Protocol-based Verification of Message-passing Parallel Programs

Hugo A. López, Eduardo R. B. Marques, Francisco Martins, César Santos, **Vasco T. Vasconcelos**
Nicholas Ng, Nobuko Yoshida

University of Lisbon
Imperial College London
Technical University of Denmark

Advances in Programming Languages and Systems
Frankfurt, December 16, 2015

Motivation

High-performance parallel computing

- Getting parallel programs right is not easy
- Testing is expensive
- We can speedup development, producing safer and cheaper solutions

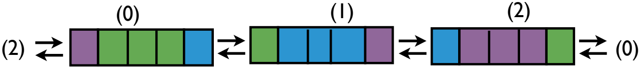
Finite differences algorithm

Input data at rank 0

Scatter data for each participant

Each participant computes its finite differences

Send/recv boundary values to determine convergence



Perform global reduction (AllReduce) to compute the max error;
Loop if convergence criterion not met, up until to MAX_ITER

Gather the solution data at rank 0, if there was convergence.

Is this program communication safe? Deadlock free?

```
MPI_Init(&argc,&argv);
...
for (iter = 1; i ≤ NUM_ITER; iter++) {
 if (rank == 0) {
  MPI_Send(&local[1],1,MPI_FLOAT,left,...);
  MPI_Send(&local[n/size],1,MPI_FLOAT,right,...);
  MPI_Recv(&local[n/size+1],1,MPI_FLOAT,right,...);
  MPI_Recv(&local[0],1,MPI_FLOAT,left,...);
 } else if (rank == size - 1) {
  MPI_Recv(&local[n/size+1],2,MPI_FLOAT,right,...);
  MPI_Send(&local[1],1,MPI_FLOAT,left,...);
  MPI_Recv(&local[0],1,MPI_FLOAT,left,...);
  MPI_Send(&local[n/size],1,MPI_FLOAT,right,...);
 } else {
  MPI_Recv(&local[0],1,MPI_INT,left,...);
  MPI_Send(&local[1],1,MPI_FLOAT,left,...);
  MPI_Send(&local[n/size],1,MPI_FLOAT,right,...);
  MPI_Recv(&local[n/size+1],1,MPI_FLOAT,right,...);
 }
}
...
MPI_Finalize();
```

Array length mismatch

```
MPI_Init(&argc,&argv);
...
for (iter = 1; i ≤ NUM_ITER; iter++) {
 if (rank == 0) {
  MPI_Send(&local[1],1,MPI_FLOAT,left,...);
  MPI_Send(&local[n/size],1,MPI_FLOAT,right,...);
  MPI_Recv(&local[n/size+1],1,MPI_FLOAT,right,...);
  MPI_Recv(&local[0],1,MPI_FLOAT,left,...);
 } else if (rank == size - 1) {
  MPI_Recv(&local[n/size+1],2,MPI_FLOAT,right,...);
  MPI_Send(&local[1],1,MPI_FLOAT,left,...);
  MPI_Recv(&local[0],1,MPI_FLOAT,left,...);
  MPI_Send(&local[n/size],1,MPI_FLOAT,right,...);
 } else {
  MPI_Recv(&local[0],1,MPI_INT,left,...);
  MPI_Send(&local[1],1,MPI_FLOAT,left,...);
  MPI_Send(&local[n/size],1,MPI_FLOAT,right,...);
  MPI_Recv(&local[n/size+1],1,MPI_FLOAT,right,...);
 }
}
...
MPI_Finalize();
```

Type mismatch

```
MPI_Init(&argc,&argv);
...
for (iter = 1; i ≤ NUM_ITER; iter++) {
 if (rank == 0) {
  MPI_Send(&local[1],1,MPI_FLOAT,left,...);
  MPI_Send(&local[n/size],1,MPI_FLOAT,right,...);
  MPI_Recv(&local[n/size+1],1,MPI_FLOAT,right,...);
  MPI_Recv(&local[0],1,MPI_FLOAT,left,...);
 } else if (rank == size - 1) {
  MPI_Recv(&local[n/size+1],2,MPI_FLOAT,right,...);
  MPI_Send(&local[1],1,MPI_FLOAT,left,...);
  MPI_Recv(&local[0],1,MPI_FLOAT,left,...);
  MPI_Send(&local[n/size],1,MPI_FLOAT,right,...);
 } else {
  MPI_Recv(&local[0],1,MPI_INT,left,...);
  MPI_Send(&local[1],1,MPI_FLOAT,left,...);
  MPI_Send(&local[n/size],1,MPI_FLOAT,right,...);
  MPI_Recv(&local[n/size+1],1,MPI_FLOAT,right,...);
 }
}
...
MPI_Finalize();
```

Deadlock: wrong send/receive order

```
MPI_Init(&argc,&argv);
...
for (iter = 1; i ≤ NUM_ITER; iter++) {
 if (rank == 0) {
  MPI_Send(&local[1],1,MPI_FLOAT,left,...);
  MPI_Send(&local[n/size],1,MPI_FLOAT,right,...);
  MPI_Recv(&local[n/size+1],1,MPI_FLOAT,right,...);
  MPI_Recv(&local[0],1,MPI_FLOAT,left,...);
 } else if (rank == size - 1) {
  MPI_Recv(&local[n/size+1],2,MPI_FLOAT,right,...);
  MPI_Send(&local[1],1,MPI_FLOAT,left,...);
  MPI_Recv(&local[0],1,MPI_FLOAT,left,...);
  MPI_Send(&local[n/size],1,MPI_FLOAT,right,...);
 } else {
  MPI_Recv(&local[0],1,MPI_INT,left,...);
  MPI_Send(&local[1],1,MPI_FLOAT,left,...);
  MPI_Send(&local[n/size],1,MPI_FLOAT,right,...);
  MPI_Recv(&local[n/size+1],1,MPI_FLOAT,right,...);
 }
}
...
MPI_Finalize();
```

Deadlock: a send operation is missing

```
MPI_Init(&argc,&argv);
...
for (iter = 1; i ≤ NUM_ITER; iter++) {
 if (rank == 0) {
  MPI_Send(&local[1],1,MPI_FLOAT,left,...);
  MPI_Send(&local[n/size],1,MPI_FLOAT,right,...);
  MPI_Recv(&local[n/size+1],1,MPI_FLOAT,right,...);
  MPI_Recv(&local[0],1,MPI_FLOAT,left,...);
 } else if (rank == size - 1) {
  MPI_Recv(&local[n/size+1],2,MPI_FLOAT,right,...);
  MPI_Send(&local[1],1,MPI_FLOAT,left,...);
  MPI_Recv(&local[0],1,MPI_FLOAT,left,...);
  MPI_Send(&local[n/size],1,MPI_FLOAT,right,...);
 } else {
  MPI_Recv(&local[0],1,MPI_INT,left,...);
  MPI_Send(&local[1],1,MPI_FLOAT,left,...);
  MPI_Send(&local[n/size],1,MPI_FLOAT,right,...);
  MPI_Recv(&local[n/size+1],1,MPI_FLOAT,right,...);
 }
}
...
MPI_Finalize();
```

State-of-the-art tools

- Model checking and symbolic execution (e.g., TASS, CIVL)
  - Search space grows exponentially with the number of processes
  - Verification of real-world applications limits the number of processes to less than a dozen
- Runtime verification (e.g., ISP, MUST)
  - Cannot guarantee the absence of faults
  - Difficulty in producing meaningful test suites
  - Time to run the test suite
  - Tests need to be run on hardware similar to that of the production environment

Our approach

We attack the problem from a programming language angle:

- propose a protocol (type) language suited for describing the most common scenarios in the practice of parallel programming, and
- statically check that programs conform to a given protocol, thus
- effectively guaranteeing the absence of deadlocks for well-typed programs, regardless of the number of process involved

Behavioural types

- Behavioural types can improve the development of large scale software systems
- Appropriate for describing the interaction part of software systems: modules, objects, message-passing, ...
- Narrate the details, the order, the choices, ..., of the different individual interactions
- Talk about all partners in a computation

Benefits

- Provide for an abstract view of the global interaction of applications' components
- Suitable to discuss the (communication) behaviour of applications
- Are by construction exempt from deadlocks

The types

Types for individual processes

- Values exchanged among processes influence the rest of the protocol $\rightarrow$ dependent types
- Type checking dependent types is undecidable in general
- We use a restricted form of dependent types, where types depend on objects of a separate domain (Xi&Pfenning, POPL'99)

The type for the finite differences algorithm

```
// problem dimension
val n : {x:integer | x ≥ 0 && x % size=0}.
// number of iterations
broadcast 1 numIter: integer.
// distribute input array
scatter 1 {a:float[] | len(a)*size=n};
forall k ≤ numIter.
  forall i ≤ size.
    // message to right neighbour
    message i (i % size+1) float;
    // message to left neighbour
    message i ((i-2+size) % size+1) float;
  reduce 1;
// gather the solution
gather 1 {b: float[] | len(b)*size=n}
```

Type equivalence

- Central to dependent type systems
- Caters for the semantics of primitive recursion:

$$\frac{\Gamma \vdash i < 1 \textbf{ true} \quad (\dots)}{\Gamma \vdash \mathsf{foreach}\, x \leq i.T \equiv \mathsf{skip}}$$

$$\frac{\Gamma \vdash i \geq 1 \textbf{ true} \quad (\dots)}{\Gamma \vdash \mathsf{foreach}\, x \leq i.T \equiv (T\{i/x\}; \mathsf{foreach}\, x \leq i-1.T)}$$

- Projects types into ranks:

$$\frac{\Gamma \vdash i_1, i_2 \neq \mathsf{rank} \textbf{ true} \qquad (\dots)}{\Gamma \vdash \mathsf{message}\; i_1\; i_2\; D \equiv \mathsf{skip}}$$

Type equivalence in action

$$\text{size}\colon \{x\colon \text{int} \mid x = 3\}, \text{rank}\colon \{y\colon \text{int} \mid y = 1\} \vdash$$

$$\text{foreach } j \leq \text{size}.$$

$$\quad \text{msg } j\ (j\%\text{size} + 1); \text{msg } j\ ((j - 2 + \text{size})\%\text{size} + 1)$$

$$\equiv$$

$$\text{msg } 3\ 1; \text{msg } 3\ 2;$$

$$\text{msg } 2\ 3; \text{msg } 2\ 1;$$

$$\text{msg } 1\ 2; \text{msg } 1\ 3;$$

$$\text{skip}$$

$$\equiv$$

$$\text{msg } 3\ 1; \text{skip}; \text{skip}; \text{msg } 2\ 1; \text{msg } 1\ 2; \text{msg } 1\ 3; \text{skip}$$

$$\equiv$$

$$\text{msg } 3\ 1; \text{msg } 2\ 1; \text{msg } 1\ 2; \text{msg } 1\ 3$$

Program types

- Process types describe individual processes
- Program types
  - describe programs (vectors of processes)
  - are vector of types; one type per process
  - all types in the vector are equivalent; the only program type formation rule:

$$\frac{\text{size} = n, \text{rank} = k \vdash T_k \equiv T : \textbf{type} \quad (1 \leq k \leq n)}{T_1, \ldots, T_n : \textbf{ptype}}$$

Examples of **non** program types

$$(\mathsf{msg}\,1\,2), (\mathsf{msg}\,2\,1)$$
$$(\mathsf{scatter}\,1), (\mathsf{reduce}\,1)$$
$$(\mathsf{msg}\,1\,3; \mathsf{scatter}\,1), (\mathsf{msg}\,1\,3; \mathsf{reduce}\,1), (\mathsf{msg}\,1\,3; \mathsf{scatter}\,1)$$
$$(\mathsf{msg}\,3\,1; \mathsf{msg}\,1\,2), (\mathsf{msg}\,1\,2; \mathsf{msg}\,2\,3), (\mathsf{msg}\,2\,3; \mathsf{msg}\,3\,1)$$

All these types are either deadlocked or lead to a deadlock

The programs

Our programming language

- A small imperative while-language equipped with communication primitives
- Processes are store-expression pairs
- Programs are vectors of processes
- Main judgement: P is a program (behaving as program type S)

$$P : S$$

Main results

Theorem (Agreement for reduction; cf. subject reduction)

If $P_1 \to P_2$ *then* $P_1 : S_1$ *and* $P_2 : S_2$

Theorem (Progress for programs)

If $P_1 : S$ *then* P_1 **halted** *or* $P_1 \to P_2$

A program his halted when all its processes are halted.

Deductive verification of C+MPI code

Message Passing Interface

- For programs written in C or Fortran
- The MPI 3.0 specification is 822 pages long written in good old english
- There are hundreds of MPI primitives
- Includes a multitude of concepts: synchronous vs. asynchronous, immediate vs. blocking, direct vs. broadcast vs. one-sided communication, message tags, communicators, reception from any source of messages with any tag, . . .

Method to verify C+MPI source code against a protocol

1. Write a protocol for the program (the protocol serves as further documentation for the program)
2. Convert it to VCC, a deductive software verifier for C
3. Introduce the required annotations in the C+MPI source code, and
4. Use VCC to check conformance to the protocol

If VCC runs successfully, then the program is guaranteed to follow the protocol and to be exempt from deadlocks, regardless of the number of processes, problem dimension, number of iterations, or any other parameters.

Verification Flow

1. The contract for MPI_Init initializes a ghost variable p with the protocol the program must follow
2. Contracts for MPI communication primitives match p against source code, while advancing p
3. Each occurrence of a C control structure that is related to the protocol is checked against p, relying on adequate annotations
4. The contract for MPI_Finalize asserts that p is be equivalent to **skip**

Evaluation

Tools under test

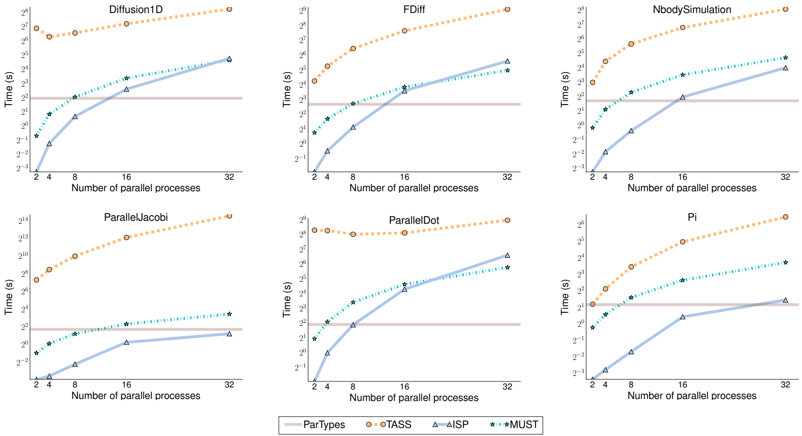
We performed a comparative analysis of ParTypes against state-of-the-art MPI verifiers with similar safety guarantees

- **TASS** A model checker which uses symbolic execution
- **ISP** A dynamic verifier that employs dynamic partial order reduction to select the relevant process schedules
- **MUST** A dynamic verifier that employs a graph-based deadlock detection approach

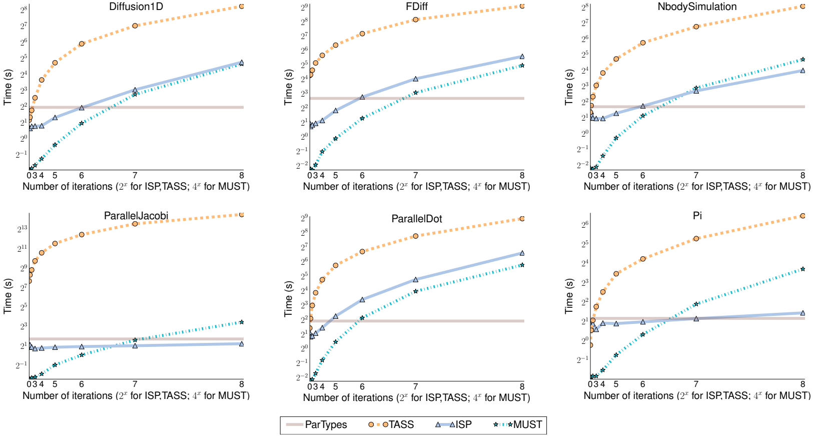
Benchmark suite

- Programs taken from text books (e.g., I. Foster; P. Pacheco) and from the FEVS suite
- The analysis include
  - 1-D heat diffusion simulation
  - finite differences
  - N-body simulation
  - parallel Jacobi equation solver
  - parallel dot product
  - pi approximation

Results for the experiments varying the number of processes (log scale)



Results for the experiments varying the number of loop iterations (log scale)



Conclusion

Conclusion

- We presented a type-based methodology for checking message-passing parallel programs
- By checking that a program follows a given protocol, we guarantee a set of safety properties for the program, in particular that it does not run into deadlocks
- In contrast to other state-of-the-art approaches that suffer from scalability issues, our approach is insensitive to parameters such as the number of processes, problem size, or the number of iterations of a program

Future work

- Address further MPI primitives, e.g., non-blocking operations and wildcard receive (the ability to receive from any source)
- Our VCC methodology is sound but not complete with respect to the core programming language. Try accepting more programs

Google “partypes”

Try it online at http://gloss.di.fc.ul.pt/tryit/ParTypes