

On Probabilistic Parallel Programs with Process Creation and Synchronisation

Stefan Kiefer¹ Dominik Wojtczak^{1,2}

¹University of Oxford, UK

²University of Liverpool, UK

TACAS, Saarbrücken
30 March 2011

Split-Join Systems: A Model for Programs with Process Spawning

An example run:

The rules of a SJS:

X

split: $X \hookrightarrow \langle XX \rangle$

intern: $X \hookrightarrow q$

intern: $X \hookrightarrow r$

join: $\langle qr \rangle \hookrightarrow X$

Split-Join Systems: A Model for Programs with Process Spawning

The rules of a SJS:

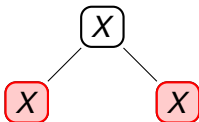
split: $X \hookrightarrow \langle XX \rangle$

intern: $X \hookrightarrow q$

intern: $X \hookrightarrow r$

join: $\langle qr \rangle \hookrightarrow X$

An example run:



Split-Join Systems: A Model for Programs with Process Spawning

The rules of a SJS:

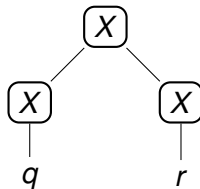
split: $X \hookrightarrow \langle XX \rangle$

intern: $X \hookrightarrow q$

intern: $X \hookrightarrow r$

join: $\langle qr \rangle \hookrightarrow X$

An example run:



Split-Join Systems: A Model for Programs with Process Spawning

The rules of a SJS:

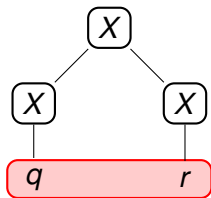
split: $X \hookrightarrow \langle XX \rangle$

intern: $X \hookrightarrow q$

intern: $X \hookrightarrow r$

join: $\langle qr \rangle \hookrightarrow X$

An example run:



Split-Join Systems: A Model for Programs with Process Spawning

The rules of a SJS:

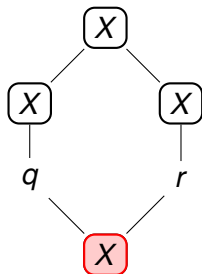
split: $X \hookrightarrow \langle XX \rangle$

intern: $X \hookrightarrow q$

intern: $X \hookrightarrow r$

join: $\langle qr \rangle \hookrightarrow X$

An example run:



Split-Join Systems: A Model for Programs with Process Spawning

The rules of a SJS:

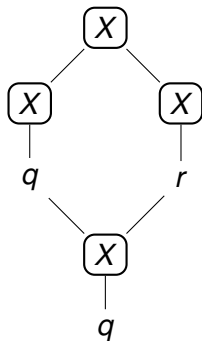
split: $X \hookrightarrow \langle XX \rangle$

intern: $X \hookrightarrow q$

intern: $X \hookrightarrow r$

join: $\langle qr \rangle \hookrightarrow X$

An example run:



Split-Join Systems: A Model for Programs with Process Spawning

The rules of a SJS:

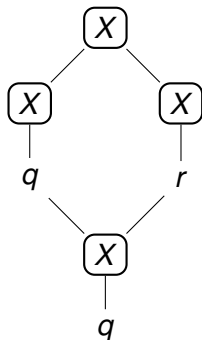
split: $X \hookrightarrow \langle XX \rangle$

intern: $X \hookrightarrow q$

intern: $X \hookrightarrow r$

join: $\langle qr \rangle \hookrightarrow X$

An example run:



Synchronisation states (here q, r) can be used to return values.

Split-Join Systems: A Model for Programs with Process Spawning

The rules of a SJS:

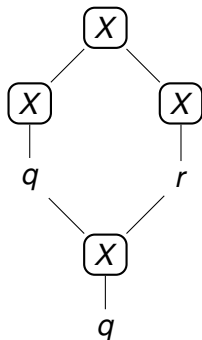
split: $X \hookrightarrow \langle XX \rangle$

intern: $X \hookrightarrow q$

intern: $X \hookrightarrow r$

join: $\langle qr \rangle \hookrightarrow X$

An example run:



Synchronisation states (here q, r) can be used to return values.

Associated to a run: **Time** $T = 4$, **Work** $W = 5$, **Space** $S = 2$

Split-Join Systems: A Model for Programs with Process Spawning

The rules of a pSJS:

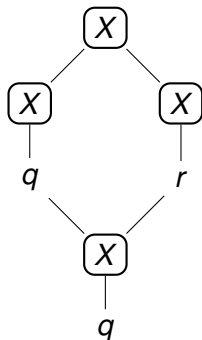
$$\text{split: } X \xrightarrow{0.5} \langle XX \rangle$$

$$\text{intern: } X \xrightarrow{0.3} q$$

$$\text{intern: } X \xrightarrow{0.2} r$$

$$\text{join: } \langle qr \rangle \xrightarrow{1.0} X$$

An example run:



Synchronisation states (here q, r) can be used to return values.

Associated to a run: **Time** $T = 4$, **Work** $W = 5$, **Space** $S = 2$

Split-Join Systems: A Model for Programs with Process Spawning

The rules of a pSJS:

$$\text{split: } X \xrightarrow{0.5} \langle XX \rangle$$

$$\text{intern: } X \xrightarrow{0.3} q$$

$$\text{intern: } X \xrightarrow{0.2} r$$

$$\text{join: } \langle qr \rangle \xrightarrow{1.0} X$$

Another example run:



Split-Join Systems: A Model for Programs with Process Spawning

The rules of a pSJS:

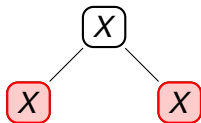
$$\text{split: } X \xrightarrow{0.5} \langle XX \rangle$$

$$\text{intern: } X \xrightarrow{0.3} q$$

$$\text{intern: } X \xrightarrow{0.2} r$$

$$\text{join: } \langle qr \rangle \xrightarrow{1.0} X$$

Another example run:



Split-Join Systems: A Model for Programs with Process Spawning

The rules of a pSJS:

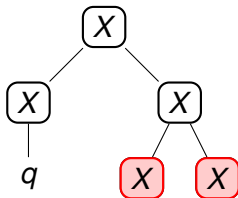
$$\text{split: } X \xrightarrow{0.5} \langle XX \rangle$$

$$\text{intern: } X \xrightarrow{0.3} q$$

$$\text{intern: } X \xrightarrow{0.2} r$$

$$\text{join: } \langle qr \rangle \xrightarrow{1.0} X$$

Another example run:



Split-Join Systems: A Model for Programs with Process Spawning

The rules of a pSJS:

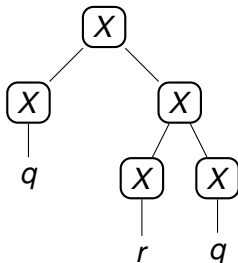
$$\text{split: } X \xrightarrow{0.5} \langle XX \rangle$$

$$\text{intern: } X \xrightarrow{0.3} q$$

$$\text{intern: } X \xrightarrow{0.2} r$$

$$\text{join: } \langle qr \rangle \xrightarrow{1.0} X$$

Another example run:



Split-Join Systems: A Model for Programs with Process Spawning

The rules of a pSJS:

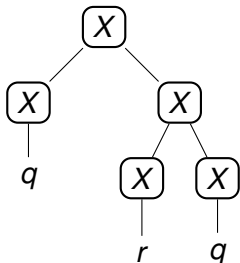
$$\text{split: } X \xrightarrow{0.5} \langle XX \rangle$$

$$\text{intern: } X \xrightarrow{0.3} q$$

$$\text{intern: } X \xrightarrow{0.2} r$$

$$\text{join: } \langle qr \rangle \xrightarrow{1.0} X$$

Another example run:



Time $T = 3$, Work $W = 5$, Space $S = 3$

Why the Sibling Constraint?

The **sibling constraint** prevents arbitrary synchronisation.

- Arbitrary Synchronisation would lead to Petri-Nets.
→ more difficult to analyse
- Sibling Synchronisation is enough for modeling purposes.

Relationship to pPDSs

pSJSs subsume (probabilistic) pushdown systems (pPDSs).

Pushdown rules:

push: $qX \hookrightarrow rYZ$

intern: $rY \hookrightarrow sW$

pop: $sW \hookrightarrow t$

pop: $tZ \hookrightarrow u$

Run $qX \Rightarrow rYZ \Rightarrow sWZ \Rightarrow tZ \Rightarrow u$

Relationship to pPDSs

pSJSs subsume (probabilistic) pushdown systems (pPDSs).

View this run as:

Pushdown rules:

push: $qX \hookrightarrow rYZ$

intern: $rY \hookrightarrow sW$

pop: $sW \hookrightarrow t$

pop: $tZ \hookrightarrow u$

qX

Run $qX \Rightarrow rYZ \Rightarrow sWZ \Rightarrow tZ \Rightarrow u$

Relationship to pPDSs

pSJSs subsume (probabilistic) pushdown systems (pPDSs).

Pushdown rules:

push: $qX \hookrightarrow rYZ$

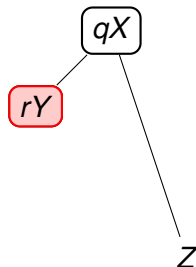
intern: $rY \hookrightarrow sW$

pop: $sW \hookrightarrow t$

pop: $tZ \hookrightarrow u$

Run $qX \Rightarrow rYZ \Rightarrow sWZ \Rightarrow tZ \Rightarrow u$

View this run as:



Relationship to pPDSs

pSJSs subsume (probabilistic) pushdown systems (pPDSs).

Pushdown rules:

push: $qX \hookrightarrow rYZ$

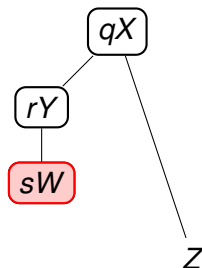
intern: $rY \hookrightarrow sW$

pop: $sW \hookrightarrow t$

pop: $tZ \hookrightarrow u$

Run $qX \Rightarrow rYZ \Rightarrow sWZ \Rightarrow tZ \Rightarrow u$

View this run as:



Relationship to pPDSs

pSJSs subsume (probabilistic) pushdown systems (pPDSs).

Pushdown rules:

push: $qX \hookrightarrow rYZ$

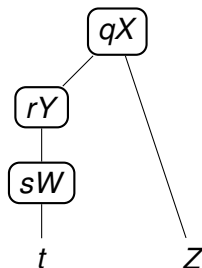
intern: $rY \hookrightarrow sW$

pop: $sW \hookrightarrow t$

pop: $tZ \hookrightarrow u$

Run $qX \Rightarrow rYZ \Rightarrow sWZ \Rightarrow tZ \Rightarrow u$

View this run as:



Relationship to pPDSs

pSJSs subsume (probabilistic) pushdown systems (pPDSs).

Pushdown rules:

push: $qX \hookrightarrow rYZ$

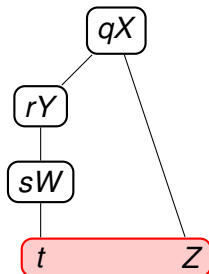
intern: $rY \hookrightarrow sW$

pop: $sW \hookrightarrow t$

pop: $tZ \hookrightarrow u$

Run $qX \Rightarrow rYZ \Rightarrow sWZ \Rightarrow tZ \Rightarrow u$

View this run as:



Relationship to pPDSs

pSJSs subsume (probabilistic) pushdown systems (pPDSs).

Pushdown rules:

push: $qX \hookrightarrow rYZ$

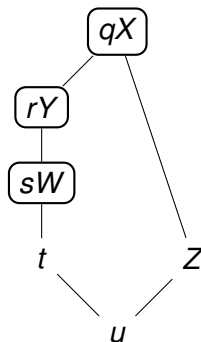
intern: $rY \hookrightarrow sW$

pop: $sW \hookrightarrow t$

pop: $tZ \hookrightarrow u$

Run $qX \Rightarrow rYZ \Rightarrow sWZ \Rightarrow tZ \Rightarrow u$

View this run as:



Relationship to pPDSs

pSJSs subsume (probabilistic) pushdown systems (pPDSs).

Pushdown rules:

push: $qX \hookrightarrow rYZ$

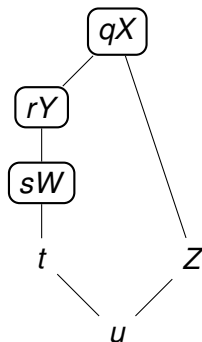
intern: $rY \hookrightarrow sW$

pop: $sW \hookrightarrow t$

pop: $tZ \hookrightarrow u$

Run $qX \Rightarrow rYZ \Rightarrow sWZ \Rightarrow tZ \Rightarrow u$

View this run as:



Time $T = 4$, **Work** $W = T = 4$, **Space** $S = 2$

Conversely, any pSJS can be **sequentialised**.

The resulting pPDS is equivalent with respect to

- returned value
- work

Conversely, any pSJS can be **sequentialised**.

The resulting pPDS is equivalent with respect to

- returned value
- work

Bottomline:

- pPDSs: **sequential** programs
- pSJSs: **parallel** programs

Relationship to Branching Processes

A **branching process** is a pSJS without join rules.

Example

$$X \xrightarrow{0.5} \langle XY \rangle$$

$$Y \xrightarrow{0.3} X$$

$$X \xrightarrow{0.5} q$$

$$Y \xrightarrow{0.7} q$$

A single synchronisation state suffices.

classical mathematical model

→ biology, physics, natural language processing, ...

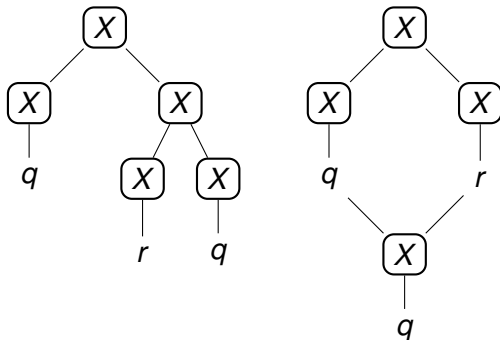
Our pSJS model **generalises** pPDSs and branching process.

- **procedures** from pPDSs
 - may be recursive
 - may return values
- parallel **spawns** from branching processes
- new: **joins** for process synchronisation and communication

Termination

terminating runs:

$$\begin{aligned} \text{split: } X &\xrightarrow{0.5} \langle XX \rangle \\ \text{intern: } X &\xrightarrow{0.3} q \\ \text{intern: } X &\xrightarrow{0.2} r \\ \text{join: } \langle qr \rangle &\xrightarrow{1.0} X \end{aligned}$$



One can transform the pSJS so that
all terminating runs terminate in a single state.

Termination Probability

$[X \downarrow q]$ = Prob. that X terminates in q

Example

$$X \xrightarrow{2/3} \langle XX \rangle \quad X \xrightarrow{1/3} q \quad \langle qq \rangle \xrightarrow{1} q$$

X can only terminate in q . We have $[X \downarrow q] = 1/2$.

Theorem

The $[X \downarrow q]$ are the solution of a system of polynomial equations.

Deciding whether $[X \downarrow q] = 0$ is in P .

Deciding whether $[X \downarrow q] < 1$ is PosSLP-hard even for pPDSs.

Probability of Finite Space

A pSJS may be useful even if it does not always terminate.
→ operating systems, network servers, system daemons, ...

Theorem

Let r be the *probability* that
a computation started in X needs only *finite space*.
Then r can be “efficiently expressed”.
Deciding whether $r = 0$ is in P.
Deciding whether $r < 1$ is PosSLP-hard even for pPDSs.

Proof: By transforming the pSJS so that
a run terminates if and only if it needs finite space.

Applied to pPDSs, the theorem improves on [EKM05].

Work and Time

Given a pSJS and a start process X , one can compute $\mathcal{P}(W = k)$ and $\mathcal{P}(T = k)$ for $k = 1, 2, \dots$ iteratively.

This allows to **approximate** $\mathbb{E}W$ and $\mathbb{E}T$.

Work and Time

Given a pSJS and a start process X , one can compute $\mathcal{P}(W = k)$ and $\mathcal{P}(T = k)$ for $k = 1, 2, \dots$ iteratively.

This allows to **approximate** $\mathbb{E}W$ and $\mathbb{E}T$.

Theorem

*$\mathbb{E}W$ and $\mathbb{E}T$ are either both finite or both infinite.
Distinguishing between those cases is in PSPACE
and PosSLP-hard even for pPDSs.*

Work and Time

Given a pSJS and a start process X , one can compute $\mathcal{P}(W = k)$ and $\mathcal{P}(T = k)$ for $k = 1, 2, \dots$ iteratively.

This allows to **approximate** $\mathbb{E}W$ and $\mathbb{E}T$.

Theorem

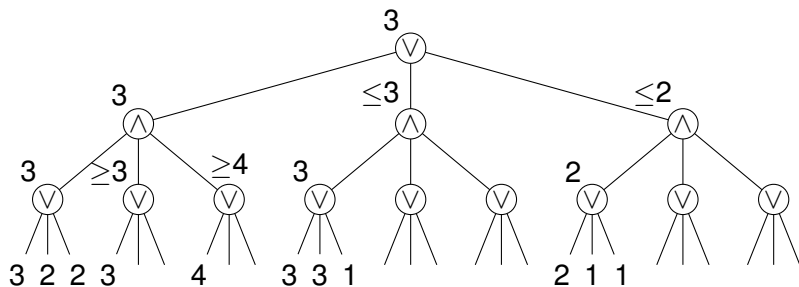
*$\mathbb{E}W$ and $\mathbb{E}T$ are either both finite or both infinite.
Distinguishing between those cases is in PSPACE
and PosSLP-hard even for pPDSs.*

Proof sketch for the upper bound:

- transform the pSJS to a **branching process** with similar distribution of W and T (uses $[X \downarrow q]$)
- set up a **matrix** A for the branching process such that $A_{X,Y}$ = expected number of spawned Y -processes when applying an X -rule
- compute the **spectral radius** of A and compare with 1

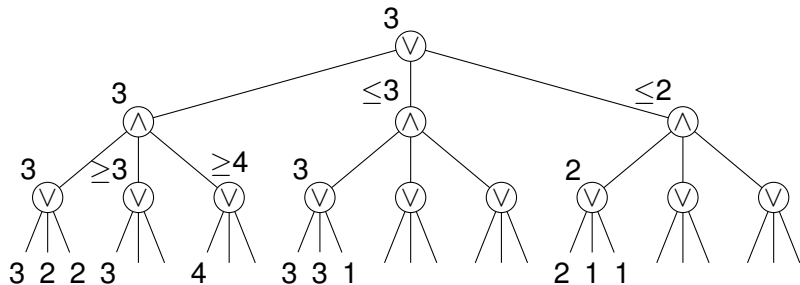
Example of a Performance Analysis: Game Tree Evaluation

We wish to analyse programs that evaluate min-max trees:



Example of a Performance Analysis: Game Tree Evaluation

We wish to analyse programs that evaluate min-max trees:



(assume 0 or 3 children per node)

A Straightforward Parallel Program

```
function parMax(node)
  if node.leaf() then return node.val()
  else parallel < val1 := parMin(node.c1),
                 val2 := parMin(node.c2),
                 val3 := parMin(node.c3) >
  return max{val1, val2, val3}
```

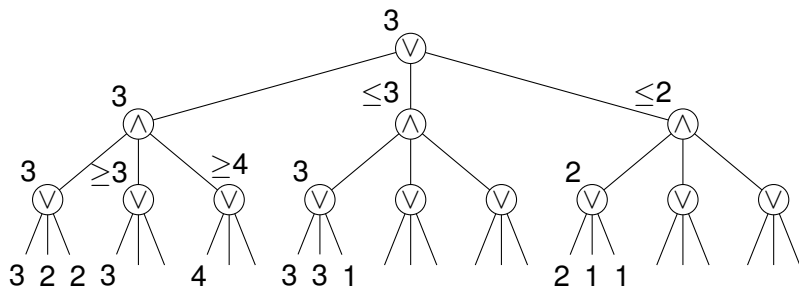
A Straightforward Parallel Program

```
function parMax(node)
  if node.leaf() then return node.val()
  else parallel < val1 := parMin(node.c1),
                 val2 := parMin(node.c2),
                 val3 := parMin(node.c3) >
    return max{val1, val2, val3}
```

→ parallel, but a lot of unnecessary work
(maybe deep in the tree)

Example of a Performance Analysis: Game Tree Evaluation

We wish to analyse programs that evaluate min-max trees:



A Smart Sequential Program

```
function seqMax(node,  $\alpha$ ,  $\beta$ )      (initially:  $\alpha = -\infty$ ,  $\beta = +\infty$ )  
  if node.leaf() then if node.val()  $\leq \alpha$  then return  $\alpha$   
    elseif node.val()  $\geq \beta$  then return  $\beta$   
    else return node.val()  
else val1 := seqMin(node.c1,  $\alpha$ ,  $\beta$ )  
  if val1 =  $\beta$  then return  $\beta$       (*cut-off after 1st child*)  
  else val2 := seqMin(node.c2, val1,  $\beta$ )  
    if val2 =  $\beta$  then return  $\beta$     (*cut-off after 2nd child*)  
    else return seqMin(node.c3, val2,  $\beta$ )
```

A Smart Sequential Program

```
function seqMax(node,  $\alpha$ ,  $\beta$ )      (initially:  $\alpha = -\infty$ ,  $\beta = +\infty$ )  
  if node.leaf() then if node.val()  $\leq \alpha$  then return  $\alpha$   
    elseif node.val()  $\geq \beta$  then return  $\beta$   
    else return node.val()  
  else val1 := seqMin(node.c1,  $\alpha$ ,  $\beta$ )  
    if val1 =  $\beta$  then return  $\beta$       (*cut-off after 1st child*)  
    else val2 := seqMin(node.c2, val1,  $\beta$ )  
      if val2 =  $\beta$  then return  $\beta$     (*cut-off after 2nd child*)  
      else return seqMin(node.c3, val2,  $\beta$ )
```

→ no unnecessary work, but sequential

A Smart Parallel Program

“Young Brothers Wait”:

Idea: cut-offs usually occur after the 1st child (“oldest brother”)

- do the oldest brother
- only if no cut-off: do the young brothers **in parallel**

```
function YBWMax(node,  $\alpha$ ,  $\beta$ )  
  if node.leaf() then if node.val()  $\leq \alpha$  then return  $\alpha$   
    elseif node.val()  $\geq \beta$  then return  $\beta$   
    else return node.val()  
  else val1 := YBWMin(node.c1,  $\alpha$ ,  $\beta$ )  
    if val1 =  $\beta$  then return  $\beta$    (*cut-off after 1st child*)  
    else parallel  $\langle$  val2 := YBWMin(node.c2, val1,  $\beta$ ),  
      val3 := YBWMin(node.c3, val1,  $\beta$ )  $\rangle$   
    return max{val2, val3}
```

These programs (par, seq, YBW) have various features:

- procedural
- recursive
- return values
- spawns and joins

These programs (par, seq, YBW) have various features:

- procedural
- recursive
- return values
- spawns and joins
- **probabilistic** if the input (the trees) are probabilistic

These programs (par, seq, YBW) have various features:

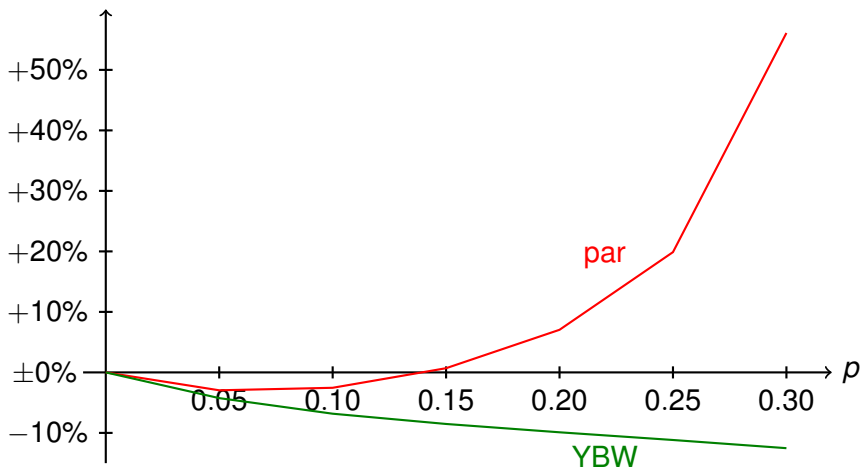
- procedural
- recursive
- return values
- spawns and joins
- **probabilistic** if the input (the trees) are probabilistic

This all fits naturally in the pSJS model.

We analyse performance under prob. assumptions on the input.

Performance Analysis

Expected runtime of **par** and **YBW** compared to seq,
as a function of ρ (ρ controls the expected tree size)



Conclusions

- **pSJS: new model** for probabilistic parallel programs with **process spawning and synchronisation**
- Basic quantitative analysis is **as expensive as for pPDSs**.
- We can **model, analyse and compare** parallel programs under probabilistic assumptions.

Thank you!