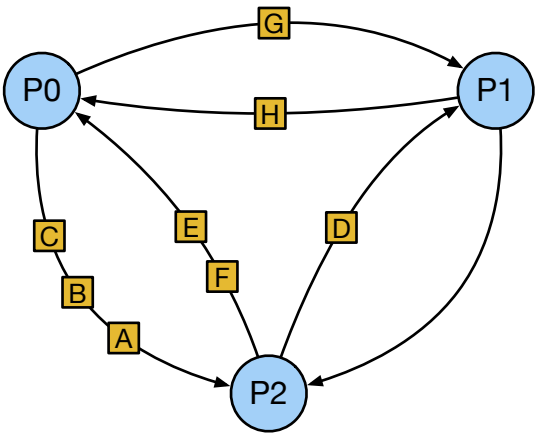
- the state of a communicator with 3 procs
- every communicator is isolated — has its own state
  - messages from one communicator are never picked up by an operation from a different communicator

Message channels: conceptual framework



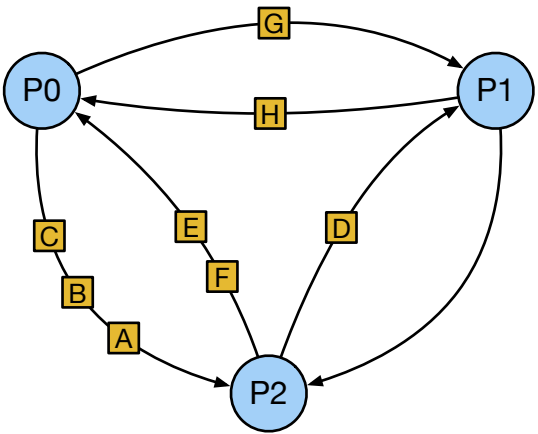
- the state of a communicator with 3 procs
- every communicator is isolated — has its own state
  - messages from one communicator are never picked up by an operation from a different communicator
- between any 2 procs, there is a p2p message channel
  - including from proc to itself (rarely used)

Message channels: conceptual framework



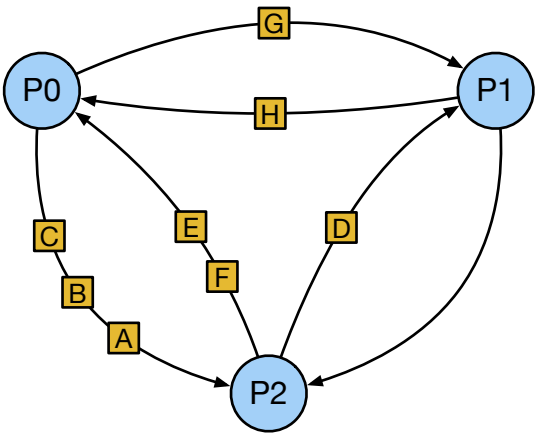
- the state of a communicator with 3 procs
- every communicator is isolated — has its own state
  - messages from one communicator are never picked up by an operation from a different communicator
- between any 2 procs, there is a p2p message channel
  - including from proc to itself (rarely used)
- `send` enqueues message

Message channels: conceptual framework



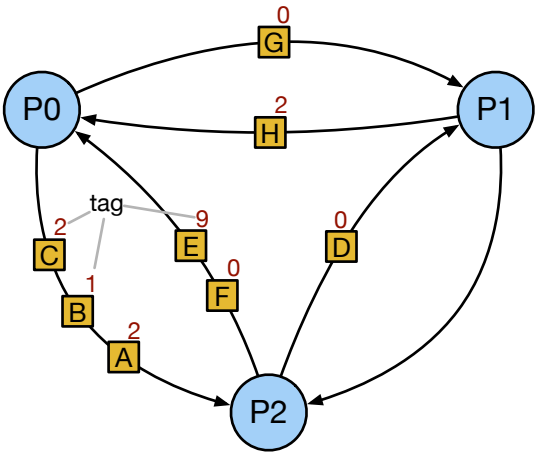
- the state of a communicator with 3 procs
- every communicator is isolated — has its own state
  - messages from one communicator are never picked up by an operation from a different communicator
- between any 2 procs, there is a p2p message channel
  - including from proc to itself (rarely used)
- `send` enqueues message
- `recv` dequeues message

Message channels: conceptual framework



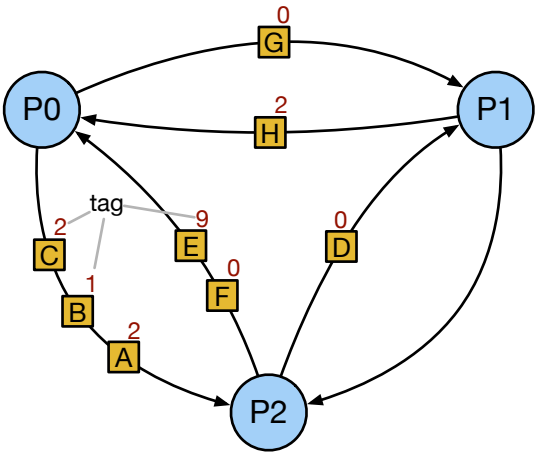
- the state of a communicator with 3 procs
- every communicator is isolated — has its own state
  - messages from one communicator are never picked up by an operation from a different communicator
- between any 2 procs, there is a p2p message channel
  - including from proc to itself (rarely used)
- `send` enqueues message
- `recv` dequeues message
- mostly a FIFO queue

Tags



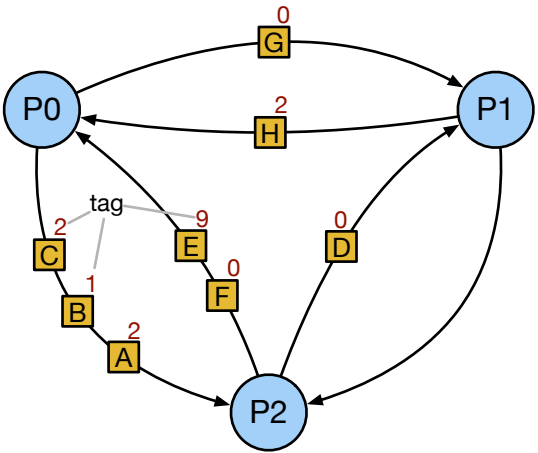
- each message has a tag

Tags



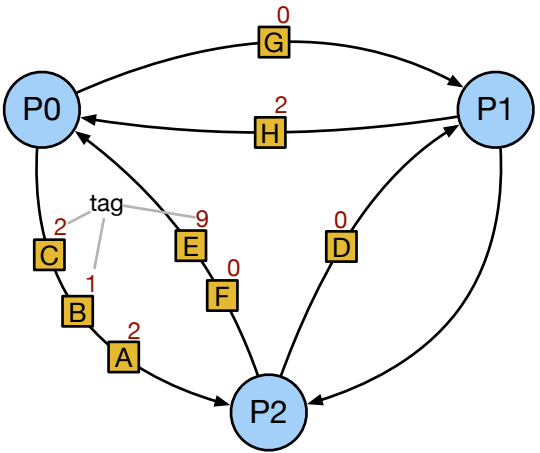
- each message has a tag
- an `int` specified by the sender

Tags



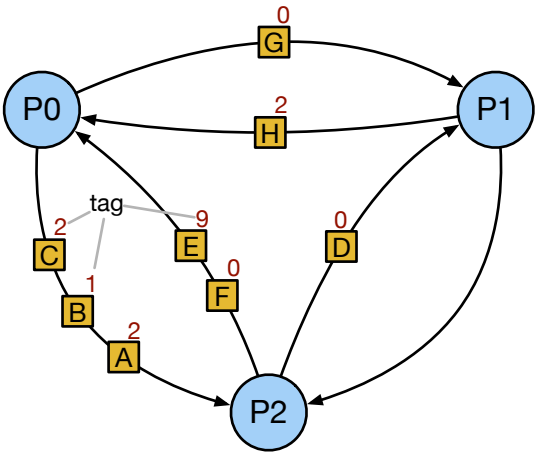
- each message has a tag
- an `int` specified by the sender
- the receiver may specify a tag
  - or can specify "any tag"

Tags



- each message has a tag
- an `int` specified by the sender
- the receiver may specify a tag
  - or can specify "any tag"
- if P2 issues recv from P0 with tag 1
  - P2 will receive message B
  - the first (oldest) message in queue with matching tag

Tags



- each message has a tag
- an `int` specified by the sender
- the receiver may specify a tag
  - or can specify "any tag"
- if P2 issues recv from P0 with tag 1
  - P2 will receive message B
  - the first (oldest) message in queue with matching tag
- if P2 issues recv from P0 with "any tag"
  - P2 will receive message A

MPI_Send

```
MPI_Send(buf, count, datatype, dest, tag, comm)
```

`buf`	address of send buffer (`void*`)
`count`	number of elements in buffer (`int`)
`datatype`	data type of elements in buffer (`MPI_Datatype`)
`dest`	rank of destination process (`int`)
`tag`	integer to attach to message envelope (`int`)
`comm`	communicator (`MPI_Comm`)

MPI_Recv

```
MPI_Recv(buf, count, datatype, source, tag, comm, status)
```

`buf`	address of receive buffer (`void*`)
`count`	number of elements in buffer (`int`)
`datatype`	data type of elements in buffer (`MPI_Datatype`)
`source`	rank of source process (`int`)
`tag`	tag of message to receive (`int`)
`comm`	communicator (`MPI_Comm`)
`status`	pointer to status object (`MPI_Status*`)

MPI_Recv

```
MPI_Recv(buf, count, datatype, source, tag, comm, status)
```

`buf`	address of receive buffer (`void*`)
`count`	number of elements in buffer (`int`)
`datatype`	data type of elements in buffer (`MPI_Datatype`)
`source`	rank of source process (`int`)
`tag`	tag of message to receive (`int`)
`comm`	communicator (`MPI_Comm`)
`status`	pointer to status object (`MPI_Status*`)

- `count` must be at least as large as count of incoming message
  - otherwise, undefined behavior

MPI_Recv

```
MPI_Recv(buf, count, datatype, source, tag, comm, status)
```

`buf`	address of receive buffer (`void*`)
`count`	number of elements in buffer (`int`)
`datatype`	data type of elements in buffer (`MPI_Datatype`)
`source`	rank of source process (`int`)
`tag`	tag of message to receive (`int`)
`comm`	communicator (`MPI_Comm`)
`status`	pointer to status object (`MPI_Status*`)

- `count` must be at least as large as count of incoming message
  - otherwise, undefined behavior
- `status`: object to store envelope information on received message
  - source, tag, count
  - if you don't need it, use `MPI_STATUS_IGNORE`

Example: cyclic exchange: `cycle1.c`

Processes attempt to exchange data in a cycle (ring).
Everyone first sends to their right, then receives from their left.

```
#include<stdio.h>
#include<mpi.h>
int main() {
  int nprocs, rank;
  MPI_Init(NULL, NULL);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  MPI_Comm_size(MPI_COMM_WORLD, &nprocs);
  const int right = (rank + 1)%nprocs, left = (rank + nprocs - 1)%nprocs;
  int rbuf, sbuf = 100 + rank;
  MPI_Send(&sbuf, 1, MPI_INT, right, 0, MPI_COMM_WORLD);
  MPI_Recv(&rbuf, 1, MPI_INT, left, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
  printf("Proc %d: received %d\n", rank, rbuf);
  MPI_Finalize();
}
```

Synchronization and potential deadlock

- `cycle1.c` has a problem
- a send operation may block until a matching receive is called
  - buffer space is finite
- each send may be buffered or may be forced to synchronize
- a correct program will behave correctly regardless of how these decisions are made
- to correct the cyclic exchange. . .
  - good: make even processes send first; odd processes receive first

Synchronization and potential deadlock

- `cycle1.c` has a problem
- a send operation may block until a matching receive is called
  - buffer space is finite
- each send may be buffered or may be forced to synchronize
- a correct program will behave correctly regardless of how these decisions are made
- to correct the cyclic exchange...
  - good: make even processes send first; odd processes receive first
  - better... this situation is so common, MPI provides a function to deal with it
  - `MPI_Sendrecv` combines one send and one receive operation into a single command
  - both operations execute concurrently

MPI_Sendrecv

```
MPI_Sendrecv(sbuf, scount, stype, dest, stag,
             rbuf, rcount, rtype, source, rtag,
             comm, status)
```

`sbuf`	address of send buffer (`void*`)
`scount`	number of elements in send buffer (`int`)
`stype`	data type of elements in sbuf (`MPI_Datatype`)
`dest`	rank of destination process (`int`)
`stag`	integer to attach to message envelope (`int`)
`rbuf`	address of receive buffer (`void*`)
`rcount`	length of receive buffer (`int`)
`rtype`	data type of elements to be received (`MPI_Datatype`)
`source`	rank of sending process (`int`)
`rtag`	tag of message to receive (`int`)
`comm`	communicator (`MPI_Comm`)
`status`	pointer to status object for receive (`MPI_Status*`)

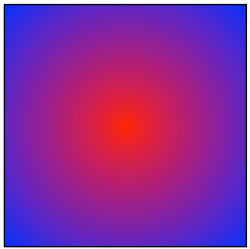
Correct cyclic exchange using MPI_Sendrecv: cycle.c

```
#include<stdio.h>
#include<mpi.h>
int main() {
  int nprocs, rank;
  MPI_Init(NULL, NULL);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  MPI_Comm_size(MPI_COMM_WORLD, &nprocs);
  const int right = (rank + 1)%nprocs, left = (rank + nprocs - 1)%nprocs;
  int rbuf, sbuf = 100 + rank;
  MPI_Sendrecv(&sbuf, 1, MPI_INT, right, 0, &rbuf, 1, MPI_INT, left, 0,
               MPI_COMM_WORLD, MPI_STATUS_IGNORE);
  printf("Proc %d: received %d\n", rank, rbuf);
  MPI_Finalize();
}
```

2-d Diffusion

- a rectangular metal plate
  - initially $100°$
  - temperature on perimeter kept at $0°$
  - over time, heat diffuses out of plate
- $u = u(x, y, t)$ temperature function
- 2d diffusion equation

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



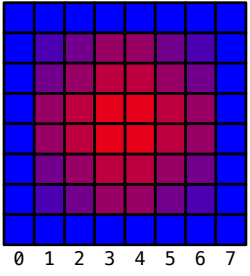
2-d Diffusion

- a rectangular metal plate
  - initially $100°$
  - temperature on perimeter kept at $0°$
  - over time, heat diffuses out of plate
- $u = u(x, y, t)$ temperature function
- 2d diffusion equation

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

- discretization

```
u_new[i][j] =  u[i][j]
  + k*(u[i+1][j] + u[i-1][j]
    + u[i][j+1] + u[i][j-1] - 4*u[i][j]);
```

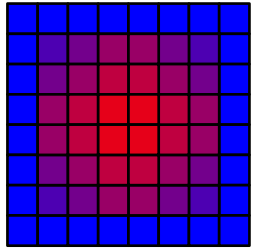


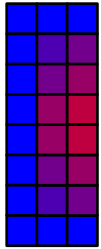
diffusion2d.c: sequential code (excerpt)

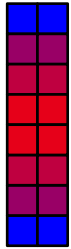
```
int nx, ny;                /* dimensions of the plate */
double k;                  /* constant controlling rate of diffusion */
int nstep;                 /* number of time steps */
double ** u, ** u_new;     /* two copies of temperature function */
...
void update() {
  for (int i = 1; i < nx - 1; i++)
    for (int j = 1; j < ny - 1; j++)
      u_new[i][j] = u[i][j] +
        k*(u[i+1][j] + u[i-1][j] + u[i][j+1] + u[i][j-1] - 4*u[i][j]);
  double ** const tmp = u_new; u_new = u; u = tmp;
}

int main() {
  for (int i = 1; i <= nstep; i++) {
    update();
    write();
  }
}
```

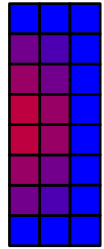
Parallelization of diffusion2d by column distribution

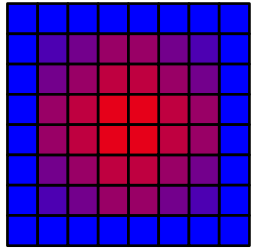
- block distribute the columns of `u` among the processes
- each process updates its columns

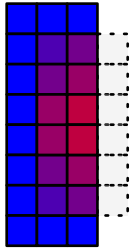

0 1 2 3 4 5 6 7

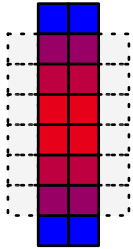

0 1 2

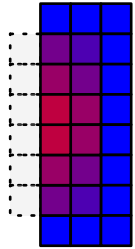

3 4

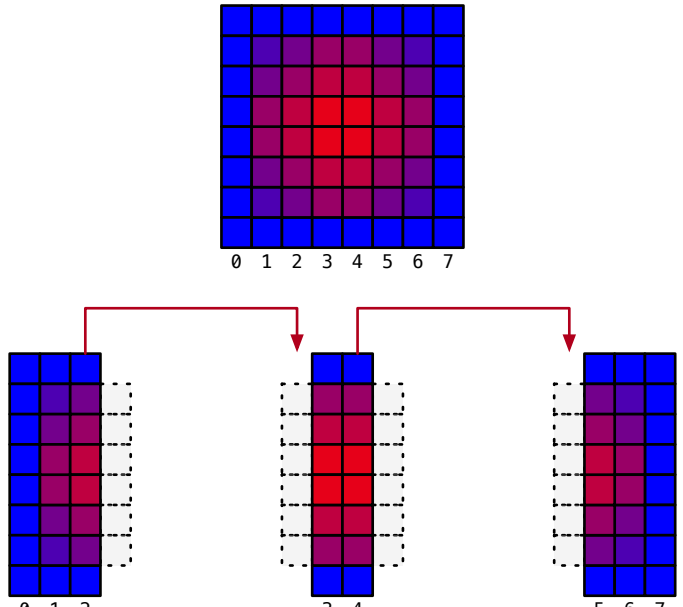

5 6 7

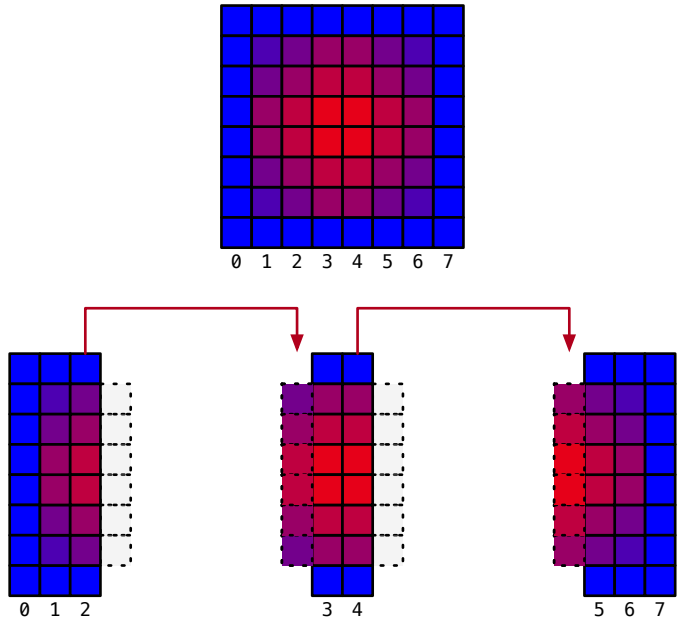

0 1 2 3 4 5 6 7

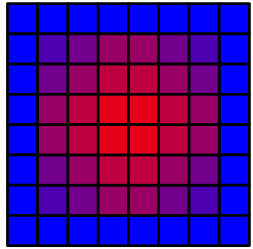

0 1 2

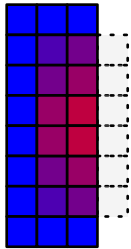

3 4

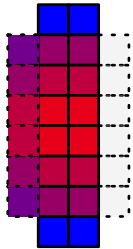

5 6 7

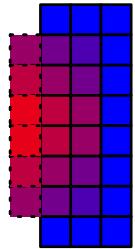

0 1 2 3 4 5 6 7
0 1 2
3 4
5 6 7

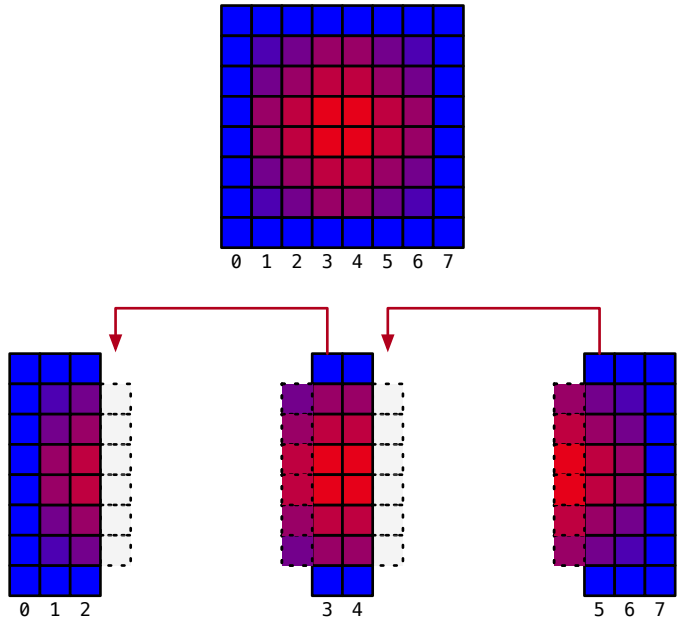

0 1 2 3 4 5 6 7
0 1 2
3 4
5 6 7

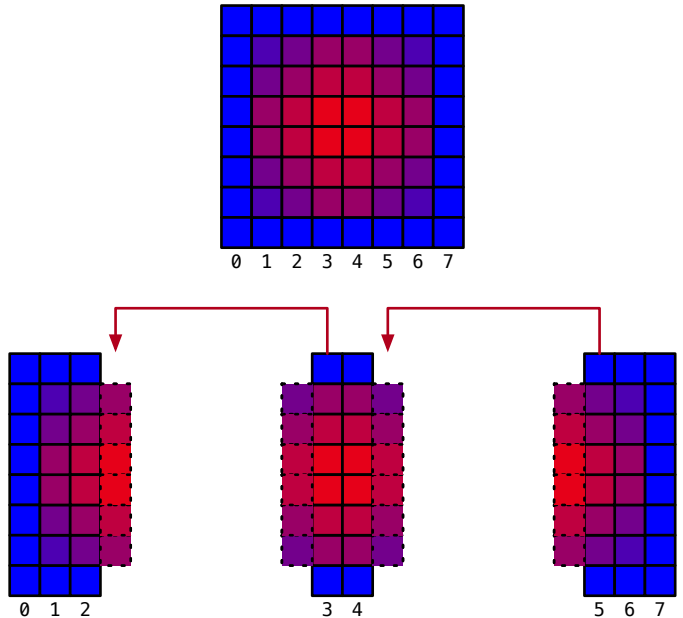

0 1 2 3 4 5 6 7

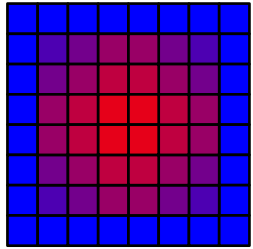

0 1 2

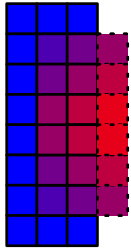

3 4

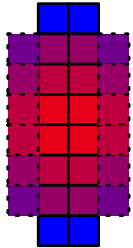

5 6 7

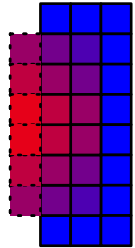

0 1 2 3 4 5 6 7
0 1 2
3 4
5 6 7


0 1 2 3 4 5 6 7
0 1 2
3 4
5 6 7


0 1 2 3 4 5 6 7


0 1 2


3 4


5 6 7

diffusion2d_mpi.c (excerpt)

```
static void exchange_ghost_cells() {
  MPI_Sendrecv(u[1], ny, MPI_DOUBLE, left, 0,
               u[nxl+1], ny, MPI_DOUBLE, right, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
  MPI_Sendrecv(u[nxl], ny, MPI_DOUBLE, right, 0,
               u[0], ny, MPI_DOUBLE, left, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
}

void update() {
  for (int i = start; i <= stop; i++)
    for (int j = 1; j < ny-1; j++)
      u_new[i][j] = u[i][j] + k*(u[i+1][j] + u[i-1][j] + u[i][j+1] + u[i][j-1] - 4*u[i][j]);
  double ** const tmp = u_new; u_new = u; u = tmp;
}

int main() {
  for (int i = 1; i <= nstep; i++) {
    exchange_ghost_cells();
    update();
    write();
  }
}
```

See diffusion2d2.mp4.

Challenge 1

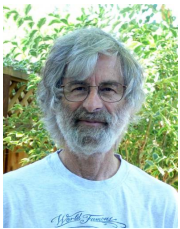
- construct mechanized proof that `diffusion2d_mpi.c` is functionally correct
  - for any `nx`, `ny`, `nprocs`
- correctness is specified by
  - functional equivalence with `diffusion2d.c`, or
  - any other reasonable way (e.g., a contract)

An attempt to verify a simple cyclic exchanger

- Ziqing Luo & I tried to verify a simple message-passing program performing a repeated cyclic exchange
- *Second International Workshop on Software Correctness for HPC Applications* (2018)

```
1 int rank, nprocs, nsteps;
2 double rbuf, sbuf;
3 #define  LEFT(pid) ((pid)>0 ? (pid)-1 : nprocs-1)
4 #define RIGHT(pid) ((pid)<nprocs-1 ? (pid)+1 : 0)
5 ...
6 void exchange() {
7     int t = 0;
8     while (t < nsteps) {
9         send(&sbuf, RIGHT(rank));
10        recv(&rbuf, LEFT(rank));
11        sbuf = rbuf;
12        t++;
13    }
14 }
```

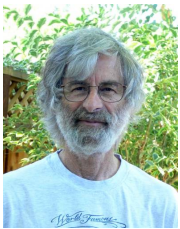
An attempt to verify a simple cyclic exchanger



Owicki and I and most everyone else thought that the Owicki-Gries method was a great improvement over Ashcroft's method because it used the program text to decompose the proof. I've since come to realize that this was a mistake. It's better to write a global invariant. . . . Ashcroft got it right.

— Leslie Lamport

An attempt to verify a simple cyclic exchanger



Owicki and I and most everyone else thought that the Owicki-Gries method was a great improvement over Ashcroft's method because it used the program text to decompose the proof. I've since come to realize that this was a mistake. It's better to write a global invariant. . . . Ashcroft got it right.

— Leslie Lamport

- we attempted a mechanized proof of C code using a global invariant
  - ACSL, Frama-C+WP: verify invariant preserved by each atomic step of a process
- variables represented as arrays of length `nprocs`
- extra array for control location of each process
- message buffers represented as arrays
- invariant relates all of above

Frama-C + WP Verification of Cyclic Exchange: Excerpt

```
/*@ axiomatic OracleSpec {
  @   logic double oracle(int t, int i);
  @   axiom oracle_ax: \forall int t,i; t > 0 ==>
  @           oracle(t-1, LEFT(i)) == oracle(t, i);
  @ }
 */

//@ . . .
//@ predicate inv1 = \forall int i; 0 <= i < nprocs ==>
                size[i] == 1 ==> chan[i] == oracle(sc[i]-1, i)

//@ . . .
//@ predicate inv2 = \forall int i; 0 <= i < nprocs ==>
                      rc[i] == sc[LEFT(i)] - size[LEFT(i)];
//@ . . .

#define inv (. . . inv1 && inv2 && . . .)
```

Frama-C + WP Verification of Cyclic Exchange: Summary

- 54 lines of ACSL annotations for 17 lines of C code
- all verification conditions but 1 discharged with Why3, Alt-Ergo, CVC4
- one VC implying deadlock-freedom could not be proved by any automated prover
  - we could prove it using CVC4 for `nprocs` ≤ 200
  - we could prove it by hand
  - Sebastiaan Joosten proved it using Isabelle ($\sim$150 lines)

```
#define NPROCSB 20
#define LEFT(rank) ((rank) > 0 ? (rank) - 1 : nprocs - 1)
//@ ghost int sc[NPROCSB], rc[NPROCSB], sizes[NPROCSB];
int nprocs, nsteps;
/* note that inv0 and inv5 is not needed for proving the deadlock-freedom condition */
/*@ axiomatic FDL {
  @
  @ predicate inv1 = \forall integer i; 0 <= i < nprocs ==> 0 <= sizes[i] <= 1;
  @ predicate inv2 = \forall integer i; 0 <= i < nprocs ==> 0 <= sc[i] <= nsteps;
  @ predicate inv3 = \forall integer i; 0 <= i < nprocs ==> 0 <= rc[i] <= nsteps;
  @ predicate inv4 = \forall integer i; 0 <= i < nprocs ==> rc[i]==sc[LEFT(i)]-sizes[LEFT(i)];
  @ predicate inv6 = \forall integer i; 0 <= i < nprocs ==> (sc[i]-rc[i]==0) || (sc[i]-rc[i]==1);
  @ predicate fdl = \exists integer i; 0 <= i < nprocs && ((sc[i]-rc[i]==0 && sizes[i]==0) || (sc[i]-rc[i]==1 && sizes[LEFT(i)]==1));
  @
  @  lemma bounded_free_of_deadlock : inv1 && inv2 && inv3 && inv4 && inv6 && 0<nprocs<=NPROCSB
  @                                   ==> fdl;
  @ } */
```

Related Work

ParTypes

- *A Type Discipline for Message Passing Parallel Programs*
  - Vasconcelos, Martins, López, Yoshida
  - ACM ToPLaS 2022
- based on **session types**
- user specifies a communication protocol in a simple language
  - this defines a type
- an algorithm checks that each process conforms to the protocol (has the specified type)
- works for deterministic programs like `diffusion2d_mpi.c`
- verifies deadlock-freedom, termination for any number of processes
- does not say anything about the computation

Nondeterminism

- all examples so far are deterministic
  - for a given input...
  - any two executions are equivalent
    - one can be obtained from the other by repeatedly transposing adjacent commuting transitions
  - for any process p: sequence of process states of p is the same for any execution
  - executions only differ by how actions from processes are interleaved
  - this is known *a priori* because of the subset of MPI used
    - in particular: each receive statement specifies its source

Nondeterminism

- all examples so far are deterministic
  - for a given input...
  - any two executions are equivalent
    - one can be obtained from the other by repeatedly transposing adjacent commuting transitions
  - for any process p: sequence of process states of p is the same for any execution
  - executions only differ by how actions from processes are interleaved
  - this is known *a priori* because of the subset of MPI used
    - in particular: each receive statement specifies its source
- many algorithms in scientific computing can be expressed deterministically

Nondeterminism

- all examples so far are deterministic
  - for a given input...
  - any two executions are equivalent
    - one can be obtained from the other by repeatedly transposing adjacent commuting transitions
  - for any process p: sequence of process states of p is the same for any execution
  - executions only differ by how actions from processes are interleaved
  - this is known *a priori* because of the subset of MPI used
    - in particular: each receive statement specifies its source
- many algorithms in scientific computing can be expressed deterministically
- but some algorithms require a process to receive from any source
- MPI provides a way to do this
  - the `source` argument to `MPI_Recv` may be `MPI_ANY_SOURCE`
  - a "wildcard" receive

A classic example of a nondeterministic algorithm: **Manager-Worker**

- break up problem into finite set of tasks — with many more tasks than processes

A classic example of a nondeterministic algorithm: **Manager-Worker**

- break up problem into finite set of tasks — with many more tasks than processes
- one process plays role of manager; remaining processes are workers

A classic example of a nondeterministic algorithm: **Manager-Worker**

- break up problem into finite set of tasks — with many more tasks than processes
- one process plays role of manager; remaining processes are workers
- manager
  1. distributes one task to each worker
  2. waits for any worker to send back result
  3. processes result and sends new task to that worker
  4. if no tasks remain, sends termination signal to worker instead
  5. when all results have been returned and termination signals sent, finished

A classic example of a nondeterministic algorithm: **Manager-Worker**

- break up problem into finite set of tasks — with many more tasks than processes
- one process plays role of manager; remaining processes are workers
- manager
  1. distributes one task to each worker
  2. waits for any worker to send back result
  3. processes result and sends new task to that worker
  4. if no tasks remain, sends termination signal to worker instead
  5. when all results have been returned and termination signals sent, finished
- worker
  1. waits for task from manager
  2. solves the task and sends result to manager
  3. repeats until termination signal received

A classic example of a nondeterministic algorithm: **Manager-Worker**

- break up problem into finite set of tasks — with many more tasks than processes
- one process plays role of manager; remaining processes are workers
- manager
  1. distributes one task to each worker
  2. waits for any worker to send back result
  3. processes result and sends new task to that worker
  4. if no tasks remain, sends termination signal to worker instead
  5. when all results have been returned and termination signals sent, finished
- worker
  1. waits for task from manager
  2. solves the task and sends result to manager
  3. repeats until termination signal received
- Example (based on example from **Using MPI**): matrix multiplication: compute AB
  - manager has A; every worker gets a complete copy of B
  - a task: compute one row of AB

Manager-Worker pseudocode

```
count := 0;
while count < P − 1 do
    send ⟨count + 1, A[count]⟩ to count + 1;
    count := count + 1;
end while
i := 0;
while i < N do
    ⟨tag, T⟩ := recv(any(source));
    C[tag − 1] := T;
    if count < N then
        send ⟨count + 1, A[count]⟩ to source;
        count := count + 1;
    end if
    i := i + 1;
end while
```

Manager

```
while true do
    ⟨tag, in⟩ := recv(0);
    out := in ∗ B;
    send ⟨tag, out⟩ to 0;
end while
```

Worker

matmat_mpi.c: matrix multiplication, manager-worker, excerpt

```
void manager() { ...
  // Broadcast entire matrix B to all workers...
  MPI_Bcast(&b[0][0], L*M, MPI_DOUBLE, 0, comm);
  // Send one task to each worker, unless you run out of tasks...
  for (count = 0; count < nprocs-1 && count < N; count++)
    MPI_Send(&a[count][0], L, MPI_DOUBLE, count+1, count+1, comm);
  // Receive result, insert into C, send the next task, repeat...
  for (int i = 0; i < N; i++) {
    MPI_Recv(tmp, M, MPI_DOUBLE, MPI_ANY_SOURCE, MPI_ANY_TAG, comm, &status);
    for (int j = 0; j < M; j++)
      c[status.MPI_TAG-1][j] = tmp[j];
    worker_counts[status.MPI_SOURCE]++;
    if (count < N) {
      MPI_Send(&a[count][0], L, MPI_DOUBLE, status.MPI_SOURCE, count+1, comm);
      count++;
    }
  }
  // send termination signals (tag=0) to all workers...
}
```

Challenge 2

- Construct a mechanized proof that for any
  - $L, M, N \geq 1$,
  - $N \times L$ matrix A and $L \times M$ matrix B,
  - `nprocs` ≥ 2
- `matmat_mpi.c` terminates with $C = AB$ on process 0.

Challenge 2

- Construct a mechanized proof that for any
  - $L, M, N \geq 1$,
  - $N \times L$ matrix A and $L \times M$ matrix B,
  - `nprocs` ≥ 2
- `matmat_mpi.c` terminates with $C = AB$ on process 0.

A first step...

- a "LaTeX proof"
- construction of a global invariant
- invariant under each atomic code block

Variables used to model state of `matmat_mpi.c`

W = set of process ranks of workers

1. u_j ($j \in W$) : buffered messages from manager to worker j
2. v_j ($j \in W$) : buffered messages from worker j to manager
3. all variables in manager
   - `count`, `i`, ...
4. auxiliary data for manager
   - Solved (set of int): set of tasks solved, initially empty
   - Out (set of int): set of tasks sent out but solution not yet received
5. for each j, all variables in worker j
   - `tag`, `in`, ...
6. auxiliary data for worker j
   - InWorker[j] (set of int): set of tasks that worker j is currently working on
     - cardinality at most 1

Auxiliary definitions used to express invariant of `matmat_mpi.c`

$$\mathsf{ToWorker}[j] = \bigcup_{\langle t,V\rangle \in u[j]} \{t-1\}$$

$$\mathsf{ToManager}[j] = \bigcup_{\langle t,T\rangle \in v[j]} \{t-1\}$$

$$\mathsf{InWorker} = \bigcup_{j \in W} \mathsf{InWorker}[j]$$

$$\mathsf{ToWorker} = \bigcup_{j \in W} \mathsf{ToWorker}[j]$$

$$\mathsf{ToManager} = \bigcup_{j \in W} \mathsf{ToManager}[j]$$

The "core invariant" of `matmat`

Let disjoint be the assertion that says the $3(P-1)$ sets InWorker$[*]$, ToWorker$[*]$, ToManager$[*]$ are pairwise disjoint.

Let Φ denote the assertion

$$
\begin{aligned}
& \mathsf{Out} \cap \mathsf{Solved} = \emptyset \\
\wedge\ & \mathsf{Out} \cup \mathsf{Solved} = \{0, \ldots, \mathit{count} - 1\} \\
\wedge\ & \mathsf{Out} = \mathsf{InWorker} \cup \mathsf{ToWorker} \cup \mathsf{ToManager} \\
\wedge\ & \mathsf{disjoint} \\
\wedge\ & (\forall j \in \mathsf{Solved}.\ C[j] = A[j] * B) \\
\wedge\ & (\forall j \in W.\ \forall \langle t, V \rangle \in u[j].\ V = A[t-1]) \\
\wedge\ & (\forall j \in W.\ \forall \langle t, T \rangle \in v[j].\ T = A[t-1] * B).
\end{aligned}
$$

Claim: Φ is invariant under each atomic block of `matmat`.

$\{\Phi\}$
$count := 0;\ \mathsf{Out} := \emptyset;\ \mathsf{Solved} = \emptyset;$
$\{0 \leq count \leq P-1 \wedge \mathsf{Solved} = \emptyset \wedge \Phi\}$
while $count < P-1$ **do**
  $\{0 \leq count < P-1 \wedge \mathsf{Solved} = \emptyset \wedge \Phi\}$
  send $\langle count+1, A[count]\rangle$ **to** $count+1;\ \mathsf{Out} := \mathsf{Out} \cup \{count\};\ count := count+1;$
  $\{0 \leq count \leq P-1 \wedge \mathsf{Solved} = \emptyset \wedge \Phi\}$
end while
$\{count = P-1 \wedge \mathsf{Solved} = \emptyset \wedge \Phi\}$
$i := 0;$
$\{0 \leq i \leq N \wedge |\mathsf{Solved}| = i \wedge count = \min\{i+P-1, N\} \wedge \Phi\}$
while $i < N$ **do**
  $\{0 \leq i < N \wedge |\mathsf{Solved}| = i \wedge count = \min\{i+P-1, N\} \wedge \Phi\}$
  $\langle tag, T\rangle :=$ **recv**(**any**(*source*));
  $C[tag-1] := T;\ \mathsf{Solved} := \mathsf{Solved} \cup \{tag-1\};\ \mathsf{Out} := \mathsf{Out} \setminus \{tag-1\};$
  $\{0 \leq i < N \wedge |\mathsf{Solved}| = i+1 \wedge count = \min\{i+P-1, N\} \wedge \Phi\}$
  if $count < N$ **then**
    $\{count < N \wedge 0 \leq i < N \wedge |\mathsf{Solved}| = i+1 \wedge count = i+P-1 \wedge \Phi\}$
    send $\langle count+1, A[count]\rangle$ **to** *source*; $\mathsf{Out} := \mathsf{Out} \cup \{count\};\ count := count+1;$
  end if
  $\{0 \leq i < N \wedge |\mathsf{Solved}| = i+1 \wedge count = \min\{i+P, N\} \wedge \Phi\}$
  $i := i+1;$
  $\{0 \leq i \leq N \wedge |\mathsf{Solved}| = i \wedge count = \min\{i+P-1, N\} \wedge \Phi\}$
end while
$\{|\mathsf{Solved}| = N \wedge \Phi\}$

Verification of Manager-Worker

- I believe this proof is correct
- how can it be formally attached to the source code and mechanized?
- how can the annotation burden and effort be minimized?
- what tools can help?