Mechanised Semantics of BSP Routines with Subgroup Synchronisation

Jean Fortin & Frédéric Gava

Laboratory of **A**lgorithms, **C**omplexity and **L**ogic (LACL)
University of Paris-East

Outline

Outline

Bridging Model: Bulk Synchronous Parallelism (BSP)

The BSP computer

Defined by:

- **p** pairs CPU/memory
- Communication network (**g**)
- Synchronisation unit (**L**)
- Super-steps execution

Properties:

Bridging Model: Bulk Synchronous Parallelism (BSP)

The BSP computer

Defined by:

- **p** pairs CPU/memory
- Communication network (**g**)
- Synchronisation unit (**L**)
- Super-steps execution

Properties:

Bridging Model: Bulk Synchronous Parallelism (BSP)

The BSP computer

Defined by:

- **p** pairs CPU/memory
- Communication network (**g**)
- Synchronisation unit (**L**)
- Super-steps execution

Properties:

- "Confluent"
- "Deadlock-free"
- Predictable performances

Bridging Model: Bulk Synchronous Parallelism (BSP)

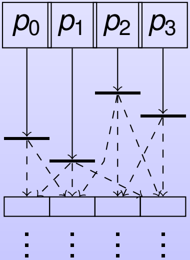
The BSP computer

Defined by:

- **p** pairs CPU/memory
- Communication network (**g**)
- Synchronisation unit (**L**)
- Super-steps execution

Properties:

- "Confluent"
- "Deadlock-free"
- Predictable performances



local computations

communication ($\otimes$**g**)

barrier ($\oplus$**L**)

next super-step

Bridging Model: Bulk Synchronous Parallelism (BSP)

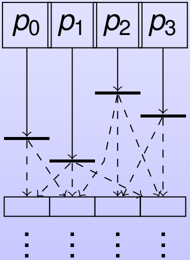
The BSP computer

Defined by:

- **p** pairs CPU/memory
- Communication network (**g**)
- Synchronisation unit (**L**)
- Super-steps execution

Properties:

- "Confluent"
- "Deadlock-free"
- Predictable performances



local computations

communication ($\otimes$**g**)

barrier ($\oplus$**L**)

next super-step

Bridging Model: Bulk Synchronous Parallelism (BSP)

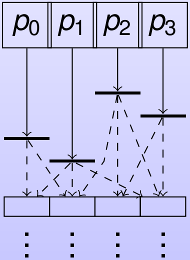
The BSP computer

Defined by:

- **p** pairs CPU/memory
- Communication network (**g**)
- Synchronisation unit (**L**)
- Super-steps execution

Properties:

- “Confluent”
- “Deadlock-free”
- Predictable performances



local computations

communication ($\otimes$**g**)

barrier ($\oplus$**L**)

next super-step

Bridging Model: Bulk Synchronous Parallelism (BSP)

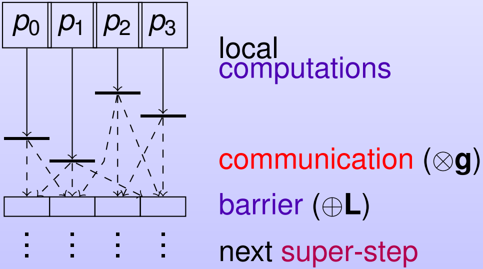
The BSP computer

Defined by:

- **p** pairs CPU/memory
- Communication network (**g**)
- Synchronisation unit (**L**)
- Super-steps execution

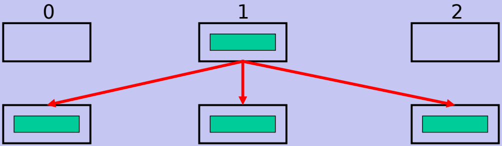
Properties:

- “Confluent”
- “Deadlock-free”
- Predictable performances



Examples: broadcasting a values

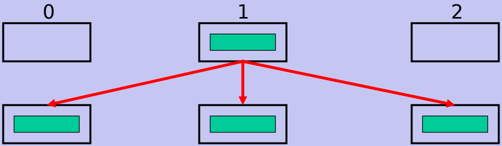
Direct broadcast (one super-step)



Cost $\equiv \mathbf{p} \times \mathbf{g} \times n \; + \; \mathbf{L}$

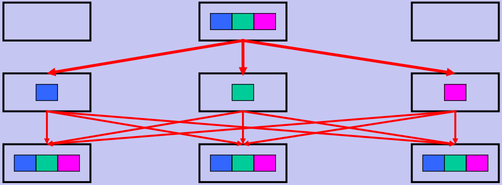
Examples: broadcasting a values

Direct broadcast (one super-step)



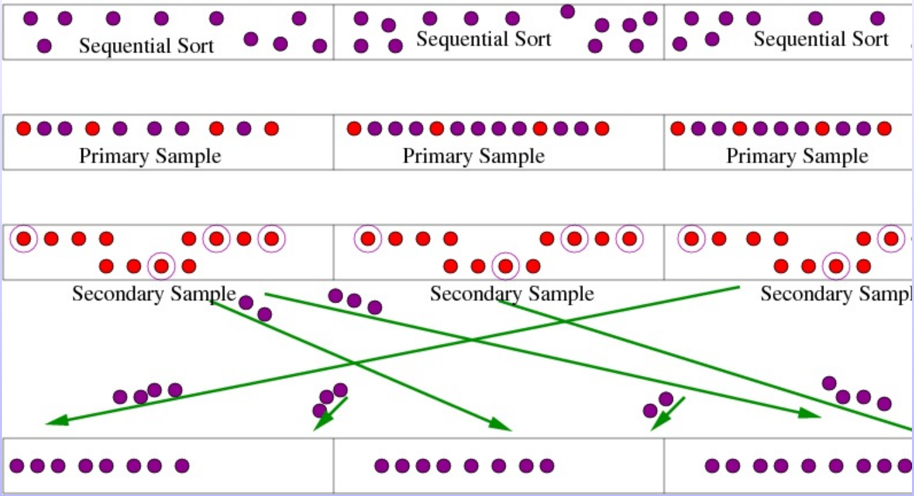
Cost $\equiv \mathbf{p} \times \mathbf{g} \times n \; + \; \mathbf{L}$

Broadcast with two super-steps



Cost $\equiv 2 \times \mathbf{g} \times n \; + \; 2 \times \mathbf{L}$

Parallel Sorting by Regular Sampling (PSRS)



BSP Imperative Programming

Languages and libraries

1. Dedicated Languages: NestStep, BSP++, BSP-Python, . . .
2. BSPLib for C and Java
3. BSPGPU, Ct, Hamma, JBSP, JPUB, . . .
4. MPI collective operations

BSP Imperative Programming

Languages and libraries

1. Dedicated Languages: NestStep, BSP++, BSP-Python, ...
2. BSPLib for C and Java
3. BSPGPU, Ct, Hamma, JBSP, JPUB, ...
4. MPI collective operations

BSP Imperative Programming

Languages and libraries

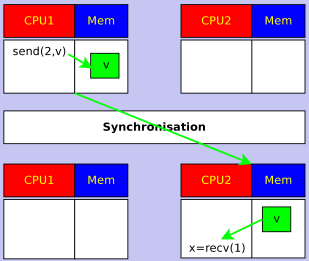
1. Dedicated Languages: NestStep, BSP++, BSP-Python, . . .
2. BSPLib for C and Java
3. BSPGPU, Ct, Hamma, JBSP, JPUB, . . .
4. MPI collective operations

BSP Imperative Programming

Languages and libraries

1. Dedicated Languages: NestStep, BSP++, BSP-Python, . . .
2. BSPLib for C and Java
3. BSPGPU, Ct, Hamma, JBSP, JPUB, . . .
4. MPI collective operations

Communications: BSMP

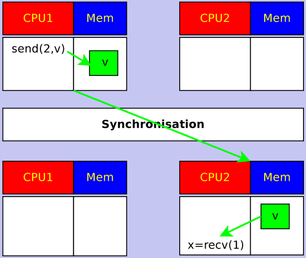


BSP Imperative Programming

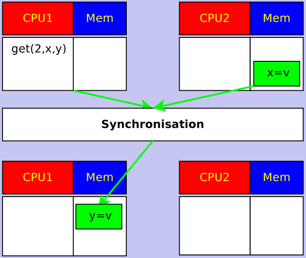
Languages and libraries

1. Dedicated Languages: NestStep, BSP++, BSP-Python, . . .
2. BSPLib for C and Java
3. BSPGPU, Ct, Hamma, JBSP, JPUB, . . .
4. MPI collective operations

Communications: BSMP



Communications: DRMA



Examples of C primitives

BSMP and DRMA

- Typical BSMP routines:
  - `bsp_send(grp,dest,buffer,size)`
  - `bsp_nmsgs(grp)`
  - `msg* bsp_findmsg(grp,proc_id,index)`
- Typical DRMA routines:
  - `bsp_push_reg(grp,ident,size)`
  - `bsp_get(grp,srcPID,src,offset,dest,nbytes)`
- `bsp_sync(grp)` (barrier)

MPI collective operations

`MPI_Scatter(sendbuf,sendcount,sendtype,recvbuf,recvcount,recvtype,root,comm)`

`MPI_Gather(sendbuf,sendcount,sendtype,recvbuf,recvcount,recvtype,root,comm)`

Examples of C primitives

BSMP and DRMA

- Typical BSMP routines:
  - `bsp_send(grp,dest,buffer,size)`
  - `bsp_nmsgs(grp)`
  - `msg* bsp_findmsg(grp,proc_id,index)`
- Typical DRMA routines:
  - `bsp_push_reg(grp,ident,size)`
  - `bsp_get(grp,srcPID,src,offset,dest,nbytes)`
- `bsp_sync(grp)` (barrier)

MPI collective operations

`MPI_Scatter(sendbuf,sendcount,sendtype,recvbuf,recvcount,recvtype,root,comm)`

`MPI_Gather(sendbuf,sendcount,sendtype,recvbuf,recvcount,recvtype,root,comm)`

Examples of C primitives

BSMP and DRMA

- Typical BSMP routines:
  - `bsp_send(grp,dest,buffer,size)`
  - `bsp_nmsgs(grp)`
  - `msg* bsp_findmsg(grp,proc_id,index)`
- Typical DRMA routines:
  - `bsp_push_reg(grp,ident,size)`
  - `bsp_get(grp,srcPID,src,offset,dest,nbytes)`
- `bsp_sync(grp)` (barrier)

MPI collective operations

`MPI_Scatter(sendbuf,sendcount,sendtype,recvbuf,recvcount,recvtype,root,comm)`

`MPI_Gather(sendbuf,sendcount,sendtype,recvbuf,recvcount,recvtype,root,comm)`

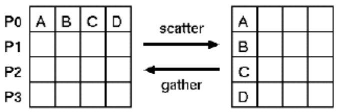
Examples of C primitives

BSMP and DRMA

- Typical BSMP routines:
  - `bsp_send(grp,dest,buffer,size)`
  - `bsp_nmsgs(grp)`
  - `msg* bsp_findmsg(grp,proc_id,index)`
- Typical DRMA routines:
  - `bsp_push_reg(grp,ident,size)`
  - `bsp_get(grp,srcPID,src,offset,dest,nbytes)`
- `bsp_sync(grp)` (barrier)

MPI collective operations

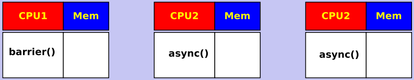
`MPI_Scatter(sendbuf,sendcount,sendtype,recvbuf,recvcount,recvtype,root,comm)`



`MPI_Gather(sendbuf,sendcount,sendtype,recvbuf,recvcount,recvtype,root,comm)`

Common Bugs in BSP Imperative Programs

"Deadlock"

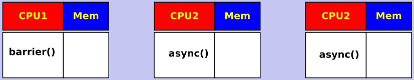


Data race, non-determinism

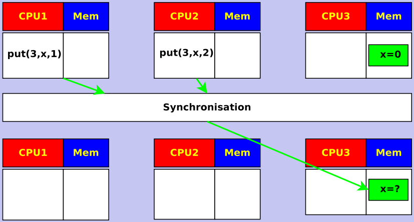
Out-of-bound errors

Common Bugs in BSP Imperative Programs

“Deadlock”



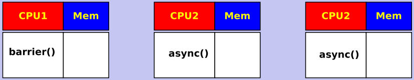
Data race, non-determinism



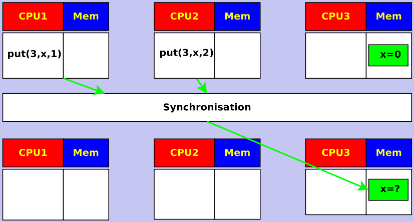
Out-of-bound errors

Common Bugs in BSP Imperative Programs

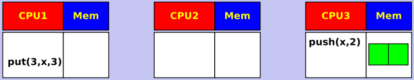
“Deadlock”



Data race, non-determinism



Out-of-bound errors



Subgroup Synchronisation

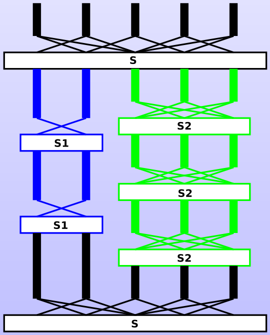
Allows the synchronisation of a subset of processes

Advantages

- Take advantage of hybrid models (better performance)
- Close to MPI's collective operations

Drawbacks

- More complex programs
- BSP cost model is lost



Subgroup Synchronisation

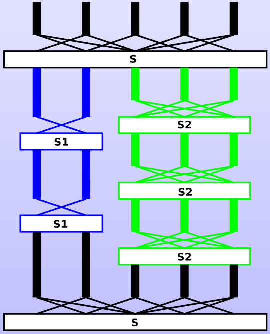
Allows the synchronisation of a subset of processes

Advantages

- Take advantage of hybrid models (better performance)
- Close to MPI's collective operations

Drawbacks

- More complex programs
- BSP cost model is lost



Subgroup Synchronisation

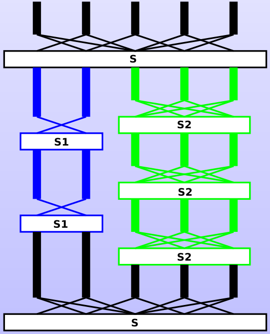
Allows the synchronisation of a subset of processes

Advantages

- Take advantage of hybrid models (better performance)
- Close to MPI's collective operations

Drawbacks

- More complex programs
- BSP cost model is lost



Examples of C primitives

Using the PUB

- `BSP_WORLD:t_bsp` ⇒ all the processors
- `bsp_dup(grp,dup)` ⇒ a copy of a subgroup
- `bsp_partition(grp,sub,nr,partition)` ⇒ new
- `bsp_done(grp)` ⇒ destroy a subgroup

Using MPI

- `MPI_COMM_WORLD:MPI_Comm` ⇒ all the processors
- `MPI_Comm_dup(comm,newcomm)` ⇒ Duplicates
- `MPI_Comm_free(comm)` ⇒ free a communicator
- `MPI_Comm_split(comm,color,key,newcomm)` ⇒ Creates new communicators based on colors and keys
- ... (merge, union, intersection, differences)

Examples of C primitives

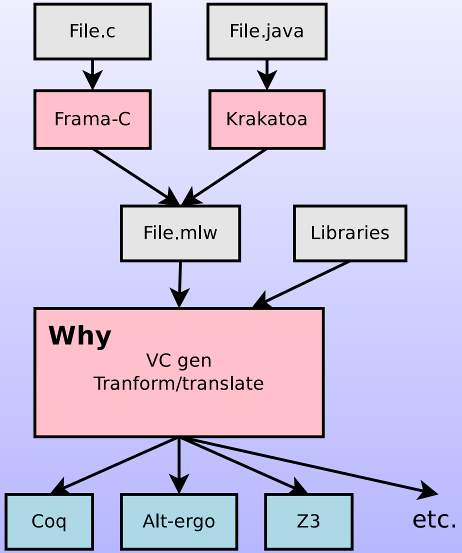
Using the PUB

- `BSP_WORLD:t_bsp` ⇒ all the processors
- `bsp_dup(grp,dup)` ⇒ a copy of a subgroup
- `bsp_partition(grp,sub,nr,partition)` ⇒ new
- `bsp_done(grp)` ⇒ destroy a subgroup

Using MPI

- `MPI_COMM_WORLD:MPI_Comm` ⇒ all the processors
- `MPI_Comm_dup(comm,newcomm)` ⇒ Duplicates
- `MPI_Comm_free(comm)` ⇒ free a communicator
- `MPI_Comm_split(comm,color,key,newcomm)` ⇒ Creates new communicators based on colors and keys
- ... (merge, union, intersection, differences)

The Why tool


File.c
File.java
Frama-C
Krakatoa
File.mlw
Libraries
Why
VC gen
Tranform/translate
Coq
Alt-ergo
Z3
etc.

The WhyML Language for Deductive Verification

```
let sqrt (n:int) =
  { n ≥ 0 }
  let count = ref 0 in
  let sum = ref 1 in
  while !sum ≤ n do
    { invariant count ≥ 0 and n ≥ count∗count
                and sum = (count+1)∗(count+1)
        variant n − sum }
    count ←!count + 1;
    sum ←!sum + 2 ∗ !count + 1
  done;
  !count
  { result ≥ 0 and
    result ∗ result ≤ n < (result+1)∗(result+1) }
```

The WhyML Language for Deductive Verification

```
let sqrt (n:int) =
  { n ≥ 0 }
  let count = ref 0 in
  let sum = ref 1 in
  while !sum ≤ n do
    { invariant count ≥ 0 and n ≥ count∗count
               and sum = (count+1)∗(count+1)
      variant n − sum }
    count ←!count + 1;
    sum ←!sum + 2 ∗ !count + 1
  done;
  !count
  { result ≥ 0 and
    result ∗ result ≤ n < (result+1)∗(result+1) }
```

The WhyML Language for Deductive Verification

```
let sqrt (n:int) =
  { n ≥ 0 }
  let count = ref 0 in
  let sum = ref 1 in
  while !sum ≤ n do
    { invariant count ≥ 0 and n ≥ count∗count
                and sum = (count+1)∗(count+1)
      variant n − sum }
    count ←!count + 1;
    sum ←!sum + 2 ∗ !count + 1
  done;
  !count
  { result ≥ 0 and
    result ∗ result ≤ n < (result+1)∗(result+1) }
```

The WhyML Language for Deductive Verification

```
let sqrt (n:int) =
  { n ≥ 0 }
  let count = ref 0 in
  let sum = ref 1 in
  while !sum ≤ n do
    { invariant count ≥ 0 and n ≥ count∗count
               and sum = (count+1)∗(count+1)
      variant n − sum }
    count ←!count + 1;
    sum ←!sum + 2 ∗ !count + 1
  done;
  !count
  { result ≥ 0 and
    result ∗ result ≤ n < (result+1)∗(result+1) }
```

The BSP-Why tool

Generalities

- BSP-WhyML extends WhyML with:
  - Additional instructions for parallel operations
  - Additional notations in assertions about parallelism
- Automatic transformation to Why code (sequentialisation)

Main idea

The BSP-Why tool

Generalities

- BSP-WhyML extends WhyML with:
  - Additional instructions for parallel operations
  - Additional notations in assertions about parallelism
- Automatic transformation to Why code (sequentialisation)

Main idea

The BSP-Why tool

Generalities

- BSP-WhyML extends WhyML with:
  - Additional instructions for parallel operations
  - Additional notations in assertions about parallelism
- Automatic transformation to Why code (sequentialisation)

Main idea

The BSP-Why tool

Generalities

- BSP-WhyML extends WhyML with:
  - Additional instructions for parallel operations
  - Additional notations in assertions about parallelism
- Automatic transformation to Why code (sequentialisation)

Main idea

The BSP-Why tool

Generalities

- BSP-WhyML extends WhyML with:
  - Additional instructions for parallel operations
  - Additional notations in assertions about parallelism
- Automatic transformation to Why code (sequentialisation)

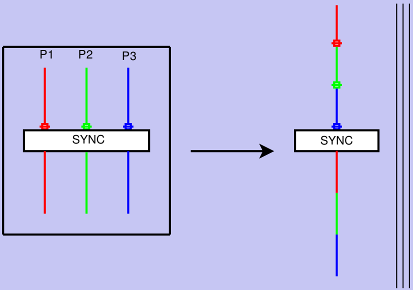
Main idea

The BSP-Why tool

Generalities

- BSP-WhyML extends WhyML with:
  - Additional instructions for parallel operations
  - Additional notations in assertions about parallelism
- Automatic transformation to Why code (sequentialisation)

Main idea

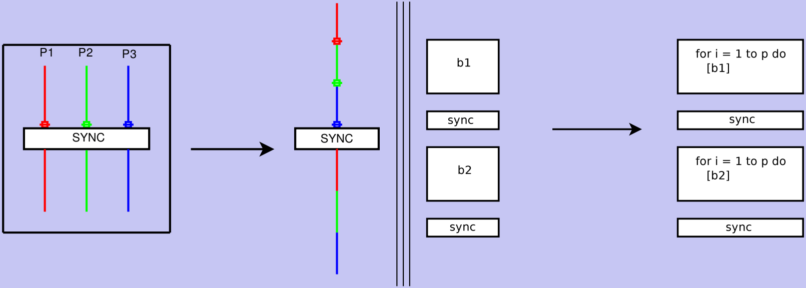


The BSP-Why tool

Generalities

- BSP-WhyML extends WhyML with:
  - Additional instructions for parallel operations
  - Additional notations in assertions about parallelism
- Automatic transformation to Why code (sequentialisation)

Main idea



The BSP-Why tool (2)

Language definition

BSPWhy	::=	*Why*	
	|	**sync**	synchronisation and parameters
	|	**push**(x)	Register x for global access
	|	**put**(e, x, y)	Distant writing
	|	**send**(x, e)	Message passing

Logic extensions

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

The BSP-Why tool (2)

Language definition

BSPWhy	$::=$	*Why*	
	$\mid$	**sync**	synchronisation and parameters
	$\mid$	**push**(x)	Register x for global access
	$\mid$	**put**(e, x, y)	Distant writing
	$\mid$	**send**(x, e)	Message passing

Logic extensions

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

The BSP-Why tool (2)

Language definition

BSPWhy	::=	*Why*	
	|	**sync**	synchronisation and parameters
	|	**push**(x)	Register x for global access
	|	**put**(e, x, y)	Distant writing
	|	**send**(x, e)	Message passing

Logic extensions

- x, to represent the value of x on the current processor
- $x\langle i \rangle$, to represent the value of x on the processor i
- $\langle x \rangle$, to represent the parallel variable x as an array

The BSP-Why tool (2)

Language definition

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

Logic extensions

- x, to represent the value of x on the current processor
- $x{<}i{>}$, to represent the value of x on the processor i
- $<x>$, to represent the parallel variable x as an array

The BSP-Why tool (2)

Language definition

BSPWhy	::=	*Why*	
	|	**sync**	synchronisation and parameters
	|	**push**(x)	Register x for global access
	|	**put**(e, x, y)	Distant writing
	|	**send**(x, e)	Message passing

Logic extensions

- x, to represent the value of x on the current processor
- $x\!<\!i\!>$, to represent the value of x on the processor i
- $<\!x\!>$, to represent the parallel variable x as an array

Outline

Mechanised Semantics ...

Mechanised semantics allow for a better confidence

Big-step/natural semantics

- Big-step (natural) semantics are defined as a reference
- Co-inductive semantics for infinite programs

Small-step semantics

- More precise simulation of the execution (interleaving)
- Use of continuations (code after synchronisation)
- More convenient to prove code transformation

Mechanised Semantics ...

Mechanised semantics allow for a better confidence

Big-step/natural semantics

- Big-step (natural) semantics are defined as a reference
- Co-inductive semantics for infinite programs

Small-step semantics

- More precise simulation of the execution (interleaving)
- Use of continuations (code after synchronisation)
- More convenient to prove code transformation

Mechanised Semantics ...

Mechanised semantics allow for a better confidence

Big-step/natural semantics

- Big-step (natural) semantics are defined as a reference
- Co-inductive semantics for infinite programs

Small-step semantics

- More precise simulation of the execution (interleaving)
- Use of continuations (code after synchronisation)
- More convenient to prove code transformation

... In Coq

Use of the Coq proof assistant

- All semantics defined in Coq
- Inductive and CoInductive definitions

Proof of basic properties

- Confluence of semantics
- Exclusive between finite and infinite rules
- Equivalence Big-step $\Leftrightarrow$ Small-step

... In Coq

Use of the Coq proof assistant

- All semantics defined in Coq
- Inductive and CoInductive definitions

Proof of basic properties

- Confluence of semantics
- Exclusive between finite and infinite rules
- Equivalence Big-step $\Leftrightarrow$ Small-step

Big-Step Semantics: Examples of Local Rules

$$\frac{}{s, \textbf{pid} \Downarrow^i s, i} \qquad \frac{}{s, \textbf{nprocs} \Downarrow^i s, \textbf{p}}$$

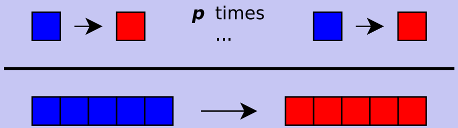
$$\frac{s, e_1 \Downarrow^i s', v \qquad s'[x \leftarrow v], e_2 \Downarrow^i s'', o}{s, \textbf{let } x = e_1 \textbf{ in } e_2 \Downarrow^i s'', o}$$

$$\frac{s, e_1 \Downarrow^i s', \textbf{SYNC}(C, e')}{s, \textbf{let } x = e_1 \textbf{ in } e_2 \Downarrow^i s', \textbf{SYNC}(C, \textbf{let } x = e' \textbf{ in } e_2)}$$

$$\frac{s, e \Downarrow^i s', \textbf{SYNC}(C, e')}{s, x := e \Downarrow^i s', \textbf{SYNC}(C, x := e')}$$

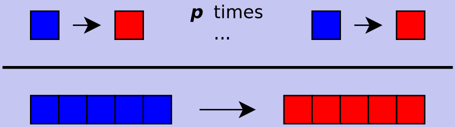
Big-Step Semantics (without subgroup)

Execution without synchronisation

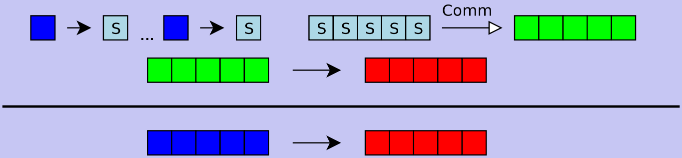


Big-Step Semantics (without subgroup)

Execution without synchronisation



Execution with synchronisation



Diverging Big-Step Semantics (without subgroup)

$$\frac{\exists i \quad s_i, e_i \Downarrow^i_\infty}{\langle (s_0, e_0), \ldots, (s_{\mathbf{p}-1}, e_{\mathbf{p}-1}) \rangle \Downarrow_\infty}$$

$$\frac{\forall i \quad s_i, e_i \Downarrow^i s'_i, \mathbf{SYNC}(e'_i) \qquad \mathbf{AllComm}\{\langle (s'_0, e'_0), \ldots, (s'_{\mathbf{p}-1}, e'_{\mathbf{p}-1}) \rangle\} \Downarrow_\infty}{\langle (s_0, e_0), \ldots, (s_{\mathbf{p}-1}, e_{\mathbf{p}-1}) \rangle \Downarrow_\infty}$$

Diverging Big-Step Semantics (without subgroup)

$$\frac{\exists i \quad s_i, e_i \Downarrow^i_\infty}{\langle (s_0, e_0), \ldots, (s_{\mathbf{p}-1}, e_{\mathbf{p}-1}) \rangle \Downarrow_\infty}$$

$$\frac{\forall i \quad s_i, e_i \Downarrow^i s'_i, \mathbf{SYNC}(e'_i) \qquad \mathbf{AllComm}\{\langle (s'_0, e'_0), \ldots, (s'_{\mathbf{p}-1}, e'_{\mathbf{p}-1}) \rangle\} \Downarrow_\infty}{\langle (s_0, e_0), \ldots, (s_{\mathbf{p}-1}, e_{\mathbf{p}-1}) \rangle \Downarrow_\infty}$$

Small-Step Semantics (without subgroup)

Naive solutions

- $\langle \ldots (s_i, \textbf{bsp\_sync}; e_i) \ldots \rangle \rightharpoonup \langle \ldots (s'_i, e_i) \ldots \rangle$
  $\Rightarrow$ Impossible to evaluate: **if** b **then bsp_sync else** e
- $s, \textbf{bsp\_sync} \overset{i}{\rightharpoonup} s, \mathit{Wait}(\textbf{skip})$
  $\langle \ldots (s_i, \mathit{Wait}(e_i)) \ldots \rangle \rightharpoonup \langle \ldots (s_i, e_i) \ldots \rangle$(Tesson, Loulergue)
  $\Rightarrow$ Impossible to evaluate: $(e_1; \textbf{bsp\_sync}); e_2$
- Congruence "$(e_1; \textbf{bsp\_sync}); e_2 \equiv e_1; (\textbf{bsp\_sync}; e_2)$" (Fortin, Gava)
- Use of contexts $\langle \ldots \Delta[\textbf{bsp\_sync}] \ldots \rangle$

Chosen solution: continuations (à la Blazy/Leroy)

$$s, \textbf{nprocs} \bullet \kappa \overset{i}{\rightharpoonup} s, \textbf{p} \bullet \kappa$$

$$s, \textbf{let } x = e_1 \textbf{ in } e_2 \bullet \kappa \overset{i}{\rightharpoonup} s, e_1 \bullet (\textbf{let } x = \_ \textbf{ in } e_2) \bullet \kappa$$

$$s, v \bullet (\textbf{let } x = \_ \textbf{ in } e_2) \bullet \kappa \overset{i}{\rightharpoonup} s[x \leftarrow v], e_2 \bullet \kappa$$

Small-Step Semantics (without subgroup)

Naive solutions

- $\langle \ldots (s_i, \textbf{bsp\_sync}; e_i) \ldots \rangle \rightharpoonup \langle \ldots (s'_i, e_i) \ldots \rangle$
  $\Rightarrow$ Impossible to evaluate: **if** b **then bsp_sync else** e
- $s, \textbf{bsp\_sync} \overset{i}{\rightharpoonup} s, \mathit{Wait}(\textbf{skip})$
  $\langle \ldots (s_i, \mathit{Wait}(e_i)) \ldots \rangle \rightharpoonup \langle \ldots (s_i, e_i) \ldots \rangle$(Tesson, Loulergue)
  $\Rightarrow$ Impossible to evaluate: $(e_1; \textbf{bsp\_sync}); e_2$
- Congruence "$(e_1; \textbf{bsp\_sync}); e_2 \equiv e_1; (\textbf{bsp\_sync}; e_2)$" (Fortin, Gava)
- Use of contexts $\langle \ldots \Delta[\textbf{bsp\_sync}] \ldots \rangle$

Chosen solution: continuations (à la Blazy/Leroy)

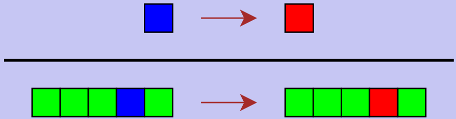
$$s, \textbf{nprocs} \bullet \kappa \overset{i}{\rightharpoonup} s, \textbf{p} \bullet \kappa$$

$$s, \textbf{let } x = e_1 \textbf{ in } e_2 \bullet \kappa \overset{i}{\rightharpoonup} s, e_1 \bullet (\textbf{let } x = \_ \textbf{ in } e_2) \bullet \kappa$$

$$s, v \bullet (\textbf{let } x = \_ \textbf{ in } e_2) \bullet \kappa \overset{i}{\rightharpoonup} s[x \leftarrow v], e_2 \bullet \kappa$$

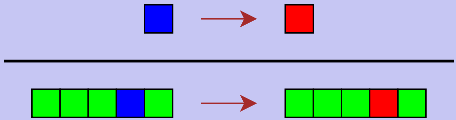
Small-Step Semantics (without subgroup)

Local execution

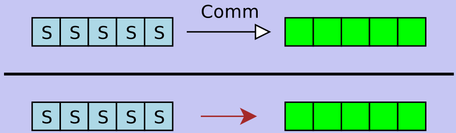


Small-Step Semantics (without subgroup)

Local execution

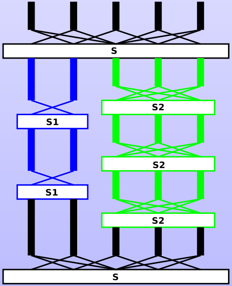


Synchronisation

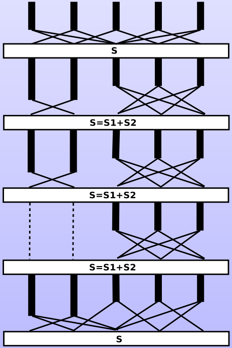


Big-Step Semantics with Subgroup Synchronisation

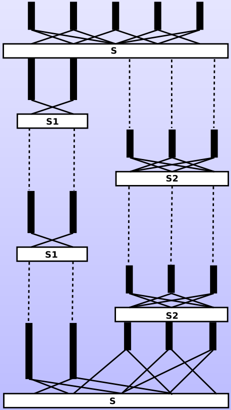
Machine execution



AllSub option

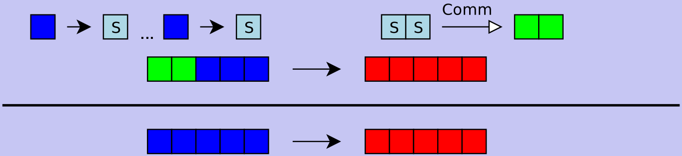


Diamond option



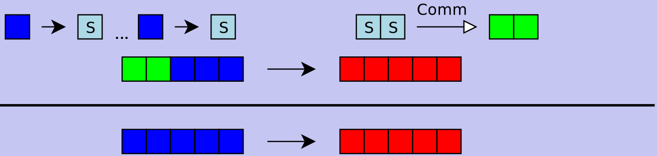
Big-Step Semantics with Subgroup Synchronisation

One synchronisation (**Diamond**)

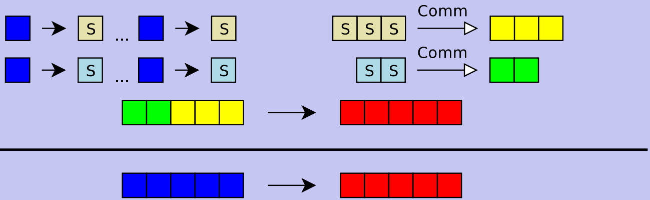


Big-Step Semantics with Subgroup Synchronisation

One synchronisation (**Diamond**)



All synchronisations (**AllSub**)



Big-Step Semantics with Subgroup Synchronisation

Diamond variation:

$$\frac{\exists C\ \forall i \in C \quad s_i, e_i \Downarrow^i s'_i, \textbf{SYNC}(C, e'_i) \qquad \textbf{CommDia}\{C, (s'_0, e'_0), \ldots (s'_{\mathbf{p}-1}, e'_{\mathbf{p}-1})\} \Downarrow_{Diam} (s''_0, v_0), \ldots (s''_{\mathbf{p}-1}, v_{\mathbf{p}-1})}{(s_0, e_0), \ldots (s_{\mathbf{p}-1}, e_{\mathbf{p}-1}) \Downarrow_{Diam} (s''_0, v_0), \ldots (s''_{\mathbf{p}-1}, v_{\mathbf{p}-1})}$$

AllSub variation:

$$\frac{\{0, \ldots, \mathbf{p}-1\} = N \oplus C_1 \oplus \cdots \oplus C_k \qquad \forall i \in C_j \quad s_i, e_i \Downarrow^i s'_i, \textbf{SYNC}(C_j, e'_j) \qquad \forall i \in N \quad s_i, e_i \Downarrow^i s'_i, v \qquad \textbf{AllCommSub}\{C_1 \ldots C_k, v, (s'_0, e'_0), \ldots (s'_{\mathbf{p}-1}, e'_{\mathbf{p}-1})\} \Downarrow_{All} (s''_0, v_0), \ldots (s''_{\mathbf{p}-1}, v_{\mathbf{p}-1})}{(s_0, e_0), \ldots (s_{\mathbf{p}-1}, e_{\mathbf{p}-1}) \Downarrow_{All} (s''_0, v_0), \ldots (s''_{\mathbf{p}-1}, v_{\mathbf{p}-1})}$$

Diverging rules follow these rules

Big-Step Semantics with Subgroup Synchronisation

Diamond variation:

$$\frac{\exists C\ \forall i \in C \quad s_i, e_i \Downarrow^i s'_i, \textbf{SYNC}(C, e'_i) \qquad \textbf{CommDia}\{C, (s'_0, e'_0), \ldots (s'_{\mathbf{p}-1}, e'_{\mathbf{p}-1})\} \Downarrow_{Diam} (s''_0, v_0), \ldots (s''_{\mathbf{p}-1}, v_{\mathbf{p}-1})}{(s_0, e_0), \ldots (s_{\mathbf{p}-1}, e_{\mathbf{p}-1}) \Downarrow_{Diam} (s''_0, v_0), \ldots (s''_{\mathbf{p}-1}, v_{\mathbf{p}-1})}$$

AllSub variation:

$$\frac{\{0, \ldots, \mathbf{p}-1\} = N \oplus C_1 \oplus \cdots \oplus C_k \qquad \forall i \in C_j \quad s_i, e_i \Downarrow^i s'_i, \textbf{SYNC}(C_j, e'_i) \qquad \forall i \in N \quad s_i, e_i \Downarrow^i s'_i, v \qquad \textbf{AllCommSub}\{C_1 \ldots C_k, v, (s'_0, e'_0), \ldots (s'_{\mathbf{p}-1}, e'_{\mathbf{p}-1})\} \Downarrow_{All} (s''_0, v_0), \ldots (s''_{\mathbf{p}-1}, v_{\mathbf{p}-1})}{(s_0, e_0), \ldots (s_{\mathbf{p}-1}, e_{\mathbf{p}-1}) \Downarrow_{All} (s''_0, v_0), \ldots (s''_{\mathbf{p}-1}, v_{\mathbf{p}-1})}$$

Diverging rules follow these rules

Big-Step Semantics with Subgroup Synchronisation

Diamond variation:

$$\frac{\exists C\ \forall i \in C \quad s_i, e_i \Downarrow^i s_i', \mathbf{SYNC}(C, e_i') \qquad \mathbf{CommDia}\{C, (s_0', e_0'), \ldots (s_{\mathbf{p}-1}', e_{\mathbf{p}-1}')\} \Downarrow_{Diam} (s_0'', v_0), \ldots (s_{\mathbf{p}-1}'', v_{\mathbf{p}-1})}{(s_0, e_0), \ldots (s_{\mathbf{p}-1}, e_{\mathbf{p}-1}) \Downarrow_{Diam} (s_0'', v_0), \ldots (s_{\mathbf{p}-1}'', v_{\mathbf{p}-1})}$$

AllSub variation:

$$\frac{\{0, \ldots, \mathbf{p}-1\} = N \oplus C_1 \oplus \cdots \oplus C_k \qquad \forall i \in C_j \quad s_i, e_i \Downarrow^i s_i', \mathbf{SYNC}(C_j, e_i') \qquad \forall i \in N \quad s_i, e_i \Downarrow^i s_i', v \qquad \mathbf{AllCommSub}\{C_1 \ldots C_k, v, (s_0', e_0'), \ldots (s_{\mathbf{p}-1}', e_{\mathbf{p}-1}')\} \Downarrow_{All} (s_0'', v_0), \ldots (s_{\mathbf{p}-1}'', v_{\mathbf{p}-1})}{(s_0, e_0), \ldots (s_{\mathbf{p}-1}, e_{\mathbf{p}-1}) \Downarrow_{All} (s_0'', v_0), \ldots (s_{\mathbf{p}-1}'', v_{\mathbf{p}-1})}$$

Diverging rules follow these rules

Small-Step Semantics with Subgroup Synchronisation

$$\frac{s_i, e_i \bullet \kappa_i \xrightharpoonup{i} s'_i, e'_i \bullet \kappa'_i}{\langle (\ldots, (s_i, e_i \bullet \kappa_i), \ldots) \rangle \rightharpoonup \langle \ldots, (s'_i, e'_i \bullet \kappa'_i), \ldots \rangle}$$

$$\frac{\exists C\ \forall i \in C \qquad O_i \equiv \textbf{bsp\_sync}\ C \bullet \kappa_i}{\langle (s_0, O_0), \ldots, (s_{\mathbf{p}-1}, O_{\mathbf{p}-1}) \rangle \rightharpoonup \textbf{CommDia}\{C, \langle (s_0, O_0), \ldots, (s_{\mathbf{p}-1}, O_{\mathbf{p}-1}) \rangle\}}$$

Small-Step Semantics with Subgroup Synchronisation

$$\frac{s_i, e_i \bullet \kappa_i \overset{i}{\rightharpoonup} s'_i, e'_i \bullet \kappa'_i}{\langle (\ldots, (s_i, e_i \bullet \kappa_i), \ldots) \rangle \rightharpoonup \langle \ldots, (s'_i, e'_i \bullet \kappa'_i), \ldots \rangle}$$

$$\frac{\exists C\ \forall i \in C \qquad O_i \equiv \textbf{bsp\_sync}\ C \bullet \kappa_i}{\langle (s_0, O_0), \ldots, (s_{\mathbf{p}-1}, O_{\mathbf{p}-1}) \rangle \rightharpoonup \textbf{CommDia}\{C, \langle (s_0, O_0), \ldots, (s_{\mathbf{p}-1}, O_{\mathbf{p}-1}) \rangle\}}$$

Results

Properties about semantics

- All semantics are confluent (Lemmas 1,2,7)
- Finite and diverging rules are mutually exclusive (4, 5)
- Finite Big-step and small-step semantics are equivalent (6)
- Two equivalent semantics for subgroup synchronisation (3)

Results

Properties about semantics

- All semantics are confluent (Lemmas 1,2,7)
- Finite and diverging rules are mutually exclusive (4, 5)
- Finite Big-step and small-step semantics are equivalent (6)
- Two equivalent semantics for subgroup synchronisation (3)

Benchmarks

Language	Rules	1	2	3	4	5	6	7	ALL
Our Why-ML	140	40	$\star$	$\star$	23	10	90	26	670
BSP-Why-ML	270	130	$\star$	$\star$	66	22	350	100	1300
With Subgroups	320	$\star$	416	531	85	35	490	446	1500
CompCert	513	1700	$\star$	$\star$	1200	100	?	1800	Big
IMP	30	12	$\star$	$\star$	14	8	53	11	135

Results

Properties about semantics

- All semantics are confluent (Lemmas 1,2,7)
- Finite and diverging rules are mutually exclusive (4, 5)
- Finite Big-step and small-step semantics are equivalent (6)

Analysis

1. Work(BSP-With-Subgroup) $\equiv$ 10 $\otimes$ Work(core-seq-language)
2. C language $\Rightarrow$ 10 years of work for team
3. Next talk $\Rightarrow$ 100 years (work for C+BSP+Subgroups)

BSP-Why-ML	270	130	$\star$	$\star$	66	22	350	100	1300
With Subgroups	320	$\star$	416	531	85	35	490	446	1500
CompCert	513	1700	$\star$	$\star$	1200	100	?	1800	Big
IMP	30	12	$\star$	$\star$	14	8	53	11	135

Outline

1. BSP Programming
2. Mechanised Semantics
3. Conclusion

Conclusion

BSP-WHY-ML

- BSP-WHY-ML is a BSP extension of WHY-ML
- BSP-WHY programs are transformed into WHY programs
- Subgroups take into account hierarchical architectures

Coq semantics

Conclusion

BSP-WHY-ML

- BSP-WHY-ML is a BSP extension of WHY-ML
- BSP-WHY programs are transformed into WHY programs
- Subgroups take into account hierarchical architectures

Coq semantics

Conclusion

BSP-WHY-ML

- BSP-WHY-ML is a BSP extension of WHY-ML
- BSP-WHY programs are transformed into WHY programs
- Subgroups take into account hierarchical architectures

Coq semantics

- 2 big-steps and small-step semantics
- Proofs of equivalance, confluence, exclusive, etc.
- That do not scale ! (sizes of the proofs)

Conclusion

BSP-WHY-ML

- BSP-WHY-ML is a BSP extension of WHY-ML
- BSP-WHY programs are transformed into WHY programs
- Subgroups take into account hierarchical architectures

Coq semantics

- 2 big-steps and small-step semantics
- Proofs of equivalence, confluence, exclusive, etc.
- That do not scale ! (sizes of the proofs)

Conclusion

BSP-WHY-ML

- BSP-WHY-ML is a BSP extension of WHY-ML
- BSP-WHY programs are transformed into WHY programs
- Subgroups take into account hierarchical architectures

Coq semantics

- 2 big-steps and small-step semantics
- Proofs of equivalence, confluence, exclusive, *etc.*
- That do not scale ! (sizes of the proofs)

Conclusion

BSP-WHY-ML

- BSP-WHY-ML is a BSP extension of WHY-ML
- BSP-WHY programs are transformed into WHY programs
- Subgroups take into account hierarchical architectures

Coq semantics

- 2 big-steps and small-step semantics
- Proofs of equivalence, confluence, exclusive, *etc.*
- That do not scale ! (sizes of the proofs)

Conclusion

BSP-WHY-ML

- BSP-WHY-ML is a BSP extension of WHY-ML
- BSP-WHY programs are transformed into WHY programs
- Subgroups take into account hierarchical architectures

Coq semantics

- 2 big-steps and small-step semantics
- Proofs of equivalence, confluence, exclusive, *etc.*
- That do not scale ! (sizes of the proofs)

Conclusion

BSP-WHY-ML

- BSP-WHY-ML is a BSP extension of WHY-ML
- BSP-WHY programs are transformed into WHY programs
- Subgroups take into account hierarchical architectures

Coq semantics

- 2 big-steps and small-step semantics
- Proofs of equivalence, confluence, exclusive, *etc.*
- That do not scale ! (sizes of the proofs)

Perspectives (Ongoing/Future Work)

Close to this subject

- Application to graph algorithms (Big-Data)
- Fully Mechanised BSP-WHY (diverging rules)
- Find how to better automate the proofs $\Rightarrow$ less human work
- Forget C, switch to a higher-level language such as Java

Perspectives (Ongoing/Future Work)

Close to this subject

- Application to graph algorithms (Big-Data)
- Fully Mechanised BSP-WHY (diverging rules)
- Find how to better automate the proofs ⇒ less human work
- Forget C, switch to a higher-level language such as Java

Long term (team)

- Algorithms, semantics and verification tools for multi-ML
- Fully verified multi-BSP model checker ?
- Application to graph algorithms ?
- BSP/Skeleton abstract interpretation

Merci !