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

```plaintext
1 protocol FiniteDifferences (size >= 2) {
2     val iterations : natural
3     broadcast 0 n: {x: natural | x % size = 0}
4     scatter 0 float [n]
5     foreach iter: 0 .. iterations {
6         foreach i: 0 .. size-1 {
7             message i (size+i-1)%size float
8             message i (i+1)%size float
9         }
10     }
11     reduce 0 max float
12     gather 0 float [n]
13 }
```
Protocol compiler (Eclipse plugin)
## Foreach expansion

<table>
<thead>
<tr>
<th>i</th>
<th>Loop body</th>
<th>Rank 0</th>
<th>Rank 1</th>
<th>...</th>
<th>Rank size-2</th>
<th>Rank size-1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>message 0 size-1</td>
<td>send size-1</td>
<td>recv 0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>message 0 1</td>
<td>send 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>message 1 0</td>
<td>recv 1</td>
<td>send 0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>message 1 2</td>
<td></td>
<td>send 2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>message 2 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>message 2 3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>size-2</td>
<td>message size-2 size-3</td>
<td>send size-3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>message size-2 size-1</td>
<td>send size-1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>size-1</td>
<td>message size-1 size-2</td>
<td>send size-2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>message size-1 0</td>
<td>recv size-1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>message size-1</td>
<td></td>
<td>.recv size-1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>send 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
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;
}
```

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

Deductive Verification of Parallel Programs Using Why3

César Santos
Why3 theory for protocols

```plaintext
1  type protocol =
2       Val datatype
3   | Broadcast int datatype continuation
4   | Scatter int datatype protocol
5   | Gather int datatype protocol
6   | Message int int datatype protocol
7   | Reduce int op datatype protocol
8   | Skip
9   | AllGather datatype continuation
10  | Foreach int int (cont int) protocol
11  | AllReduce op datatype continuation
12  with
13    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

```ocaml
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

1. `let body = foreach inbody in (* Annotation *)
2. if (rank = 0) then (
3.   let f1 = expand body 0 in (* Annotation *)
4.   send left local[1] f1;
5.   send right local[n/size] f1; isSkip f1; (* Annotation *)
6.   let f2 = expand body 1 in (* Annotation *)
7.   local[n/size+1] <- recv right f2; isSkip f2; (* Annotation *)
8.   let f3 = expand body (np-1) in (* Annotation *)
9.   local[0] <- recv left f3; isSkip f3); (* Annotation *)
10. else if (rank = size-1) then (
11.   let f1 = expand body 0 in (* Annotation *)
12.   local[lsize+1] <- recv right f1; isSkip f1; (* Annotation *)
13.   let f2 = expand body (np-2) in (* Annotation *)
14.   local[0] <- recv left f2; isSkip f2; (* Annotation *)
15.   let f3 = expand body (np-1) in (* Annotation *)
16.   send left local[1] f3;
17.   send right local[lsize] f3; isSkip f3); (* Annotation *)
18. else (  isSkip inbody; (* Annotation *)
19. (* Computation is performed here, removed for simplicity *)

Fixed communication

1 let body = foreach inbody in (* Annotation *)
2 if (rank = 0) then (  
3     let f = expand body [size-1, 0, 1] in (* Annotation *)
4     send left local[1] f;
5     send right local[n/size] f;
6     local[n/size+1] <- recv right f;
7     local[0] <- recv left f;
8     isSkip f); (* Annotation *)
9 else if (rank = size-1) then (  
10    let f = expand body [size-2, size-1, 0] in (* Annotation *)
11    local[lsize+1] <- recv right f;
12    local[0] <- recv left f;
13    send left local[1] f;
14    send right local[lsize] f;
15    isSkip f); (* Annotation *)
16 else (  
17    ...
18    isSkip inbody; (* Annotation *)
19    (* Computation is performed here, removed for simplicity *)

### Results: Annotation effort and verification time

<table>
<thead>
<tr>
<th>Program</th>
<th>Why3 LOC</th>
<th>Why3 Anot</th>
<th>Ratio</th>
<th>VCC LOC</th>
<th>VCC Anot</th>
<th>Why3/VCC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pi</td>
<td>33</td>
<td>6</td>
<td>18%</td>
<td>42</td>
<td>10</td>
<td>23%</td>
</tr>
<tr>
<td>Finite differences</td>
<td>86</td>
<td>29</td>
<td>33%</td>
<td>128</td>
<td>49</td>
<td>38%</td>
</tr>
<tr>
<td>Parallel dot</td>
<td>61</td>
<td>11</td>
<td>18%</td>
<td>99</td>
<td>30</td>
<td>30%</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Program</th>
<th>Why3 Sub-Proofs</th>
<th>Why3 Time (s)</th>
<th>VCC Time (s)</th>
<th>Why3/VCC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pi</td>
<td>27</td>
<td>1,6</td>
<td>2,4</td>
<td>66,7%</td>
</tr>
<tr>
<td>Finite differences</td>
<td>374</td>
<td>14,9</td>
<td>16,1</td>
<td>92,5%</td>
</tr>
<tr>
<td>Parallel dot</td>
<td>298</td>
<td>7,9</td>
<td>7,4</td>
<td>106,7%</td>
</tr>
</tbody>
</table>
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?