Verification of imperative BSP programs: application to cost and model-checking

Frédéric Gava and Jean Fortin

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

Do you trust you programs ? Is there a bug ?

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 (verification a posteriori/deductive)
    - partial correcteness
    - other properties
    - more automatic than Coq/Isabelle ?
    - less difficult than Coq/Isabelle ?
  - 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 (verification a posteriori/deductive)
    - partial correctness
    - other properties
    - more automatic than Coq/Isabelle ?
    - less difficult than Coq/Isabelle ?
  - 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 (verification a posteriori/deductive)
    - partial correcteness
    - other properties
    - more automatic than Coq/Isabelle ?
    - less difficult than Coq/Isabelle ?
  - 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 (verification a posteriori/deductive)
    - partial correcteness
    - other properties
    - more automatic than Coq/Isabelle ?
    - less difficult than Coq/Isabelle ?
  - 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 (verification a posteriori/deductive)
    - partial correcteness
    - other properties
    - more automatic than Coq/Isabelle ?
    - less difficult than Coq/Isabelle ?
  - Generation of proof obligations

BSPlib/PUB

Library for the BSP model:

- C Language
- Send/Receive routines
- DRMA routines

BSPlib/PUB

Library for the BSP model:

- C Language
- Send/Receive routines
- DRMA routines

BSPlib/PUB

Library for the BSP model:

- C Language
- Send/Receive routines
- DRMA routines

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 (deductive)
- Annotated programs (pre- post conditions)
- Several back-end provers (Coq, Alt-ergo, Simplify, Z3 . . .)
- need axiomatisation for set/list etc.
- 'invariant' and 'variant' for each loop
- need sometime 'ghost codes'
- Provers can generate certificates (Isabelle/Coq)

The WHY Language

WHY: an intermediate language

- For program verification (deductive)
- Annotated programs (pre- post conditions)
- Several back-end provers (Coq, Alt-ergo, Simplify, Z3 . . .)
- need axiomatisation for set/list etc.
- 'invariant' and 'variant' for each loop
- need sometime 'ghost codes'
- Provers can generate certificates (Isabelle/Coq)

The WHY Language

WHY: an intermediate language

- For program verification (deductive)
- Annotated programs (pre- post conditions)
- Several back-end provers (Coq, Alt-ergo, Simplify, Z3 . . .)
- need axiomatisation for set/list etc.
- 'invariant' and 'variant' for each loop
- need sometime 'ghost codes'
- Provers can generate certificates (Isabelle/Coq)

The WHY Language

WHY: an intermediate language

- For program verification (deductive)
- Annotated programs (pre- post conditions)
- Several back-end provers (Coq, Alt-ergo, Simplify, Z3 . . .)
- need axiomatisation for set/list etc.
- 'invariant' and 'variant' for each loop
- need sometime 'ghost codes'
- Provers can generate certificates (Isabelle/Coq)

The WHY Language

WHY: an intermediate language

- For program verification (deductive)
- Annotated programs (pre- post conditions)
- Several back-end provers (Coq, Alt-ergo, Simplify, Z3 . . .)
- need axiomatisation for set/list etc.
- 'invariant' and 'variant' for each loop
- need sometime 'ghost codes'
- Provers can generate certificates (Isabelle/Coq)

The WHY Language

WHY: an intermediate language

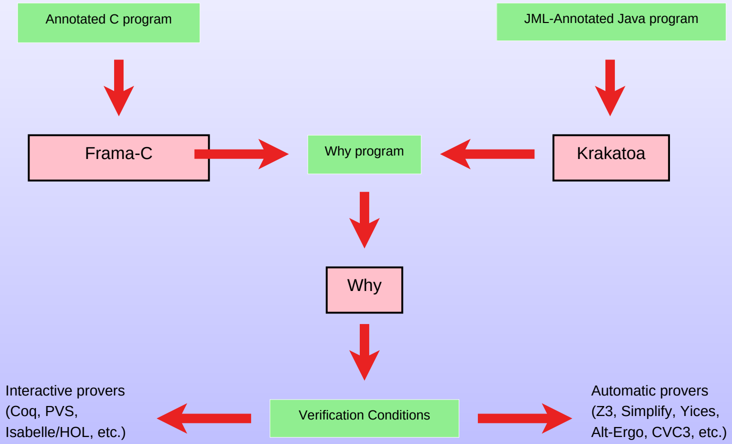
- For program verification (deductive)
- Annotated programs (pre- post conditions)
- Several back-end provers (Coq, Alt-ergo, Simplify, Z3 . . .)
- need axiomatisation for set/list etc.
- 'invariant' and 'variant' for each loop
- need sometime 'ghost codes'
- Provers can generate certificates (Isabelle/Coq)

The WHY Language

WHY: an intermediate language

- For program verification (deductive)
- Annotated programs (pre- post conditions)
- Several back-end provers (Coq, Alt-ergo, Simplify, Z3 . . .)
- need axiomatisation for set/list etc.
- ’invariant’ and ’variant’ for each loop
- need sometime ’ghost codes’
- Provers can generate certificates (Isabelle/Coq)

The WHY tools



Language definition

- BSP-WHY is extended from WHY
- Additional instructions for parallel operations
- Additional notations in assertions about parallelism
- Automatic transformation to Why code (sequentialisation)

Language definition

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

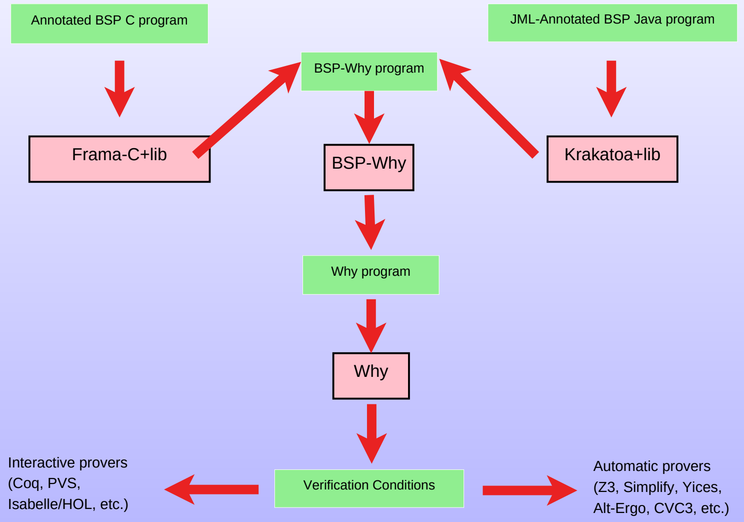
now a 'Parameter' with a 'sync' side-effect can be used instead of a **sync** (MPI collective operations)

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
- $t =<< f(pid) >>$ is a syntaxic sugar to $\forall i.\, proc(i) \rightarrow t[i] = f(i)$

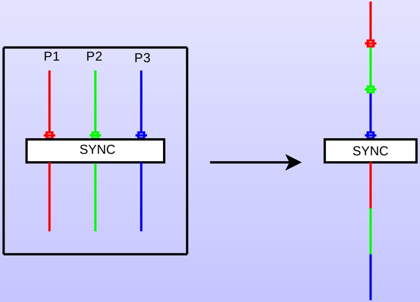
Trying to prove its correctness

BSP-WHY, an extension of WHY for BSP algorithms:



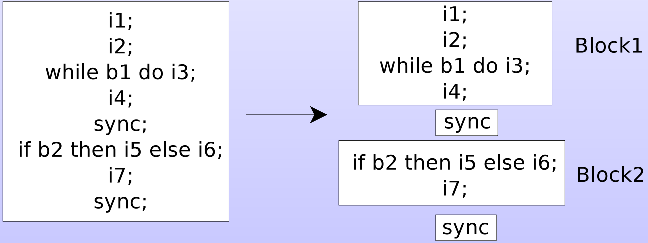
General idea of the transformation BSP-WHY ⇒ WHY

Simulation of the parallel execution by a sequential execution



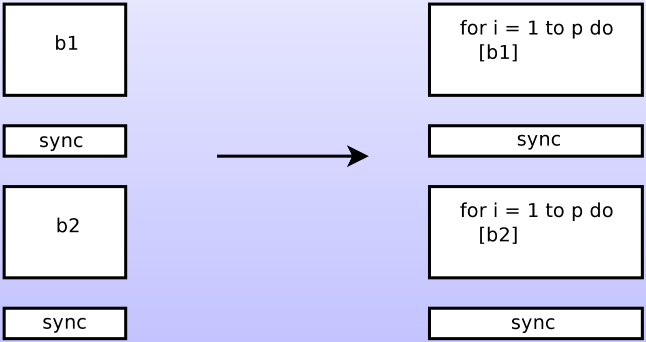
Decomposition into blocks (1/3)

We extract the biggest blocks of code without synchronization:



Decomposition into blocks (2/3)

Each block is transformed into a *for* loop:



Decomposition into blocks (3/3)

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 (3/3)

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 $\rightarrow$ 1 processor : need to simulate p memories in one.

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

Transformation of variables

BSP-WHY 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

Remote Memory Access: put/get operations (1/2)

- 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

Remote Memory Access: put/get operations (1/2)

- 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

Remote Memory Access: put/get operations (1/2)

- 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

Remote Memory Access: put/get operations (1/2)

- 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

Remote Memory Access: put/get operations (2/2)

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

Remote Memory Access: put/get operations (2/2)

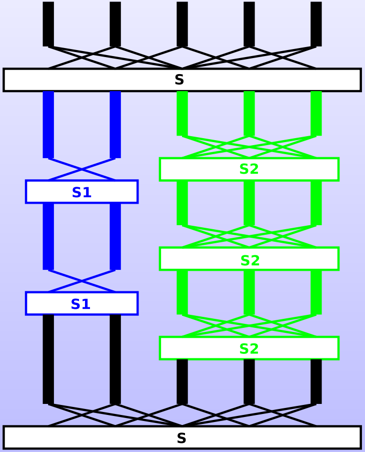
The association table is needed :

```
Proc 1    Proc 2

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

$P1$	$P2$
x	y
y	x

Subgroup synchronization



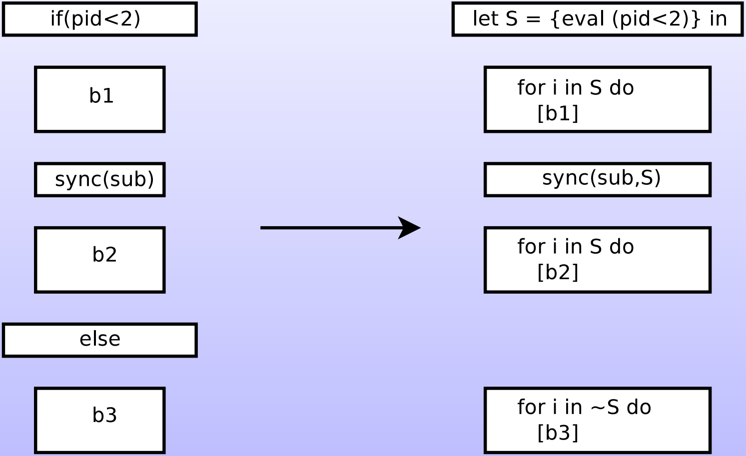
S = {0,1,2,3,4}
S1 = {0,1}
S2 = {2,3,4}

Subgroup synchronization : example in C/PUB

```
t_bsp subbsp;
int part[2];
part[0] = 2;
part[1] = bsp_nprocs(bsp);
bsp_partition (bsp, &subbsp, 2, part);
if(bsp_pid()<2) {
  ...
  bsp_sync(&subbsp);
  ...
} else {
  ...
}
bsp_done (&subbsp);
```

same kind of operation in MPI (even in collective operations)

Subgroup synchronization : transformation



Subgroup synchronization : safety

To avoid deadlocks, we check that all processors of a subgroup will synchronize at the same time :

$$assert(\forall i \in S, sub[i] \subset S)$$

```
bsp_sync(sub,S)
```

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

- 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

- 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.)

Now using BSP-WHY !

Cost

- Worst-case analysis ⇒ adding counting variables (ticks) for some operations
  - for each operation ⇒ adding one tick to the counter (side effect=monad in Coq)
  - bigger invariants
- $O(n)$ ⇒ more difficult
- more difficult than only correctness (even for sequential computations)
- less papers on machine-checked proofs but many 'Worst-case static analysis' papers

Cost

- Worst-case analysis ⇒ adding counting variables (ticks) for some operations
  - for each operation ⇒ adding one tick to the counter (side effect=monad in Coq)
  - bigger invariants
- $O(n)$ ⇒ more difficult
- more difficult than only correctness (even for sequential computations)
- less papers on machine-checked proofs but many 'Worst-case static analysis' papers

Cost

- Worst-case analysis ⇒ adding counting variables (ticks) for some operations
  - for each operation ⇒ adding one tick to the counter (side effect=monad in Coq)
  - bigger invariants
- $O(n)$ ⇒ more difficult
- more difficult than only correctness (even for sequential computations)
- less papers on machine-checked proofs but many 'Worst-case static analysis' papers

Cost

- Worst-case analysis ⇒ adding counting variables (ticks) for some operations
  - for each operation ⇒ adding one tick to the counter (side effect=monad in Coq)
  - bigger invariants
- $O(n)$ ⇒ more difficult
- more difficult than only correctness (even for sequential computations)
- less papers on machine-checked proofs but many 'Worst-case static analysis' papers

Cost

- Worst-case analysis ⇒ adding counting variables (ticks) for some operations
  - for each operation ⇒ adding one tick to the counter (side effect=monad in Coq)
  - bigger invariants
- $O(n)$ ⇒ more difficult
- more difficult than only correctness (even for sequential computations)
- less papers on machine-checked proofs but many 'Worst-case static analysis' papers

Cost

- Worst-case analysis ⇒ adding counting variables (ticks) for some operations
  - for each operation ⇒ adding one tick to the counter (side effect=monad in Coq)
  - bigger invariants
- $O(n)$ ⇒ more difficult
- more difficult than only correctness (even for sequential computations)
- less papers on machine-checked proofs but many 'Worst-case static analysis' papers

Now using BSP-WHY !

State-space construction (model-checking)

- initial state s_0
- successors given by $succ(s)$
- transition $s \to s'$ whenever $s' \in succ(s)$
- inductive (iteration) computation of the state space
  - as a graph
  - as a set of reachable states
- we here only present sets for simplicity

State-space construction (model-checking)

- initial state s_0
- successors given by $succ(s)$
- transition $s \rightarrow s'$ whenever $s' \in succ(s)$
- inductive (iteration) computation of the state space
  - as a graph
  - as a set of reachable states
- we here only present sets for simplicity

State-space construction (model-checking)

- initial state s_0
- successors given by $succ(s)$
- transition $s \rightarrow s'$ whenever $s' \in succ(s)$
- inductive (iteration) computation of the state space
  - as a graph
  - as a set of reachable states
- we here only present sets for simplicity

State-space construction (model-checking)

- initial state s_0
- successors given by $succ(s)$
- transition $s \rightarrow s'$ whenever $s' \in succ(s)$
- inductive (iteration) computation of the state space
  - as a graph
  - as a set of reachable states
- we here only present sets for simplicity

State-space construction (model-checking)

- initial state s_0
- successors given by $succ(s)$
- transition $s \to s'$ whenever $s' \in succ(s)$
- inductive (iteration) computation of the state space
  - as a graph
  - as a set of reachable states
- we here only present sets for simplicity

State-space construction (model-checking)

- initial state s_0
- successors given by $succ(s)$
- transition $s \rightarrow s'$ whenever $s' \in succ(s)$
- inductive (iteration) computation of the state space
  - as a graph
  - as a set of reachable states
- we here only present sets for simplicity

State-space construction (model-checking)

- initial state s_0
- successors given by $succ(s)$
- transition $s \to s'$ whenever $s' \in succ(s)$
- inductive (iteration) computation of the state space
  - as a graph
  - as a set of reachable states
- we here only present sets for simplicity

Simple sequential algorithm in Python

```
def normal():
  knonw={}
  todo={s0}
  while todo:
    s=todo.pop()
    known.add(s)
    todo.update(succ(s)−known)
  return known
```

Sequential dfs algorithm in Python

```
def dfs(s):
  known.add(s)
  for new_s in succ(s)−known:
    if new_s not in known:
      dfs(new_s)

def main_dfs ()
  known={}
  dfs(s0)
  return known
```

Sequential breadth-first algorithm in Python

```
def breadth_first() =
  known={}
  current={s0}
  next={}
  while current:
    for s in current:
      known.add(s)
      next.update(succ(s)−known−current)
    current=next.copy();
    next={}
  return known
```

Now using WHY !

Naive BSP algorithm

- partition function *cpu* to place states onto processors
  - hash the state (modulo number of processors)
  - most used approach
- each processor i computes $succ(s)$ iff $cpu(s) = i$
- other states are sent to their owners
- stop when no processor has computed new states

BSP algorithm (main loop) using BSP-Python

```
def main_par_state_space ():
  knonw={}
  pastsend={}
  total =1
  if  cpu(s0)=my_pid:
    todo={s0}
  else:
    todo={}
  while total>0:
    tosend=local_successor(known,todo,pastsend)
    exchange(total,todo,known,tosend,pastsend)
  return known
```

local computations

```
def local_successors (known, todo, pastsend):
  while todo:
    s=todo.pop()
    known.add(s)
    for new_s in succ(s)−known−pastsend:
      tgt=cpu(new_s)
      if tgt==my_pid:
        todo.add(new_s)
      else:
        tosend[tgt].add(new_s)
  return tosend
```

Exchange of data and new todo/total/pastsend

```
def exchange (total, todo, known, tosend, pastsend) :
  total , received=BSP_EXCHANGE(tosend)
  todo=received−known
  for i  in xrange(0,nprocs)
    pastsend.update(tosend[i])
```

local computations (only pre- and post-conditions)

```
local_successors: known: state set ref → todo:state set ref → pastsend: state set ref →
{ (known ⊆ StSpace) and (todo ⊆ StSpace) and (pastsend ⊆ StSpace) and (known ∩ todo)=∅
  and (∀ s:state. s ∈(known ∪ todo) → cpu(s)=my_pid) and (∀ s:state. s ∈past_send → cpu(s)≠ my_pid)
}
  state set fparray writes known, todo
{ (todo=∅) and (known ⊆ StSpace) and (∀ s:state. s ∈known → cpu(s)=my_pid)
    and (⋃(result) ⊆ StSpace) and ((result ∩ pastsend)=∅)
    and (∀ i:int. isproc(i) → ∀s:state. s ∈result<i> → cpu(s)≠ my_pid)
    and ((known@ ∪ todo@) ⊆ known)
    and (∀ s:state. s ∈known → succ(s) ⊆ (known ∪ ⋃(result) ∪ pastsend))
    and (todo@=∅ → ⋃(result)=∅)
}
```

Main BSP loop

```
while total>0 do
{
invariant ⋃(<known>) ∪ ⋃(<todo>) ⊆ StSpace
      and (⋃(<known>) ∩ ⋃(<todo>))=∅
      and GoodPar(<known>) and GoodPart(<todo>)
      and (∀ i,j:int. isproc(i) → isproc(j) → total<i> = total<j>)
      and total<0> ≥ |⋃(<todo>)|
      and s0 ∈(⋃(<known>) ∪ ⋃(<todo>))
      and (∀ e:state. e ∈⋃(<known>) → succ(e) ⊆ (⋃(<known>) ∪ ⋃(<todo>)))
      and ⋃(<pastsend>) ⊆ StSpace
      and (∀ i:int. isproc(i) → ∀e:state. e ∈pastsend<i> → cpu(e)≠ i)
      and ⋃(<pastsend>) ⊆ (⋃(<known>) ∪ ⋃(<todo>))
variant pair(paccess(total,0),| S \ ⋃(known) |) for lexico_order
}
 let tosend=(local_successors known todo pastsend) in
  exchange todo total !known !tosend
done;
!known
{StSpace=⋃(<result>) and GoodPart(<result>)}
```

Now using BSP-WHY !

Conclusion

- 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
- Examples: cost or BSP algorithms for state space computation

Conclusion

- 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
- Examples: cost or BSP algorithms for state space computation

Conclusion

- 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
- Examples: cost or BSP algorithms for state space computation

Conclusion

- 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
- Examples: cost or BSP algorithms for state space computation

Perspectives (ongoing work)

- Use BSP-WHY for our own BSP algoriths for checking security protocols
- Semantics and proof of the transformation of BSP-WHY using Coq
- Verified BSP implementation of data-parallel skeletons
- Proof of a subset synchronisation example
- Use this work to prove MPI programs with only global operations

Perspectives (ongoing work)

- Use BSP-WHY for our own BSP algoriths for checking security protocols
- Semantics and proof of the transformation of BSP-WHY using Coq
- Verified BSP implementation of data-parallel skeletons
- Proof of a subset synchronisation example
- Use this work to prove MPI programs with only global operations

Perspectives (ongoing work)

- Use BSP-WHY for our own BSP algoriths for checking security protocols
- Semantics and proof of the transformation of BSP-WHY using Coq
- Verified BSP implementation of data-parallel skeletons
- Proof of a subset synchronisation example
- Use this work to prove MPI programs with only global operations

Perspectives (ongoing work)

- Use BSP-WHY for our own BSP algoriths for checking security protocols
- Semantics and proof of the transformation of BSP-WHY using Coq
- Verified BSP implementation of data-parallel skeletons
- Proof of a subset synchronisation example
- Use this work to prove MPI programs with only global operations

Perspectives (ongoing work)

- Use BSP-WHY for our own BSP algoriths for checking security protocols
- Semantics and proof of the transformation of BSP-WHY using Coq
- Verified BSP implementation of data-parallel skeletons
- Proof of a subset synchronisation example
- Use this work to prove MPI programs with only global operations

Perspectives (future work)

- The aim is to generate BSP-WHY code from a BSP/MPI-C program
- Use of Frama-C with the Jessie plugin
- a true tool for costing analaysis
- LTL/CTL* machine-checked model checking algorithms
- adding tactics and theories for helping provers ($\sum_{i=0}^{p}$)

Perspectives (future work)

- The aim is to generate BSP-WHY code from a BSP/MPI-C program
- Use of Frama-C with the Jessie plugin
- a true tool for costing analaysis
- LTL/CTL* machine-checked model checking algorithms
- adding tactics and theories for helping provers ($\sum_{i}^{p} a_i$)

Perspectives (future work)

- The aim is to generate BSP-WHY code from a BSP/MPI-C program
- Use of Frama-C with the Jessie plugin
- a true tool for costing analaysis
- LTL/CTL* machine-checked model checking algorithms
- adding tactics and theories for helping provers ($\sum_{i=0}^{p}$)

Perspectives (future work)

- The aim is to generate BSP-WHY code from a BSP/MPI-C program
- Use of Frama-C with the Jessie plugin
- a true tool for costing analaysis
- LTL/CTL* machine-checked model checking algorithms
- adding tactics and theories for helping provers ($\sum_{i=0}^{p}$)

Perspectives (future work)

- The aim is to generate BSP-WHY code from a BSP/MPI-C program
- Use of Frama-C with the Jessie plugin
- a true tool for costing analaysis
- LTL/CTL* machine-checked model checking algorithms
- adding tactics and theories for helping provers ($\sum_{i=0}^{p}$)

Merci !