On Probabilistic Parallel Programs with Process Creation and Synchronisation

Stefan Kiefer\textsuperscript{1}  Dominik Wojtczak\textsuperscript{1,2}

\textsuperscript{1}University of Oxford, UK
\textsuperscript{2}University of Liverpool, UK

TACAS, Saarbrücken
30 March 2011
Split-Join Systems: A Model for Programs with Process Spawning

The rules of a SJS:

- **split:** $X \mapsto \langle XX \rangle$
- **intern:** $X \mapsto q$
- **intern:** $X \mapsto r$
- **join:** $\langle qr \rangle \mapsto X$

An example run:

Stefan Kiefer, Dominik Wojtczak

On Probabilistic Parallel Programs with Process Creation
Split-Join Systems: A Model for Programs with Process Spawning

The rules of a SJS:

- **split**: $X \leftarrow \langle XX \rangle$
- **intern**: $X \leftarrow q$
- **intern**: $X \leftarrow r$
- **join**: $\langle qr \rangle \leftarrow X$

An example run:

```
X
```

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

Associated to a run: \( T = 4 \), \( W = 5 \), \( S = 2 \)
The rules of a SJS:

- **split**: $X \mapsto \langle XX \rangle$
- **intern**: $X \mapsto q$
- **intern**: $X \mapsto r$
- **join**: $\langle qr \rangle \mapsto X$

An example run:

```
X
```

```
q
```

```
r
```
The rules of a SJS:

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

intern: $X \leftrightarrow q$

intern: $X \leftrightarrow r$

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

An example run:

```
X
- X
- X
```

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

Associated to a run: Time $T = 4$, Work $W = 5$, Space $S = 2$.
The rules of a SJS:

- **split**: $X \leftarrow \langle XX \rangle$
- **intern**: $X \leftarrow q$
- **intern**: $X \leftarrow r$
- **join**: $\langle qr \rangle \leftarrow X$

An example run:

```
\[X\]

\[X\] \[X\] \[X\]

q \[X\]

r
```
The rules of a SJS:

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

intern: $X \leftrightarrow q$

intern: $X \leftrightarrow r$

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

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

The rules of a SJS:

- split: $X \leftrightarrow \langle XX \rangle$
- intern: $X \leftrightarrow q$
- intern: $X \leftrightarrow r$
- join: $\langle qr \rangle \leftrightarrow X$

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: $X \xleftarrow{} \langle XX \rangle$

intern: $X \xleftarrow{} q$

intern: $X \xleftarrow{} r$

join: $\langle qr \rangle \xleftarrow{} X$

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:

- **split**: $X \xrightarrow{0.5} \langle XX \rangle$
- **intern**: $X \xrightarrow{0.3} q$
- **intern**: $X \xrightarrow{0.2} r$
- **join**: $\langle qr \rangle \xleftarrow{1.0} X$

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

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

split: $X \overset{0.5}{\rightarrow} \langle XX \rangle$

intern: $X \overset{0.3}{\rightarrow} q$

intern: $X \overset{0.2}{\rightarrow} r$

join: $\langle qr \rangle \overset{1.0}{\leftarrow} X$

Another example run:
The rules of a pSJS:

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

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

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

**join:** $\langle qr \rangle \xleftarrow{1.0} X$

Another example run:

![Diagram showing a tree structure with labeled nodes X and arrows indicating transitions between states]
The rules of a pSJS:

\[\text{split: } X \xrightarrow{0.5} \langle XX\rangle\]
\[\text{intern: } X \xrightarrow{0.3} q\]
\[\text{intern: } X \xrightarrow{0.2} r\]
\[\text{join: } \langle qr\rangle \xleftarrow{1.0} X\]

Another example run:

```
  X
 / \   /
 q  X  X
```

\[\text{Time } T = 3, \text{ Work } W = 5, \text{ Space } S = 3\]
Split-Join Systems: A Model for Programs with Process Spawning

The rules of a pSJS:

- **split:** $X \overset{0.5}{\leftrightarrow} \langle XX \rangle$
- **intern:** $X \overset{0.3}{\leftrightarrow} q$
- **intern:** $X \overset{0.2}{\leftrightarrow} r$
- **join:** $\langle qr \rangle \overset{1.0}{\leftrightarrow} X$

Another example run:

```
X
  / \  /
 X  X q r
```

Time $T = 3$, Work $W = 5$, Space $S = 3$
The rules of a pSJS:

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

intern: $X \xrightarrow{0.3} q$

intern: $X \xrightarrow{0.2} r$

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

Another example run:

Time $T = 3$, Work $W = 5$, Space $S = 3$
The **sibling constraint** prevents arbitrary synchronisation.

- Arbitrary Synchronisation would lead to Petri-Nets.  
  \[\rightarrow\text{ more difficult to analyse}\]
- Sibling Synchronisation is enough for modeling purposes.
pSJSs subsume (probabilistic) pushdown systems (pPDSs).

**Pushdown rules:**

- push: \( qX \leftrightarrow rYZ \)
- intern: \( rY \leftrightarrow sW \)
- pop: \( sW \leftrightarrow t \)
- pop: \( tZ \leftrightarrow u \)

Run \( qX \Rightarrow rYZ \Rightarrow sWZ \Rightarrow tZ \Rightarrow u \)
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:

$qX$
pSJSs subsume (probabilistic) pushdown systems (pPDSs).

**Pushdown rules:**

- **push:** $qX \leftarrow rYZ$
- **intern:** $rY \leftarrow sW$
- **pop:** $sW \leftarrow t$
- **pop:** $tZ \leftarrow u$

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

View this run as:

```
  qX
     \------\-----
     \      \   
      rY   sW
       \----/   \\
         \   /   \
          Z u
```
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:

```
\[
\begin{array}{c}
qX \\
rY \\
sW \\
Z \\
\end{array}
\]
```
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:
pSJSs subsume (probabilistic) pushdown systems (pPDSs).

Pushdown rules:

- push: $qX \mapsto rYZ$
- intern: $rY \mapsto sW$
- pop: $sW \mapsto t$
- pop: $tZ \mapsto u$

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

View this run as:

Diagram:

- $qX$ (root)
- $rY$
- $sW$
- $t$
- $Z$

Time $T = 4$, Work $W = T = 4$, Space $S = 2$
pSJSs subsume (probabilistic) pushdown systems (pPDSs).

**Pushdown rules:**

push: \( qX \mapsto rYZ \)

intern: \( rY \mapsto sW \)

pop: \( sW \mapsto t \)

pop: \( tZ \mapsto u \)

Run \( qX \Rightarrow rYZ \Rightarrow sWZ \Rightarrow tZ \Rightarrow u \)
Relationship to pPDSs

pSJSs subsume (probabilistic) pushdown systems (pPDSs).

Pushdown rules:

push: $qX \leadsto rYZ$

intern: $rY \leadsto sW$

pop: $sW \leadsto t$

pop: $tZ \leadsto u$

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

View this run as:

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

Time $T = 4$, Work $W = T = 4$, Space $S = 2$
Conversely, any pSJS can be \textit{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
A branching process is a pSJS without join rules.

Example

A single synchronisation state suffices.

classical mathematical model

→ biology, physics, natural language processing, . . .
pSJSs are general.

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

\[ [X \downarrow q] = \text{Prob. that } X \text{ terminates in } q \]

**Example**

\[
\begin{align*}
X & \xrightarrow{2/3} \langle XX \rangle \\
X & \xrightarrow{1/3} q \\
\langle qq \rangle & \xleftarrow{1} q
\end{align*}
\]


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.
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 $k = 1, 2, \ldots$ 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 $T$. 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.
Work and Time

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

This allows to approximate $E W$ and $E T$.

**Theorem**

$E W$ and $E T$ 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 $P(W = k)$ and $P(T = k)$ for $k = 1, 2, \ldots$ 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 $A_{X,Y} = \text{expected number of spawned } Y\text{-processes when applying an } X\text{-rule}$
- compute the spectral radius of $A$ and compare with 1
We wish to analyse programs that evaluate min-max trees:
We wish to analyse programs that evaluate min-max trees:

(assume 0 or 3 children per node)
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}
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)
We wish to analyse programs that evaluate min-max trees:
function seqMax(node, α, β) (initially: α = −∞, β = +∞)
    if node.leaf() then
        if node.val() ≤ α then return α
        elsif node.val() ≥ β then return β
        else return node.val()
    else
        val1 := seqMin(node.c1, α, β)
        if val1 = β then return β (*cut-off after 1st child*)
        else
            val2 := seqMin(node.c2, val1, β)
            if val2 = β then return β (*cut-off after 2nd child*)
            else return seqMin(node.c3, val2, β)
        end if
    end if
function seqMax(node, α, β)  (initially: α = −∞, β = +∞)
  if node.leaf() then
    if node.val() ≤ α then return α
    elsif node.val() ≥ β then return β
    else return node.val()
  else
    val1 := seqMin(node.c1, α, β)
    if val1 = β then return β (*cut-off after 1st child*)
    else
      val2 := seqMin(node.c2, val1, β)
      if val2 = β then return β (*cut-off after 2nd child*)
      else return seqMin(node.c3, val2, β)
  end if

→ no unnecessary work, but sequential
“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 \textbf{in parallel}

\begin{verbatim}
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 \(\beta\) \hspace{1em} (*cut-off after 1st child*)
        else parallel \{ val2 := YBWMin(node.c2, val1, \(\beta\)),
                         val3 := YBWMin(node.c3, val1, \(\beta\)) \}
        return max\{val2, val3\}
\end{verbatim}
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.
Expected runtime of \texttt{par} and \texttt{YBW} compared to seq, as a function of $p$ ($p$ 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!