Deductive Verification of Parallel Programs Using Why3

César Santos, Vasco Vasconcelos, Francisco Martins

LaSIGE, Faculty of Sciences of the University of Lisbon

July 5, 2015

Outline

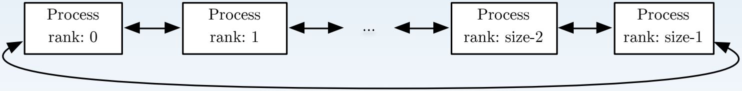
1. Introduction
2. Approach
3. Protocol language
4. Programming language
5. Results and conclusions

MPI

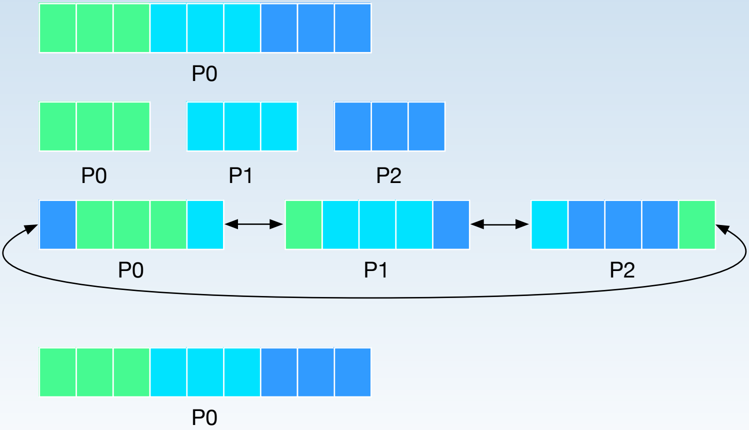
- Message-based specification for parallel computing
- Industry standard (C/Fortran libraries)
- Single-Program Multiple-Data (SPMD)
- Every process is issued a rank (process number)
- Point-to-point and collective communication
- Used for simulations that require a lot of computational power

Example: Finite differences

- Numeric method for solving differential equations
- The program starts with an initial solution X_0, and calculates X_1,X_2,X_3,... iteratively until a maximum number of iterations are executed
- Processes are organized in a ring topology



Example: Finite differences



Example: Finite differences

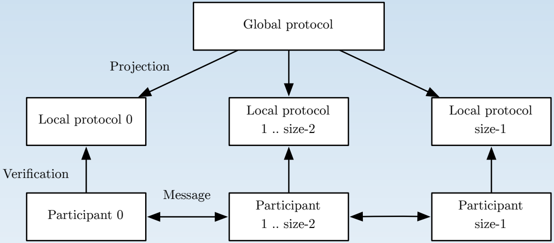
```c
int main(int argc, char** argv) {
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);
    MPI_Scatter(data, n/size, MPI_FLOAT, &local[1], n/size, MPI_FLOAT, 0, MPI_COMM_WORLD);
    int left = rank == 0 ? size - 1 : rank - 1;
    int right = rank == size - 1 ? 0 : rank + 1;
    for (iter = 1; i <= ITERATIONS; iter++) {
        MPI_Send(&local[1], 1, MPI_FLOAT, left, 0, MPI_COMM_WORLD);
        MPI_Send(&local[n/size], 1, MPI_FLOAT, right, 0, MPI_COMM_WORLD);
        MPI_Recv(&local[n/size+1], 1, MPI_FLOAT, right, 0, MPI_COMM_WORLD, &status);
        MPI_Recv(&local[0], 1, MPI_FLOAT, left, 0, MPI_COMM_WORLD, &status);
        // Computation is performed here, removed for simplicity
    }
    MPI_Reduce(&localErr, &globalErr, 1, MPI_FLOAT, MPI_MAX, 0, MPI_COMM_WORLD);
    MPI_Gather(&local[1], n/size, MPI_FLOAT, data, n/size, MPI_FLOAT, 0, MPI_COMM_WORLD);
    MPI_Finalize();
    return 0;
}
```

Does it deadlock? Is it communication type safe?

Challenges

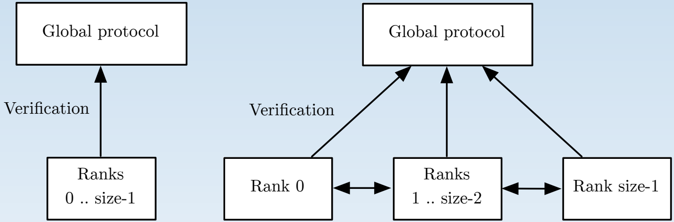
- Verifying parallel programs is difficult
  - State explosion problem
- Many verification tools only work at runtime
  - Dependent on the quality of the test set
  - Performance costs

Multi-party session types



- Theory for communication protocols
- Global protocol is projected for each process
- Properties guaranteed for program: absence of deadlocks and of communication errors
- Does not suffer from the state explosion problem

ParTypes

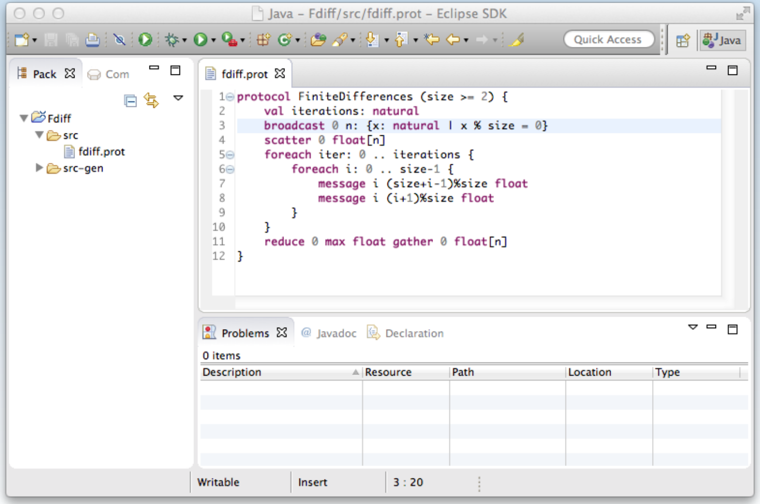


- Inspired by multi-party session types
- Point-to-point and collective communication, for loops and collective choices
- No separate projection step
- Can be used with both SPMD (like MPI) and MPMD programs

Protocol language

```
protocol FiniteDifferences (size >= 2) {
    val iterations: natural
    broadcast 0 n: {x: natural | x % size = 0}
    scatter 0 float[n]
    foreach iter: 0 .. iterations {
        foreach i: 0 .. size-1 {
            message i (size+i-1)%size float
            message i (i+1)%size float
        }
    }
    reduce 0 max float
    gather 0 float[n]
}
```

Protocol compiler (Eclipse plugin)



Foreach expansion

i	Loop body	Rank 0	Rank 1	...	Rank size-2	Rank size-1
0	message 0 size-1 message 0 1	send size-1 send 1	recv 0			recv 0
1	message 1 0 message 1 2	recv 1	send 0 send 2			
2	message 2 1 message 2 3		recv 2			
...						
size-2	message size-2 size-3 message size-2 size-1				send size-3 send size-1	recv size-1
size-1	message size-1 size-2 message size-1 0	recv size-1			recv size-1	send size-2 send 0

Example: Finite differences

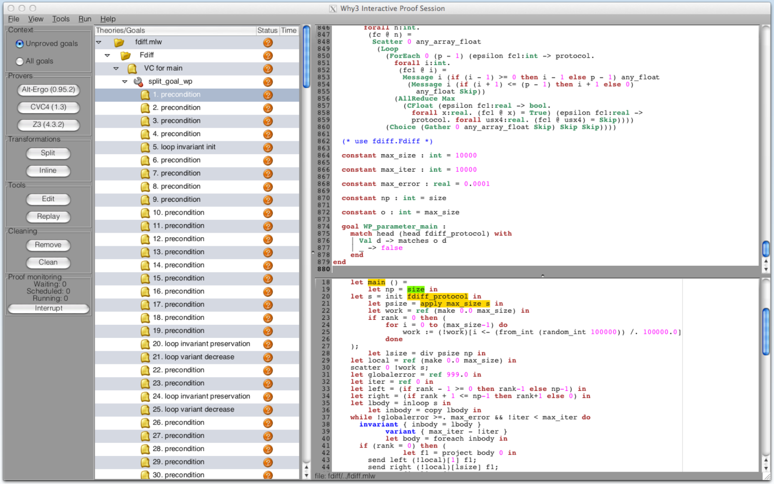
```
int main(int argc, char** argv) {
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);
    MPI_Scatter(data, n/size, MPI_FLOAT, &local[1], n/size, MPI_FLOAT, 0, MPI_COMM_WORLD);
    int left = rank == 0 ? size - 1 : rank - 1;
    int right = rank == size - 1 ? 0 : rank + 1;
    for (iter = 1; i <= ITERATIONS; iter++) {
        MPI_Send(&local[1], 1, MPI_FLOAT, left, 0, MPI_COMM_WORLD);
        MPI_Send(&local[n/size], 1, MPI_FLOAT, right, 0, MPI_COMM_WORLD);
        MPI_Recv(&local[n/size+1], 1, MPI_FLOAT, right, 0, MPI_COMM_WORLD, &status);
        MPI_Recv(&local[0], 1, MPI_FLOAT, left, 0, MPI_COMM_WORLD, &status);
        // Computation is performed here, removed for simplicity
    }
    MPI_Reduce(&localErr, &globalErr, 1, MPI_FLOAT, MPI_MAX, 0, MPI_COMM_WORLD);
    MPI_Gather(&local[1], n/size, MPI_FLOAT, data, n/size, MPI_FLOAT, 0, MPI_COMM_WORLD);
    MPI_Finalize();
    return 0;
}
```

This is going to deadlock

Why3

- Deductive software verification platform
- Specification language (Why) + programming language (WhyML)
- Verification conditions can be split in parts and proven with different SMT solvers (or with proof assistants such as Coq)
- Chosen to avoid the annotation overhead required for static verification of C/Fortran programs
- Experiment in programming methodology

Why3 IDE


Why3 Interactive Proof Session
File View Tools Run Help
Context
Unproved goals
All goals
Provers
Alt-Ergo (0.95.2)
CVC4 (1.3)
Z3 (4.3.2)
Transformations
Split
Inline
Tools
Edit
Replay
Cleaning
Remove
Clean
Proof monitoring
Waiting: 0
Scheduled: 0
Running: 0
Interrupt
Theories/Goals
Status
Time
fdiff.mlw
Fdiff
VC for main
split_goal_wp
1. precondition
5. loop invariant init
20. loop invariant preservation
21. loop variant decrease
constant max_size : int = 10000
constant max_iter : int = 10000
constant max_error : real = 0.0001
goal WP_parameter_main :
file: fdiff/../fdiff.mlw

Why3 theory for protocols

```
type protocol =
      Val datatype
    | Broadcast int datatype continuation
    | Scatter int datatype protocol
    | Gather int datatype protocol
    | Message int int datatype protocol
    | Reduce int op datatype protocol
    | Skip
    | AllGather datatype continuation
    | Foreach int int (cont int) protocol
    | AllReduce op datatype continuation
with
    op = Max | Min | Sum | ...
```

WhyML library

- Inspired by MPI
- Point-to-point and collective communication
- Primitives annotated with pre and post-conditions
- Verification guided by the protocol
- Requires program annotations for loops and choices

Finite differences in WhyML

```
let main () =
    let s = init fdiff_protocol in
    let iterations = apply 100000 s in
    let n = broadcast 0 input s in
    let local = scatter 0 work s in
    let left = (if rank > 0 then rank-1 else size-1) in
    let right = (if rank < size-1 then rank+1 else 0) in
    let inbody = expand (foreach s) rank in (* Annotation *)
    for iter = 1 to iterations do
        (* Loop body *)
    done;
    globalerror := reduce 0 Max !localerror s;
    gather 0 local s;
    isSkip s; (* Annotation *)
```

Fixed communication

```
let body = foreach inbody in (* Annotation *)
if (rank = 0) then (
   let f1 = expand body 0 in (* Annotation *)
   send left local[1] f1;
   send right local[n/size] f1; isSkip f1; (* Annotation *)
   let f2 = expand body 1 in (* Annotation *)
   local[n/size+1] <- recv right f2; isSkip f2; (* Annotation *)
   let f3 = expand body (np-1) in (* Annotation *)
   local[0] <- recv left f3; isSkip f3); (* Annotation *)
else if (rank = size-1) then (
   let f1 = expand body 0 in (* Annotation *)
   local[lsize+1] <- recv right f1; isSkip f1; (* Annotation *)
   let f2 = expand body (np-2) in (* Annotation *)
   local[0] <- recv left f2; isSkip f2; (* Annotation *)
   let f3 = expand body (np-1) in (* Annotation *)
   send left local[1] f3;
   send right local[lsize] f3; isSkip f3); (* Annotation *)
else (
   ...
isSkip inbody; (* Annotation *)
(* Computation is performed here, removed for simplicity *)
```

Fixed communication

```
let body = foreach inbody in (* Annotation *)
if (rank = 0) then (
    let f = expand body [size-1, 0, 1] in (* Annotation *)
    send left local[1] f;
    send right local[n/size] f;
    local[n/size+1] <- recv right f;
    local[0] <- recv left f;
    isSkip f); (* Annotation *)
else if (rank = size-1) then (
    let f = expand body [size-2, size-1, 0] in (* Annotation *)
    local[lsize+1] <- recv right f;
    local[0] <- recv left f;
    send left local[1] f;
    send right local[lsize] f;
    isSkip f); (* Annotation *)
else (
    ...
isSkip inbody; (* Annotation *)
(* Computation is performed here, removed for simplicity *)
```

Results: Annotation effort and verification time

Program	Why3 LOC	Why3 Anot	Ratio	VCC LOC	VCC Anot	Why3/VCC
Pi	33	6	18%	42	10	23%
Finite differences	86	29	33%	128	49	38%
Parallel dot	61	11	18%	99	30	30%

Program	Why3 Sub-Proofs	Why3 Time (s)	VCC Time (s)	Why3/VCC
Pi	27	1,6	2,4	66,7%
Finite differences	374	14,9	16,1	92,5%
Parallel dot	298	7.9	7,4	106,7%

Conclusions

- Our approach does not suffer from the state-explosion problem, typical of model checking
- Nor does it require any sort of runtime verification
- Results comparable with the closest tool, with less annotations and more verification options
- Many program annotations still required
- ... but can be simplified
- Protocols have the advantage of also serving as documentation

Future work

- Larger subset of MPI primitives (e.g. non-blocking, wildcard receive)
- Reduce annotation effort
- Adapt work for industry use
- Cover real-world applications

Thank you!

Questions?