Imperative Characterization of BSP Algorithms

Yoann Marquer & Frédéric Gava

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

Outline

Outline

1. BSP Programming
2. Algorithmic Equivalences
3. Conclusion

Introduction

What is an algorithm?

- Not a Turing machine
- Not a programming language
- Every designer writes them in different forms

Introduction

What is an algorithm?

- Not a Turing machine
- Not a programming language
- Every designer writes them in different forms

So?

- Axiomatic definition of a sequential algorithm and ASM
- Equivalence with a core programming language

Introduction

What is an algorithm?

- Not a Turing machine
- Not a programming language
- Every designer writes them in different forms

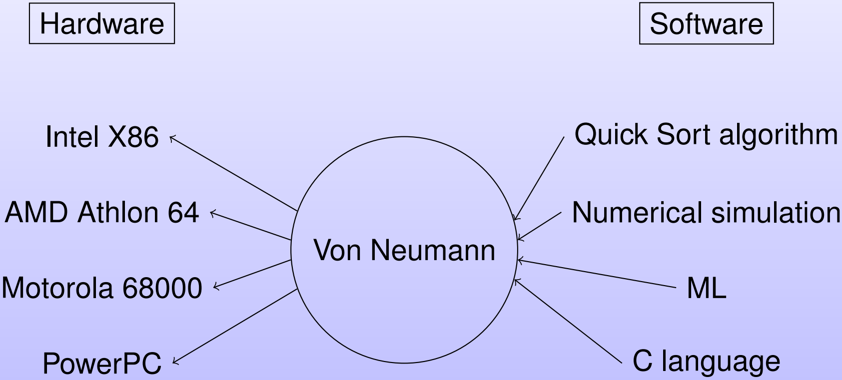
So?

- Axiomatic definition of a sequential algorithm and ASM
- Equivalence with a core programming language

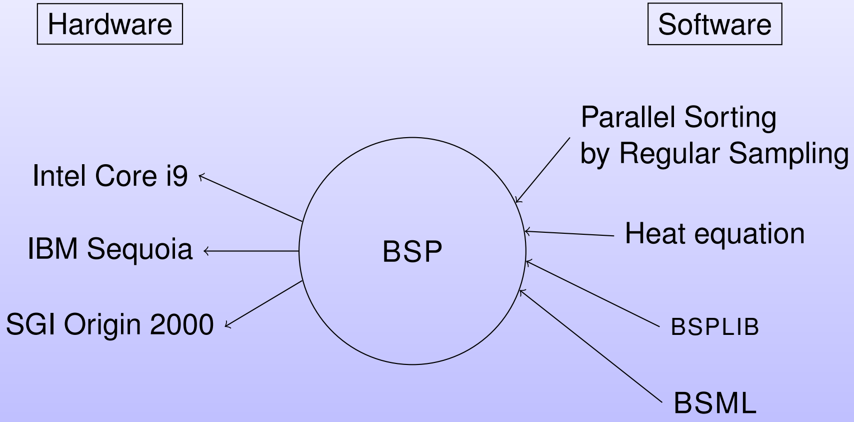
And then?

- Parallel and/or distributed ASM (Gurevich *and al*)
- Without equivalences with a core programming language
- Because no cost model and not a bridging model

What is a bridging model? For sequential computing



What is a bridging model? For HPC computing



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)

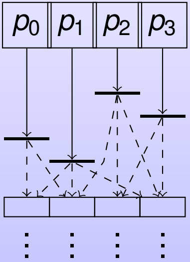
The BSP computer

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)

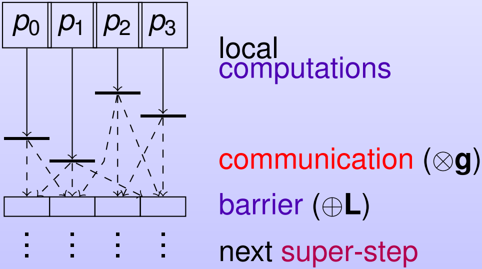
The BSP computer

Defined by:

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

Properties:

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



Bridging model: Bulk Synchronous Parallelism (BSP)

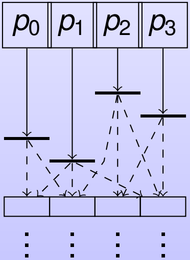
The BSP computer

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)

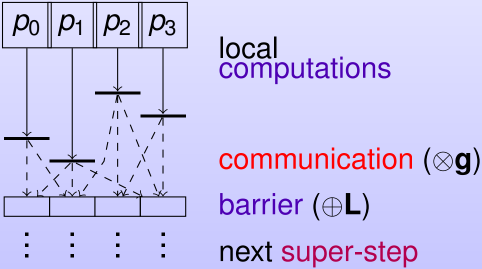
The BSP computer

Defined by:

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

Properties:

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



Bridging model: Bulk Synchronous Parallelism (BSP)

The BSP computer

Defined by:

Pro and cons

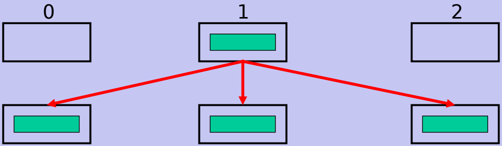
Pro	Cons
Cost model	Not all parallel patterns
Structured parallelism	Too regular
Easy to learn	No asynchronous warnings

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

next super-step

Examples: broadcasting a value

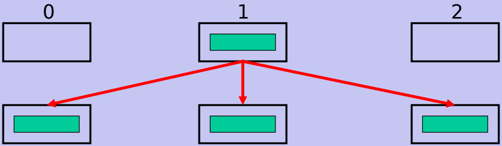
Direct broadcast (one super-step)



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

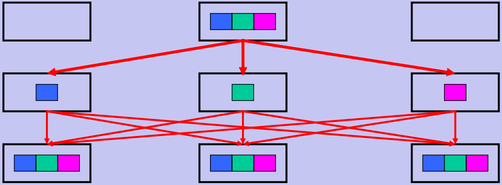
Examples: broadcasting a value

Direct broadcast (one super-step)



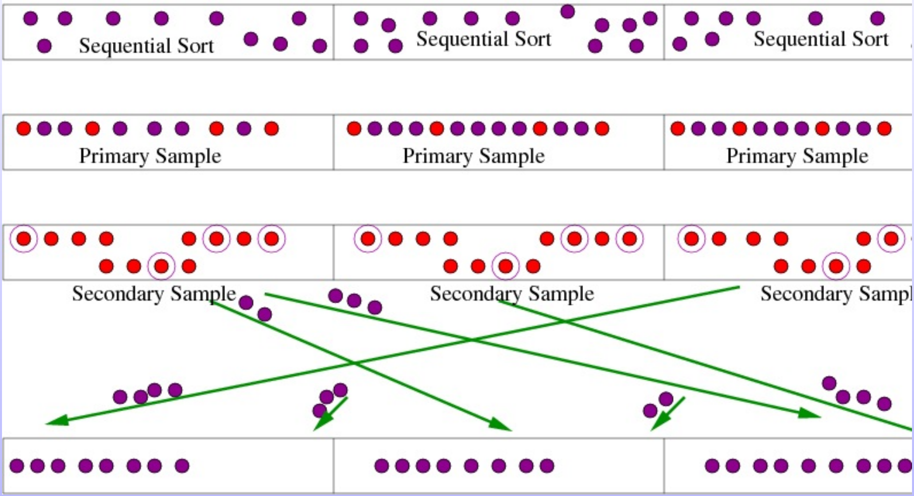
Cost $\equiv \mathbf{p} \times \mathbf{g} \times n \quad + \quad \mathbf{L}$

Broadcast with two super-steps



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

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

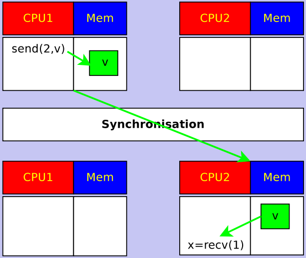
1. Dedicated languages: NestStep, BSP++, BSP-Python, ...
2. BSPLib for C and Java
3. BSPGPU, Ct, Hamma, JBSP, JPUB, ...
4. MPI collective operations

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

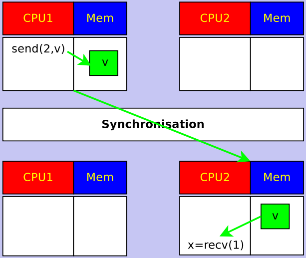


BSP imperative programming

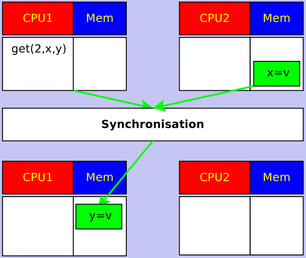
Languages and libraries

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

MPI collective operations

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

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

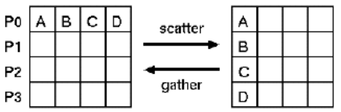
Examples of C primitives

BSMP and DRMA

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

MPI collective operations

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



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

One example: PSRS sorting

```
PSRS_Sorting(array<T> tab)
 begin
   seq_sort(tab);                          (* local sorting *)
   s1←sample(tab);                         (* first sampling *)
   for i←0 to nprocs−1 do
     bsp_send(i,s1[i],size(s1[i]));
   done;
   bsp_sync();
   s_all←[];                               (* second sampling *)
   for i←0 to nprocs−1 do
     merge(s_all,bsp_findmsg(i,0));
   done;
   s2←sample(s_all);
   ...
   bsp_sync();
   ...
 end
```

One example: PSRS sorting

```
PSRS_Sorting(array<T> tab)
 begin
   seq_sort(tab);                    (* local sorting *)
   s1←sample(tab);                   (* first sampling *)
```

Problems

1. It is rather a code than an algorithm ...
2. Thus, what is an BSP algorithm?
3. Is my programming language rich enough to programming all the BSP algorithms?

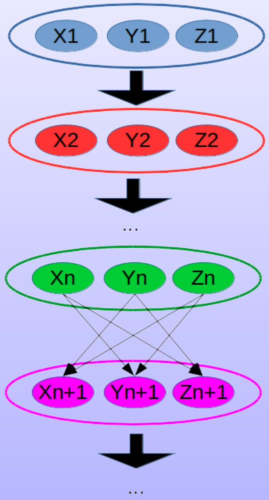
```
   done;
   s2←sample(s_all);
   ...
   bsp_sync();
   ...
 end
```

Outline

Execution using the axiomatic definition

Sequential algorithm:

BSP algorithm:



Axiomatic definition (1)

Sequential:

Axiom 1; Sequential Time

1. a set of states $S(A_{seq})$
2. a set of initial states $I(A_{seq}) \subseteq S(A_{seq})$
3. a transition function $\tau_{A_{seq}} : S(A_{seq}) \to S(A_{seq})$

Axiomatic definition (1)

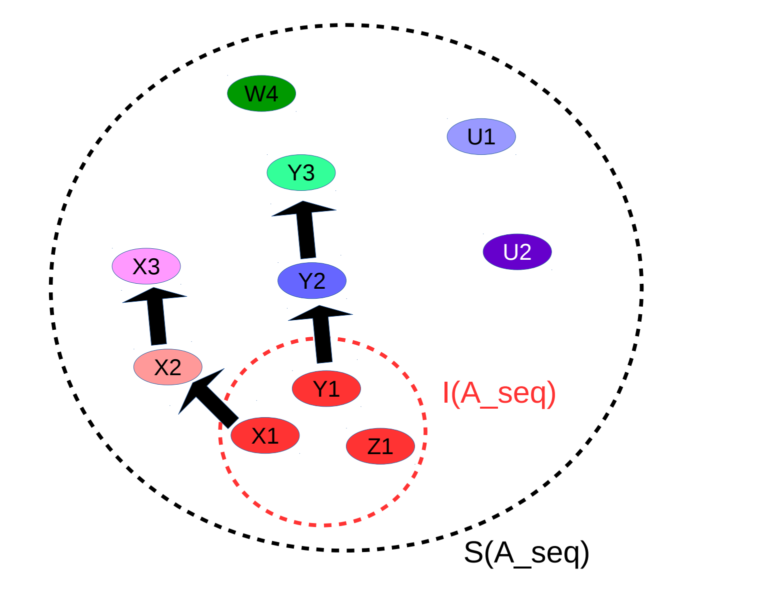
Sequ

Axio

1

2

3



Axiomatic definition (1)

Sequential:

Axiom 1; Sequential Time

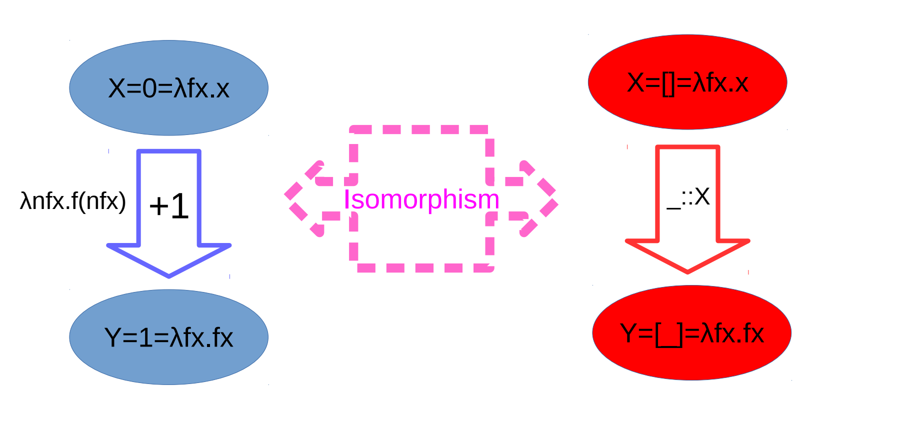
1. a set of states $S(A_{seq})$
2. a set of initial states $I(A_{seq}) \subseteq S(A_{seq})$
3. a transition function $\tau_{A_{seq}} : S(A_{seq}) \rightarrow S(A_{seq})$

Axiom 2; Abstract States

1. states are first-order structures and closed by isomorphism
2. $\tau_{A_{seq}}$ commutes with the isomorphism

Axiomatic definition (1)

Sequential:



2. $\tau_{A_{seq}}$ commutes with the isomorphism

Axiomatic definition (1)

Sequential:

Axiom 1; Sequential Time

1. a set of states $S(A_{seq})$
2. a set of initial states $I(A_{seq}) \subseteq S(A_{seq})$
3. a transition function $\tau_{A_{seq}} : S(A_{seq}) \rightarrow S(A_{seq})$

Axiom 2; Abstract States

1. states are first-order structures and closed by isomorphism
2. $\tau_{A_{seq}}$ commutes with the isomorphism

BSP:

Axiom 1; Sequential Time

1. a set of states $S(A_{BSP})$
2. a set of initial states $I(A_{BSP}) \subseteq S(A_{BSP})$
3. a transition function $\tau_{A_{BSP}} : S(A_{BSP}) \rightarrow S(A_{BSP})$

Axiom 2; Abstract States

1. states are p-tuples and closed by p-isomorphism
2. $\tau_{A_{BSP}}$ commutes with the p-isomorphism and preserves the size of the p-tuples.

Axiomatic definition (2)

Sequential:

Axiom 3; Bounded Exploration

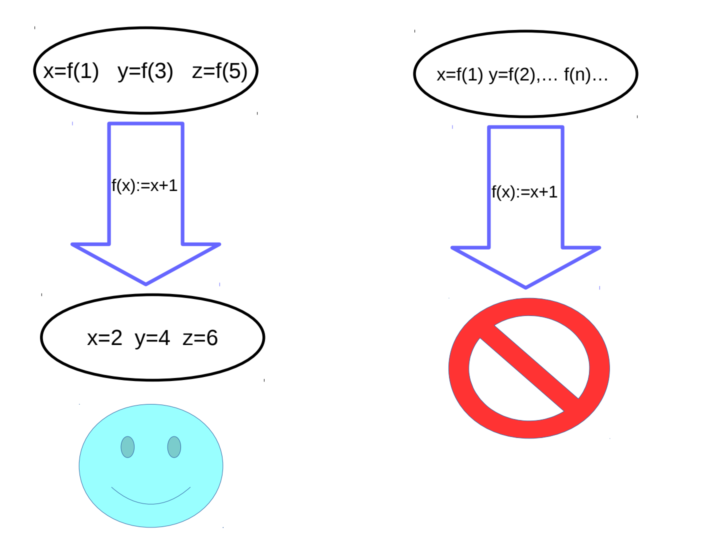
For every algorithm A there exists a finite set T of terms such that for every state X and Y, if the elements of T have the same interpretations on X and Y then $\Delta(A, X) = \Delta(A, Y)$.

Axiomatic definition (2)

Sequ

Axio

For e
exist
such
Y, if
same
Y the



Axiomatic definition (2)

Sequential:

Axiom 3; Bounded Exploration

For every algorithm A there exists a finite set T of terms such that for every state X and Y, if the elements of T have the same interpretations on X and Y then $\Delta(A, X) = \Delta(A, Y)$.

BSP:

Axiom 3; Bounded Exploration

Idem but $X = (X^1, \dots, X^p)$ and $Y = (Y^1, \dots, Y^q)$ and $p = q$ and $\vec{\Delta}(A, X) = \vec{\Delta}(A, Y)$

Axiomatic definition (2)

Sequential:

Axiom 3; Bounded Exploration

For every algorithm A there exists a finite set T of terms such that for every state X and Y, if the elements of T have the same interpretations on X and Y then $\Delta(A, X) = \Delta(A, Y)$.

BSP:

Axiom 3; Bounded Exploration

Idem but $X = (X^1, \ldots, X^p)$ and $Y = (Y^1, \ldots, Y^q)$ and $p = q$ and $\vec{\Delta}(A, X) = \vec{\Delta}(A, Y)$

Axiom 4; Barrier

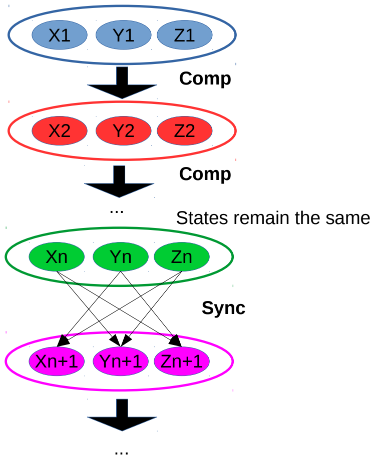
For every BSP algorithm A there exists two applications $\mathbf{comp}_A : M(A) \to M(A)$ and $\mathbf{sync}_A : S(A) \to S(A)$ such that

Axiomatic definition (2)

Sequential:

Axiom 3; Bounded E

For every algorithm
exists a finite set T
such that for every s
Y, if the elements o
same interpretations
Y then $\Delta(A, X) = \Delta$



nded Exploration

$(X^1, \ldots, X^p)$ and
$Y^q)$ and $p = q$
$= \vec{\Delta}(A, Y)$

rier

algorithm A there
plications
$) \to M(A)$ and
$\to S(A)$ such that

$$\tau_A\left(X^1, \ldots, X^p\right) = \begin{cases} \left(\mathbf{comp}_A(X^1), \ldots, \mathbf{comp}_A(X^p)\right) & \text{if } \mathbf{comp}_A(X^i) \neq X^i \\ \mathbf{sync}_A\left(X^1, \ldots, X^p\right) & \text{otherwise} \end{cases}$$

ASM and BSP-ASM

Sequential:

Definition ASM

$(\Pi, S(B), I(B))$

BSP:

Definition BSP-ASM

$(\Pi, S(B), I(B), \mathbf{sync}_B)$ where $S(B)$ contains p-tuples of structures.

ASM and BSP-ASM

Sequential:

Definition ASM

$(\Pi, S(B), I(B))$

ASM's programs Π

$$
\begin{array}{rll}
\Pi =_{def} & f(t_1, \ldots, t_\alpha) := t_0 \\
| & \texttt{if } F \texttt{ then } \Pi_1 \\
& \quad \texttt{else } \Pi_2 \texttt{ endif} \\
| & \texttt{par } \Pi_1 \| \ldots \| \Pi_n \texttt{ endpar}
\end{array}
$$

BSP:

Definition BSP-ASM

$(\Pi, S(B), I(B), \mathbf{sync}_B)$ where $S(B)$ contains p-tuples of structures.

ASM and BSP-ASM

Sequential:

Definition ASM

$(\Pi, S(B), I(B))$

ASM's programs Π

$$\begin{array}{rll} \Pi =_{def} & & f(t_1, \ldots, t_\alpha) := t_0 \\ & | & \texttt{if } F \texttt{ then } \Pi_1 \\ & & \quad \texttt{else } \Pi_2 \texttt{ endif} \\ & | & \texttt{par } \Pi_1 \| \ldots \| \Pi_n \texttt{ endpar} \end{array}$$

BSP:

Definition BSP-ASM

$(\Pi, S(B), I(B), \textbf{sync}_B)$ where $S(B)$ contains p-tuples of structures.

Extension of Π

$$\begin{array}{rll} \Pi =_{def} & & \ldots \\ & | & f_{send}(t_1, \ldots, t_\alpha) := t^{send} \\ & | & t^{rcv} := f_{rcv}(t_1, \ldots, t_\alpha) \end{array}$$

Operational semantics

Sequential

$$\Delta(f(t_1, \ldots, t_\alpha) := t_0, X) =_{def} \{(f, \overline{t_1}^X, \ldots, \overline{t_\alpha}^X, \overline{t_0}^X)\}$$

$$\Delta(\texttt{if } F \texttt{ then } \Pi_1 \texttt{ else } \Pi_2 \texttt{ endif}, X) =_{def} \Delta(\Pi_i, X)$$

$$\text{where } i = \begin{cases} 1 \text{ if } F \text{ is true on } X \\ 2 \text{ else} \end{cases}$$

$$\Delta(\texttt{par } \Pi_1 \| \ldots \| \Pi_n \texttt{ endpar}, X) =_{def} \Delta(\Pi_1, X) \cup \cdots \cup \Delta(\Pi_n, X)$$

Operational semantics

Sequential

$$\Delta(f(t_1,\ldots,t_\alpha) := t_0, X) =_{def} \{(f, \overline{t_1}^X, \ldots, \overline{t_\alpha}^X, \overline{t_0}^X)\}$$

$$\Delta(\texttt{if } F \texttt{ then } \Pi_1 \texttt{ else } \Pi_2 \texttt{ endif}, X) =_{def} \Delta(\Pi_i, X)$$

$$\text{where } i = \begin{cases} 1 \text{ if } F \text{ is true on } X \\ 2 \text{ else} \end{cases}$$

$$\Delta(\texttt{par } \Pi_1 \| \ldots \| \Pi_n \texttt{ endpar}, X) =_{def} \Delta(\Pi_1, X) \cup \cdots \cup \Delta(\Pi_n, X)$$

BSP

$$\overrightarrow{\Delta}(\Pi, (X^1, \cdots, X^{\mathbf{p}})) =_{def} \begin{cases} (\Delta(\Pi, X^1), \cdots \Delta(\Pi, X^{\mathbf{p}})) & \text{if } \exists 1 \leq i \leq \mathbf{p} \\ & \text{such that } \Delta(\Pi, X^i) \neq \emptyset \\ \mathbf{sync}_\Pi(X^1, \cdots, X^{\mathbf{p}}) & \text{else} \end{cases}$$

IMP and BSP-IMP

Sequential:

$$\begin{aligned} c =_{def}\ & f(t_1, \ldots, t_\alpha) := t_0 \\ & \mid \texttt{if}\ F\ \{P_1\}\ \texttt{else}\ \{P_2\} \\ & \mid \texttt{while}\ F\ \{P\} \end{aligned}$$

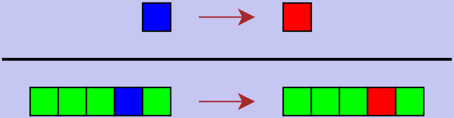
BSP:

$$\begin{aligned} c =_{def}\ & \cdots \\ & \mid f_{send}(t_1, \ldots, t_\alpha) := t^{send} \\ & \mid t^{rcv} := f_{rcv}(t_1, \ldots, t_\alpha) \\ & \mid f_{sync}() \end{aligned}$$

$$P =_{def} \quad \epsilon \mid c; P$$

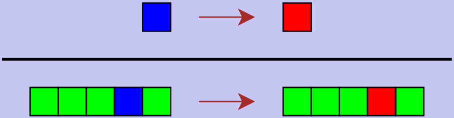
Small-Step Semantics

Local execution

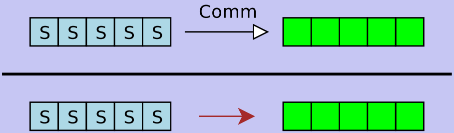


Small-Step Semantics

Local execution



Synchronisation



Small-Step Semantics rules

Local rules:

$$f(t_1, \ldots, t_\alpha) := t_0; P \star X \succ^i P \star X \oplus (f, \overline{t_1}^X, \ldots, \overline{t_\alpha}^X, \overline{t_0}^X)$$

$$\texttt{while } F \ \{P_1\}; P_2 \star X \succ^i P_1 \fatsemi \texttt{while } F \ \{P_1\}; P_2 \star X$$

if F is true in X

...

One global rule:

$$\frac{\exists 1 \leq i \leq \mathbf{p} \quad P^i \star X^i \succ^i P'^i \star X'^i}{\langle P^1 \star X^1, \ldots, P^{\mathbf{p}} \star X^{\mathbf{p}} \rangle \succ \langle P'^1 \star X'^1, \ldots, P'^{\mathbf{p}} \star X'^{\mathbf{p}} \rangle}$$

Fair Simulation

A computation model M_1 simulates M_2 if $\forall P_2 \in M_2 \; \exists \; P_1 \in M_1$:

1. "Not too many intemediate variables"

2. $\exists \; d \in \mathbb{N} \setminus \{0\}$ and $e \in \mathbb{N}$ (depending only on P_2) such that, for every execution $\vec{Y}$ of P_2, $\exists$ an execution $\vec{X}$ of P_1:

$$time(P_1, X_0) = d \times time(P_2, Y_0) + e$$

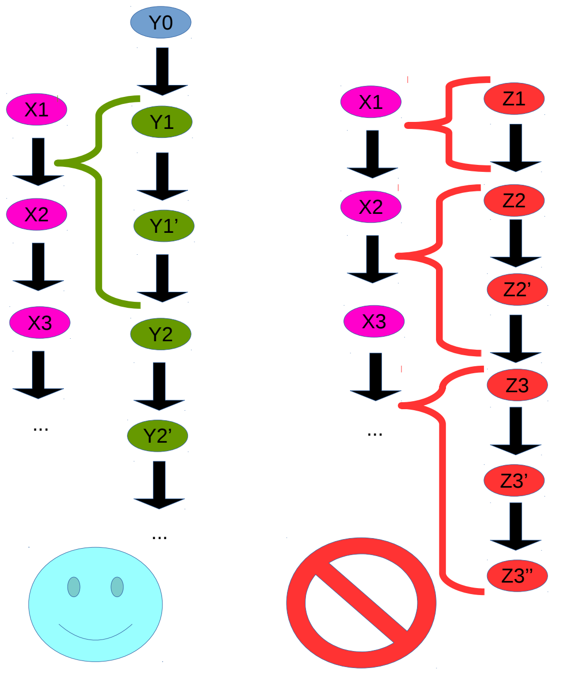
Fair Simu

A compu

1. "Not

2. $\exists$ *d*

for e



$P_1 \in M_1$:

such that,

P_1:

Compilation: from BSP-IMP to BSP-ASM

Main code

$$\Pi_P \equiv \texttt{if } \neg b_{\text{wait}} \texttt{ then } \underset{P_j \in \mathcal{G}(P)}{\texttt{par}} \texttt{ if } b_{P_j} \texttt{ then } [\![P_j]\!]_{asm} \texttt{ endpar endif}$$

Compilation: from BSP-IMP to BSP-ASM

Main code

$$\Pi_P \equiv \texttt{if } \neg b_{\text{wait}} \texttt{ then } \underset{P_j \in \mathcal{G}(P)}{\texttt{par}} \texttt{ if } b_{P_j} \texttt{ then } [\![P_j]\!]_{asm} \texttt{ endpar endif}$$

The control flow graph

$$\mathcal{G}(c;P) \equiv \mathcal{G}(c); P \cup \mathcal{G}(P)$$
$$\mathcal{G}(\texttt{while } F\ \{P\}) \equiv \mathcal{G}(P); \texttt{while } F\ \{P\}$$
$$\ldots$$

Compilation: from BSP-IMP to BSP-ASM

Main code

$$\Pi_P \equiv \texttt{if } \neg b_{\text{wait}} \texttt{ then } \underset{P_j \in \mathcal{G}(P)}{\texttt{par}} \texttt{ if } b_{P_j} \texttt{ then } [\![P_j]\!]_{asm} \texttt{ endpar endif}$$

The control flow graph

$$\mathcal{G}(c; P) \equiv \mathcal{G}(c); P \cup \mathcal{G}(P)$$
$$\mathcal{G}(\texttt{while } F \{P\}) \equiv \mathcal{G}(P); \texttt{while } F \{P\}$$
$$\ldots$$

Compilation of structures (example)

$$[\![f_{sync}(); Q]\!]_{asm} \equiv \texttt{par} \quad b_{\texttt{sync};Q} := \text{false} \quad \| b_{\text{wait}} := \text{true} \quad \| b_Q := \text{true} \quad \texttt{endpar}$$

Compilation: from BSP-ASM to BSP-IMP

Translation for the whole machine

$$
\begin{array}{l}
b_{stop} := \mathit{false}; \\
\texttt{while}\ \neg b_{stop} \\
\quad P_{step}; \\
\quad \texttt{while}\ \neg F_{\Pi}\ \{P_{step;}\} \\
\quad f_{sync}();
\end{array}
$$

Compilation: from BSP-ASM to BSP-IMP

Translation for the whole machine

$$
\begin{array}{l}
b_{stop} := \mathit{false}; \\
\texttt{while } \neg b_{stop} \\
\quad P_{step}; \\
\quad \texttt{while } \neg F_{\Pi}\ \{P_{step};\} \\
\quad f_{sync}();
\end{array}
$$

Translation of one step

$$\texttt{par } x := y \parallel y := x \texttt{ endpar}$$

$\Rightarrow \Rightarrow \Rightarrow$

$$v_y := y; v_x := x; x := v_y; y := v_x;$$

Finally

Sequential:

$$\text{Algo}_{seq} = \text{ASM} \simeq \text{Imp}_{seq}$$

BSP:

$$\text{Algo}_{BSP} = BSP\text{–ASM} \simeq \text{Imp}_{BSP}$$

Outline

Conclusion

BSP-ASM

- Axiomatic definition of BSP algorithms and $BSP-\mathrm{ASM}$
- Using fair simulations
- $\mathrm{Algo}_{BSP} \simeq BSP-\mathrm{ASM} \simeq Imp_{BSP}$
- BSPlib is algorithmic complete

Perspectives (Ongoing/Future Work)

Conclusion

BSP-ASM

- Axiomatic definition of BSP algorithms and $BSP-$ASM
- Using fair simulations
- $\text{Algo}_{BSP} \simeq BSP-\text{ASM} \simeq Imp_{BSP}$
- BSPlib is algorithmic complete

Perspectives (Ongoing/Future Work)

Conclusion

BSP-ASM

- Axiomatic definition of BSP algorithms and $BSP-$ASM
- Using fair simulations
- $\text{Algo}_{BSP} \simeq BSP-\text{ASM} \simeq Imp_{BSP}$
- BSPlib is algorithmic complete

Perspectives (Ongoing/Future Work)

Conclusion

BSP-ASM

- Axiomatic definition of BSP algorithms and $BSP-\mathrm{ASM}$
- Using fair simulations
- $\mathrm{Algo}_{BSP} \simeq BSP-\mathrm{ASM} \simeq Imp_{BSP}$
- BSPlib is algorithmic complete

Perspectives (Ongoing/Future Work)

Conclusion

BSP-ASM

- Axiomatic definition of BSP algorithms and $BSP-$ASM
- Using fair simulations
- $\text{Algo}_{BSP} \simeq BSP-\text{ASM} \simeq Imp_{BSP}$
- BSPlib is algorithmic complete

Perspectives (Ongoing/Future Work)

- Application to other bridging models (Multi-BSP, *etc.*)
- Enumerating what is BSP (BSML) and what is not (Pregel?)

Conclusion

BSP-ASM

- Axiomatic definition of BSP algorithms and $BSP-\text{ASM}$
- Using fair simulations
- $\text{Algo}_{BSP} \simeq BSP-\text{ASM} \simeq Imp_{BSP}$
- BSPlib is algorithmic complete

Perspectives (Ongoing/Future Work)

- Application to other bridging models (Multi-BSP, *etc.*)
- Enumerating what is BSP (BSML) and what is not (Pregel?)

Conclusion

BSP-ASM

- Axiomatic definition of BSP algorithms and $BSP-$ASM
- Using fair simulations
- $\text{Algo}_{BSP} \simeq BSP-\text{ASM} \simeq Imp_{BSP}$
- BSPlib is algorithmic complete

Perspectives (Ongoing/Future Work)

- Application to other bridging models (Multi-BSP, *etc.*)
- Enumerating what is BSP (BSML) and what is not (Pregel?)

Conclusion

BSP-ASM

- Axiomatic definition of BSP algorithms and $BSP-\mathrm{ASM}$
- Using fair simulations
- $\mathrm{Algo}_{BSP} \simeq BSP-\mathrm{ASM} \simeq Imp_{BSP}$
- BSPlib is algorithmic complete

Perspectives (Ongoing/Future Work)

- Application to other bridging models (Multi-BSP, *etc.*)
- Enumerating what is BSP (BSML) and what is not (Pregel?)

Merci !