Automatic Software Verification of BSPlib-programs: Replicated Synchronization

Arvid Jakobsson

2017-03-20

Supervisors: G. Hains, W. Suijlen, F. Loulergue, F. Dabrowski, W. Bousdira

Context

- **Huawei:** World-leading provider of ICT-solutions
- Huawei has an increasing need for embedded parallel software
- Successful software must be safe and efficient
- Formal method gives mathematical guarantees of safety and efficiency
- **Université d'Orléans** (Laboratoire d'Informatique Fondamental): Strong research focus on formal methods and parallel computing

Overview of AVSBSP

- Goal of the project: a secure, statically verified basis for efficient BSPlib programming
- **Bulk Synchronous Parallel**: simple but powerful model for parallel programming,
- **BSPlib**: a library for BSP-programming in C

Overview of AVSBSP

- Main track: Developing automatic tools for verification of BSPlib based on formal methods.
  - Correct synchronization
  - Correct communication
  - Correct API usage
  - $\Rightarrow$ Automatic verification of safety
- Side-track: Automatic Cost Analysis
  - Automatic BSP cost formula derivation
  - $\Rightarrow$ Automatic verification of performance

Main-track: Verification

- Main track: Developing automatic tools for verification of BSPlib based on formal methods.
  - **Correct synchronization**
  - Correct communication
  - Correct API usage

  $\Rightarrow$ Automatic verification of safety

Motivating example (1)

- Long scientific calculations on cluster in parallel.
- But come Monday: calculation crashed after 10 hours :(
- What went wrong? Let's look at the code!

Motivating example (2)

- *Single Program, Multiple data*: the same program is run in parallel on p processes:

```
// ...
double x = 0.0;
for (int i = 0; i < 100; ++i) {
        x = f(x);

        // ...
}
```

Figure: Parallel SPMD program: Iterative calculation

Motivating example (2)

```
double t0 = bsp_time();
double x = 0.0;
for (int i = 0; i < 100; ++i) {
        x = f(x);

        double t1 = bsp_time();
        if (t1 - t0 > 1.0) {
                print_progress(x);
                t0 = t1;
        }
}
```

Figure: Buggy parallel SPMD program: Harmless printing?

Motivating example (2)

```
void print_progress( double x) {
        int p = bsp_nprocs();
        // Print progress for process 0, 1, 2, ...
        for (int s = 0; s < p; ++p) {
                if (bsp_pid() == s) {
                        printf("progress␣(%d):␣%g\n", s, x);
                }
                bsp_sync();
        }
}
```

Figure: Buggy parallel SPMD program: Harmless printing?

Motivating example (2)

```
double t0 = bsp_time();
double x = 0.0;
for (int i = 0; i < 100; ++i) {
        x = f(x);

        double t1 = bsp_time();
        if (t1 - t0 > 1.0) {
                print_progress(x); // synchronizing
                t0 = t1;
        }
}
```

Figure: Buggy parallel SPMD program: Harmless printing?

Motivating example (2)

```
double t0 = bsp_time();
double x = 0.0;
for (int i = 0; i < 100; ++i) {
        x = f(x);

        double t1 = bsp_time();
        if (t1 - t0 > 1.0) { // Processes agree on this condition?
                print_progress(x); // synchronizing.
                t0 = t1;
        }
}
```

Figure: Buggy parallel SPMD program: Processes agree?

Motivating example (3): Conclusion

- Source of bug: Program hangs since choice to synchronize or not (inside `print_progress(x)`) depends on a value local to each process (`bsp_time()`).
- Possible solution: *To synchronize or not* must only depend on a condition with the same value on all processes.
- Goal: Enforce this solution statically.

Background: Bulk synchronous parallel (1)

- Bulk synchronous parallel (BSP): model of parallel computing
- BSP computation: a sequence of super-steps executed by a fixed number of p processes.
- Each super-step is composed of:
  1. Local computation by each process, followed by
  2. Communication between processes, followed by
  3. A synchronization barrier. Go back to Step 1 or terminate.

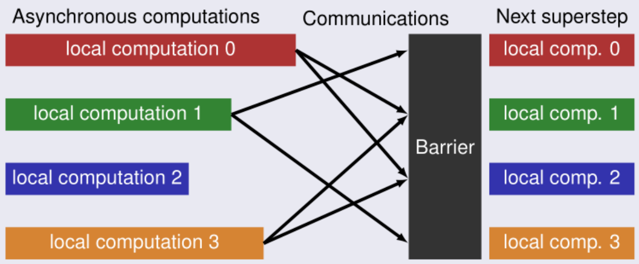

Figure: A BSP superstep

Background: Bulk synchronous parallel (2)

- Invented in the 80's by Leslie Valiant, and several implementations exists, notably: BSPlib, Pregel, MapReduce, most linear algebra packages. . .
- Benefits of BSP compared to other models of parallel computation:
  - Deadlock and data race free
  - Simple but realistic cost model
  - Simplifies algorithm design

Background: BSPlib

- BSPlib: library and interface specification for BSP in C.
- BSPlib follows the *Single Program Multiple Data*-model (SPMD).
- Small set of primitives (20):
  - `bsp_begin`, `bsp_end`, `bsp_pid`, `bsp_nprocs`, `bsp_get`, `bsp_put`, `bsp_sync`, ...
- Several implementations exists: The Oxford BSP Toolset, Paderborn University BSP, MulticoreBSP, Epiphany BSP...

BSPlite

- Toy-language "BSPlite".
- Grammar of BSPlite:

$$\begin{array}{llll} expr & \ni & e & ::= & nprocs \mid pid \mid x \mid n \mid e + e \mid e - e \mid e \times e \\ bexpr & \ni & b & ::= & \texttt{true} \mid \texttt{false} \mid e < e \mid e = e \mid b \textbf{ or } b \mid b \textbf{ and } b \mid !b \\ cmd & \ni & c & ::= & x := e \mid \texttt{skip} \mid \texttt{sync} \mid c ; c \mid \texttt{if}\, b\, \texttt{then}\, c\, \texttt{else}\, c\, \texttt{end} \\ & & & & \mid \texttt{while}\, b\, \texttt{do}\, c\, \texttt{end} \end{array}$$

- *pid*, returns local processor id from $\mathbb{P}$: it introduces variation in evaluation between processes.

BSPlite local semantics

- Local semantics for local computation in each process:

$$\begin{aligned} \to^i &: cmd \times \Sigma \to T \times \Sigma \\ \Sigma &= \mathbb{X} \to \mathbb{N} \\ T &= \{\mathbf{Ok}\} \cup \{\mathbf{Wait}(c) \mid c \in cmd\} \end{aligned}$$

- $\langle c, \sigma \rangle \to^i \langle t, \sigma' \rangle$ denotes one step of local-computation with termination state t by processor with id i.
- Local semantics are standard (big-step, operational), except `sync` which stops local computation and returns the rest of the program as a continuation.

BSPlite global semantics

- Global semantics moves the computation forward globally from one super-step to the next when all p local processes has completed:

$$\rightarrow : cmd^p \times \Sigma^p \times (\Sigma^p \cup \{\Omega\})$$

- One step of global computation either:
  1. terminates correctly: $\langle C, E \rangle \rightarrow E'$
  2. synchronization incorrectly: $\langle C, E \rangle \rightarrow \Omega$
- The BSP meaning of program c in a Single Program Multiple Data (SPMD) context: $\langle [c]_{i \in \mathbb{P}}, E \rangle \rightarrow E'$.

BSPlite example programs

Buggy program from the introduction

$$
\begin{aligned}
c_{nok} = {} & [\mathtt{I} := 0]^1; \\
& [\mathtt{X} := pid]^2; \\
& \mathtt{while}\ [\mathtt{I} < 100]^3\ \mathtt{do} \\
& \quad [\mathtt{sync}]^4; \\
& \quad \mathtt{if}\ [\mathtt{X} = 0]^5\ \mathtt{then} \\
& \qquad [\mathtt{sync}]^6 \\
& \quad \mathtt{else} \\
& \qquad [\mathtt{skip}]^7 \\
& \quad [\mathtt{end}]^; \\
& \quad [\mathtt{I} := \mathtt{I} + 1]^8; \\
& \mathtt{end}
\end{aligned}
$$

Correct program

$$
\begin{aligned}
c_{ok} = {} & [\mathtt{I} := 0]^1; \\
& \mathtt{while}\ [\mathtt{I} < 100]^2\ \mathtt{do} \\
& \quad [\mathtt{sync}]^3; \\
& \quad [\mathtt{I} := \mathtt{I} + 1]^4; \\
& \mathtt{end}
\end{aligned}
$$

Problem formulation

- A program c is *synchronization error free*, if

$$\nexists E, \langle [c]_{i \in \mathbb{P}}, E \rangle \rightarrow \Omega$$

- Goal: guarantee that BSPlib programs are synchronization error free.
- c_{ok} synchronization error free, c_{nok} is not.

Replicated synchronization

- *Textually aligned synchronization*: in each super-step, all local processors stop at the same instance of the same `sync`-primitive.
- A program with textually aligned synchronization has no synchronization errors.

Replicated synchronization

- *Textually aligned synchronization*: in each super-step, all local processors stop at the same instance of the same `sync`-primitive.
- A program with textually aligned synchronization has no synchronization errors.
- *Replicated synchronization*: statically verified condition for having textually aligned synchronization.
- Program has replicated synchronization if all conditionals and loops with bodies which contains `sync` are *pid-independent*.

Replicated synchronization

- *Textually aligned synchronization*: in each super-step, all local processors stop at the same instance of the same `sync`-primitive.
- A program with textually aligned synchronization has no synchronization errors.
- *Replicated synchronization*: statically verified condition for having textually aligned synchronization.
- Program has replicated synchronization if all conditionals and loops with bodies which contains `sync` are *pid-independent*.
- A variable is *pid-independent* when it has no data- nor control-dependency on *pid*.
- *Pid-independent* variables goes through the same series of values on all processors

BSPlite example programs

Buggy program from the introduction

$$
\begin{aligned}
c_{nok} = {} & [\mathtt{I} := 0]^1; \\
& [\mathtt{X} := pid]^2; \\
& \mathtt{while}\ [\mathtt{I} < 100]^3\ \mathtt{do} \\
& \quad [\mathtt{sync}]^4; \\
& \quad \mathtt{if}\ [\mathtt{X} = 0]^5\ \mathtt{then} \\
& \quad\quad [\mathtt{sync}]^6 \\
& \quad \mathtt{else} \\
& \quad\quad [\mathtt{skip}]^7 \\
& \quad [\mathtt{end}]^; \\
& \quad [\mathtt{I} := \mathtt{I} + 1]^8; \\
& \mathtt{end}
\end{aligned}
$$

Correct program

$$
\begin{aligned}
c_{ok} = {} & [\mathtt{I} := 0]^1; \\
& \mathtt{while}\ [\mathtt{I} < 100]^2\ \mathtt{do} \\
& \quad [\mathtt{sync}]^3; \\
& \quad [\mathtt{I} := \mathtt{I} + 1]^4; \\
& \mathtt{end}
\end{aligned}
$$

Replicated synchronization: Good software engineering practice

- *Replicate synchronization* codifies good parallel software engineering practices
- The condition is simple to understand
- Makes parallel code easier to understand
- All programs we have surveyed are implicitly written in this style
- Our analysis statically verifies that BSPlib code meets this condition, and so is synchronization error free

Statical analysis for finding *pid*-independent variables

- Reformulation of type system of Barrier Inference [Aiken & Gay '98] as a data-flow analysis
- We impose stronger requirements on the analyzed program: no synchronization in branches where guard-expression is not *pid*-independent.
- Idea is to find variables and program locations which does not have a data- or control-dependency on pid
- The abstract state in the data-flow analysis for each program location contains (1) the set of variables statically guaranteed to be *pid*-independent at that point (2) the *pid*-independence of each guard-expression in which the point is nested.

Statically verifying "Replicated synchronization"

- With data-flow analysis, simple to verify that a program has replicated synchronization: all guard-conditions for if- and while-statements which contains sync has a replicated guard-conditions:

$$RS^{\sharp}(c) = \bigwedge_{(l,b,c') \in guards(c)} [\mathtt{sync}] \notin c' \vee (FV(b) \subseteq PI(l) \wedge pid \notin b)$$

Conclusion and future work

- Contributions:
  - Formulating the correctness criterion "Replicated synchronization"
  - Formalized and proved static analysis for detecting Replicated synchronization as a data-flow analysis for BSPlite
  - Implementation as a Frama-C plugin, $\sim$2000 lines of OCaml-code
- Future work includes:
  - Use as a building block for further analyses: communication, cost-analysis, ...
  - Extend target language: pointers, functions, communication, ...
  - Coq formalization