BSP-WHY: an Intermediate Language for Deductive Verification of BSP Programs

Jean Fortin and Frédéric Gava

Laboratoire d'**A**lgorithmique, **C**omplexité et **L**ogique (LACL)
Université de Paris-Est

Introduction

- A need to prove parallel programs :
  - cost of the crash of massively parallel computations
  - more and more parallel programs
- Additional difficulties :
  - Communication procedures
  - Synchronization mechanisms
  - Interleaving of instructions
- Use of Hoare semantics
  - Annotated programs
  - Generation of proof obligations

Introduction

- A need to prove parallel programs :
  - cost of the crash of massively parallel computations
  - more and more parallel programs
- Additional difficulties :
  - Communication procedures
  - Synchronization mechanisms
  - Interleaving of instructions
- Use of Hoare semantics
  - Annotated programs
  - Generation of proof obligations

Introduction

- A need to prove parallel programs :
  - cost of the crash of massively parallel computations
  - more and more parallel programs
- Additional difficulties :
  - Communication procedures
  - Synchronization mechanisms
  - Interleaving of instructions
- Use of Hoare semantics
  - Annotated programs
  - Generation of proof obligations

Introduction

- A need to prove parallel programs :
  - cost of the crash of massively parallel computations
  - more and more parallel programs
- Additional difficulties :
  - Communication procedures
  - Synchronization mechanisms
  - Interleaving of instructions
- Use of Hoare semantics
  - Annotated programs
  - Generation of proof obligations

Introduction

- A need to prove parallel programs :
  - cost of the crash of massively parallel computations
  - more and more parallel programs
- Additional difficulties :
  - Communication procedures
  - Synchronization mechanisms
  - Interleaving of instructions
- Use of Hoare semantics
  - Annotated programs
  - Generation of proof obligations

Bulk Synchronous Parallelism (BSP)

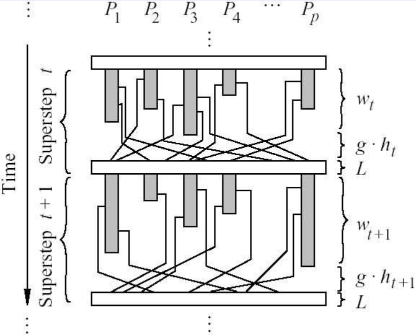
BSP computer

- *p* couples processor/memory
- with a communication network (*g*)
- and a synchronization unit (*L*)

Bulk Synchronous Parallelism (BSP)

BSP computer

- *p* couples processor/memory
- with a communication network (*g*)
- and a synchronization unit (*L*)



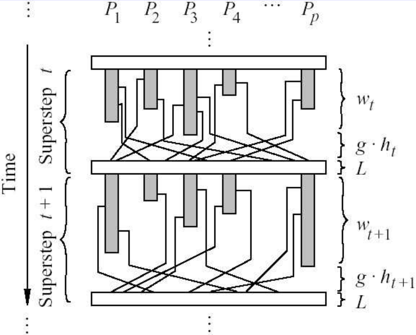
Bulk Synchronous Parallelism (BSP)

BSP computer

- *p* couples processor/memory
- with a communication network (*g*)
- and a synchronization unit (*L*)

Properties

- Determinism
- No deadlocks
- Estimation of computing time



BSPlib/PUB

Library for the BSP model :

- C Language
- Send/Receive routines
- DRMA routines
- High-performance operations (not safe)

BSPlib/PUB

Library for the BSP model :

- C Language
- Send/Receive routines
- DRMA routines
- High-performance operations (not safe)

BSPlib/PUB

Library for the BSP model :

- C Language
- Send/Receive routines
- DRMA routines
- High-performance operations (not safe)

BSPlib/PUB

Library for the BSP model :

- C Language
- Send/Receive routines
- DRMA routines
- High-performance operations (not safe)

PUB Communications

Two kinds of communications :

- Message Passing (BSMP)
  - **void** bsp_send(**int** dest,**void**∗ buffer, **int** size)
  - t_bspmsg∗ bsp_findmsg(**int** proc_id,**int** index)
- Remote Memory Access (DRMA)
  - **void** bsp_push_reg (t_bsp∗ bsp, **void**∗ ident, **int** size)
  - **void** bsp_get (t_bsp∗ bsp, **int** srcPID, **void**∗ src,**int** offset, **void**∗ dest, **int** nbytes)

Synchronisation : **void** bsp_sync(t_bsp∗ bsp)

PUB Communications

Two kinds of communications :

- Message Passing (BSMP)
  - **void** bsp_send(**int** dest,**void**∗ buffer, **int** size)
  - t_bspmsg∗ bsp_findmsg(**int** proc_id,**int** index)
- Remote Memory Access (DRMA)
  - **void** bsp_push_reg (t_bsp∗ bsp, **void**∗ ident, **int** size)
  - **void** bsp_get (t_bsp∗ bsp, **int** srcPID, **void**∗ src,**int** offset, **void**∗ dest, **int** nbytes)

Synchronisation : **void** bsp_sync(t_bsp∗ bsp)

PUB Communications

Two kinds of communications :

- Message Passing (BSMP)
  - **void** bsp_send(**int** dest,**void**$*$ buffer, **int** size)
  - t_bspmsg$*$ bsp_findmsg(**int** proc_id,**int** index)
- Remote Memory Access (DRMA)
  - **void** bsp_push_reg (t_bsp$*$ bsp, **void**$*$ ident, **int** size)
  - **void** bsp_get (t_bsp$*$ bsp, **int** srcPID, **void**$*$ src,**int** offset, **void**$*$ dest, **int** nbytes)

Synchronisation : **void** bsp_sync(t_bsp$*$ bsp)

PUB Communications

Two kinds of communications :

- Message Passing (BSMP)
  - **void** bsp_send(**int** dest,**void**∗ buffer, **int** size)
  - t_bspmsg∗ bsp_findmsg(**int** proc_id,**int** index)
- Remote Memory Access (DRMA)
  - **void** bsp_push_reg (t_bsp∗ bsp, **void**∗ ident, **int** size)
  - **void** bsp_get (t_bsp∗ bsp, **int** srcPID, **void**∗ src,**int** offset, **void**∗ dest, **int** nbytes)

Synchronisation : **void** bsp_sync(t_bsp∗ bsp)

PUB Communications

Two kinds of communications :

- Message Passing (BSMP)
  - **void** bsp_send(**int** dest,**void**∗ buffer, **int** size)
  - t_bspmsg∗ bsp_findmsg(**int** proc_id,**int** index)
- Remote Memory Access (DRMA)
  - **void** bsp_push_reg (t_bsp∗ bsp, **void**∗ ident, **int** size)
  - **void** bsp_get (t_bsp∗ bsp, **int** srcPID, **void**∗ src,**int** offset, **void**∗ dest, **int** nbytes)

Synchronisation : **void** bsp_sync(t_bsp∗ bsp)

PUB Communications

Two kinds of communications :

- Message Passing (BSMP)
  - **void** bsp_send(**int** dest,**void**∗ buffer, **int** size)
  - t_bspmsg∗ bsp_findmsg(**int** proc_id,**int** index)
- Remote Memory Access (DRMA)
  - **void** bsp_push_reg (t_bsp∗ bsp, **void**∗ ident, **int** size)
  - **void** bsp_get (t_bsp∗ bsp, **int** srcPID, **void**∗ src,**int** offset, **void**∗ dest, **int** nbytes)

Synchronisation : **void** bsp_sync(t_bsp∗ bsp)

The Why Language

Why : an intermediate language

- For program verification
- Annotated programs
- Several back-end provers (Coq, Alt-ergo, Simplify, Z3 . . .)

The Why Language

Why : an intermediate language

- For program verification
- Annotated programs
- Several back-end provers (Coq, Alt-ergo, Simplify, Z3 . . .)

The Why Language

Why : an intermediate language

- For program verification
- Annotated programs
- Several back-end provers (Coq, Alt-ergo, Simplify, Z3 . . .)

The Why Language

Why : an intermediate language

- For program verification
- Annotated programs
- Several back-end provers (Coq, Alt-ergo, Simplify, Z3 . . .)

The Why Language

Why : an intermediate language

- For program verification
- Annotated programs
- Several back-end provers (Coq, Alt-ergo, Simplify, Z3 . . .)

Language definition

- BSP-Why is extended from Why
- Additional instructions for parallel operations
- Additional notations in assertions about parallelism

Language definition

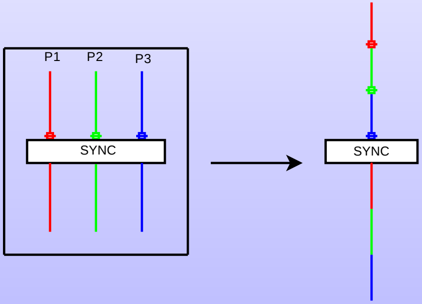
$$
\begin{array}{rcll}
\mathit{BSPWhy} & ::= & \mathit{Why} & \\
 & | & \textbf{sync} & \text{synchronisation} \\
 & | & \textbf{push}(x) & \text{Register } x \text{ for global access} \\
 & | & \textbf{put}(e, x, y) & \text{Distant writing} \\
 & | & \textbf{send}(x, e) & \text{Message passing}
\end{array}
$$

Logic extensions

- x is used to represent the value of x on the current processor
- $x < i >$ is used to represent the value of x on the processor i
- $< x >$ is used to represent the parallel variable x as an array

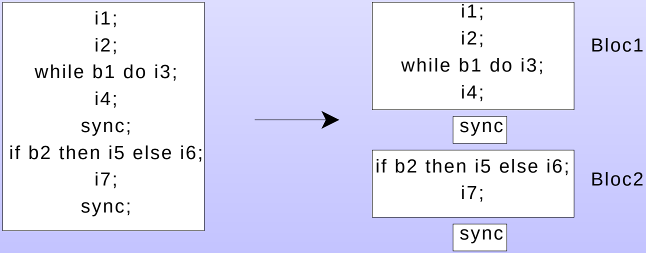
General idea of the transformation

Simulation of the parallel execution by a sequential execution



Decomposition into blocks

We extract the biggest blocks of code without synchronization :



```
i1;
i2;
while b1 do i3;
i4;
sync;
if b2 then i5 else i6;
i7;
sync;
```

→

```
i1;
i2;
while b1 do i3;
i4;
```

Bloc1

sync

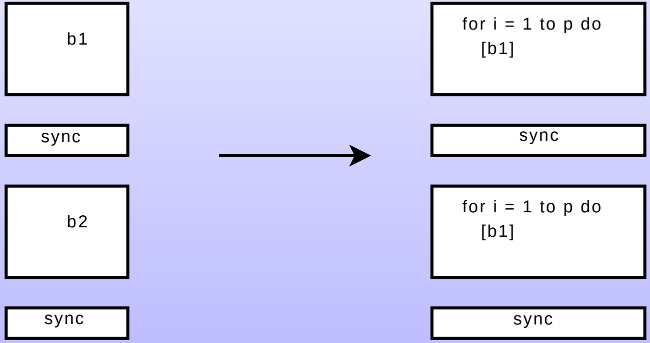
```
if b2 then i5 else i6;
i7;
```

Bloc2

sync

Decomposition into blocks

Each block is transformed into a *for* loop :



Decomposition into blocks

Need to check that the `sync` instruction match : no code such as

```
if pid=0 then sync
 else p
```

or even

```
if pid=0 then p1;sync
 else p2;sync
```

Decomposition into blocks

Need to check that the `sync` instruction match : no code such as

```
if pid=0 then sync
 else p
```

or even

```
if pid=0 then p1;sync
 else p2;sync
```

Memory management

p processors → 1 processor : need to simulate p memories in one.

- variable $x \rightarrow p$-array x
- Special arrays to store communications

Transformation of variables

BSPWhy term	Why term
`x`	`x[i]`
`<x>`	`x`
`x<j>`	`x[j]`

Variable not transformed into arrays

Some special cases :

- A variable which lives only in a sequential block
- A variable used with remote access communications

Send communications

Communications are defined in a Why prelude file.

- Messages are stored in lists
- The bsp_send function is defined as a parameter
- Send communications are done with a predicate
- The synchronisation calls each communication predicate

Send communications

Communications are defined in a Why prelude file.

- Messages are stored in lists
- The `bsp_send` function is defined as a parameter
- Send communications are done with a predicate
- The synchronisation calls each communication predicate

Send communications

Communications are defined in a Why prelude file.

- Messages are stored in lists
- The bsp_send function is defined as a parameter
- Send communications are done with a predicate
- The synchronisation calls each communication predicate

Send communications

Communications are defined in a Why prelude file.

- Messages are stored in lists
- The `bsp_send` function is defined as a parameter
- Send communications are done with a predicate
- The synchronisation calls each communication predicate

Send communications

Communications are defined in a Why prelude file.

- Messages are stored in lists
- The `bsp_send` function is defined as a parameter
- Send communications are done with a predicate
- The synchronisation calls each communication predicate

PUT / GET operations

- Memory model more complex
- A table of variables is stored
- An association table keeps records of *push* associations
- Queues for *push*, *pop*, *put* and *get* operations

PUT / GET operations

The association table is needed :

```
Proc 1    Proc 2

Push(x)   Push(y)
Push(y)   Push(x)
sync      sync
```

$P1$	$P2$
x	y
y	x

PUT / GET operations

The association table is needed :

```
Proc 1   Proc 2

Push(x)  Push(y)
Push(y)  Push(x)
sync     sync
```

$P1$	$P2$
x	y
y	x

Example : prefix calculation

- At the beginning, each processor i contains a value x_i
- At the end, each processor contains the prefix $x_0 * x_1 * \cdots * x_i$
- Useful in many calculations (FFT, n-body, graph algorithms etc.)

Example : prefix calculation

```
parameterg x: int ref

let prefixes () =
  (let y = ref (bsp_pid void + 1) in
     while(!y < nprocs) do


       bsp_send !y (cast_int !x);
       y := !y + 1
     done);

  bsp_sync;
  (
    z:=x;
    let y = ref 0 in
     while(!y < bsp_pid void) do


       z := !z + uncast_int (bsp_findmsg !y 0);
       y := !y + 1
     done )
```

Example : prefix calculation

```
parameterg x: int ref

let prefixes () = {}
  (let y = ref (bsp_pid void + 1) in
     while(!y < nprocs) do
       {
 invariant envCsendIs(j,bsp_pid + 1,y,j,x)
   variant nprocs − y
       }
       bsp_send !y (cast_int !x);
       y := !y + 1
     done);
  { envCsendIs(j,bsp_pid + 1,nprocs−1,j,x) }
  bsp_sync;
  (
    z:=x;
    let y = ref 0 in
     while(!y < bsp_pid void) do
       {
 invariant z=x+sigma_prefix(<x>, y)
   variant bsp_pid − y
       }
       z := !z + uncast_int (bsp_findmsg !y 0);
       y := !y + 1
     done )
       { z=sigma_prefix(<x>, bsp_pid)}
```

Conclusion

Summary :

- BSP-Why is an extension of the Why language for BSP programs
- BSP-Why programs are transformed into Why programs
- The proof obligations are generated by Why

Conclusion

Summary :

- BSP-Why is an extension of the Why language for BSP programs
- BSP-Why programs are transformed into Why programs
- The proof obligations are generated by Why

Conclusion

Summary :

- BSP-Why is an extension of the Why language for BSP programs
- BSP-Why programs are transformed into Why programs
- The proof obligations are generated by Why

Conclusion

Summary :

- BSP-Why is an extension of the Why language for BSP programs
- BSP-Why programs are transformed into Why programs
- The proof obligations are generated by Why

Conclusion

Summary :

- BSP-Why is an extension of the Why language for BSP programs
- BSP-Why programs are transformed into Why programs
- The proof obligations are generated by Why

Outlook

- The aim is to generate BSP-Why code from a BSP-C program
- Use of Frama-C with the Jessie plugin
- Use this work to prove MPI programs with only global operations

Outlook

- The aim is to generate BSP-Why code from a BSP-C program
- Use of Frama-C with the Jessie plugin
- Use this work to prove MPI programs with only global operations

Outlook

- The aim is to generate BSP-Why code from a BSP-C program
- Use of Frama-C with the Jessie plugin
- Use this work to prove MPI programs with only global operations