ParaStack: Efficient Hang Detection for MPI Programs at Large Scale

Hongbo Li
Zizhong Chen & Rajiv Gupta

Question Solution Evaluation

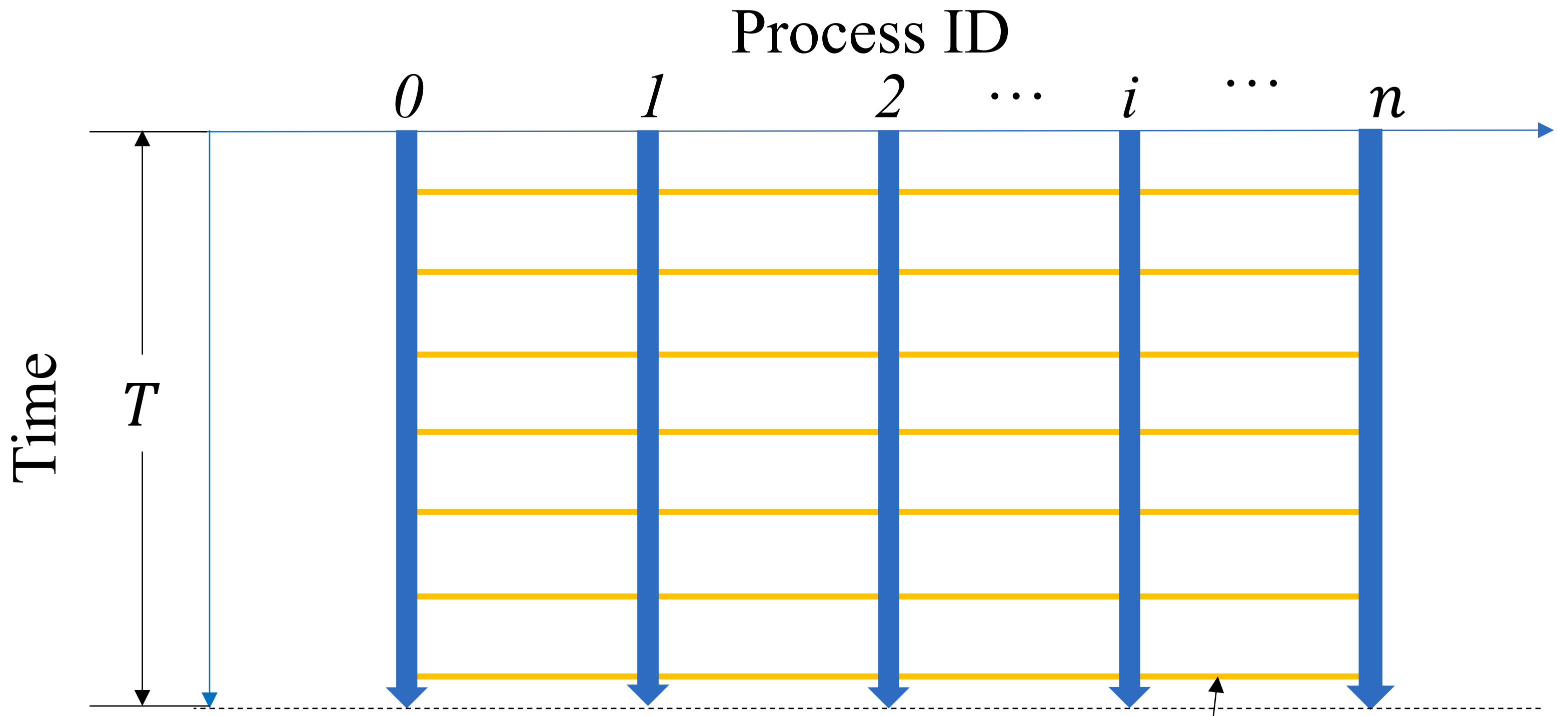
Question → Solution → Evaluation

Program Hang

Resource Wastage

Current Solution

Execution in Batch Mode

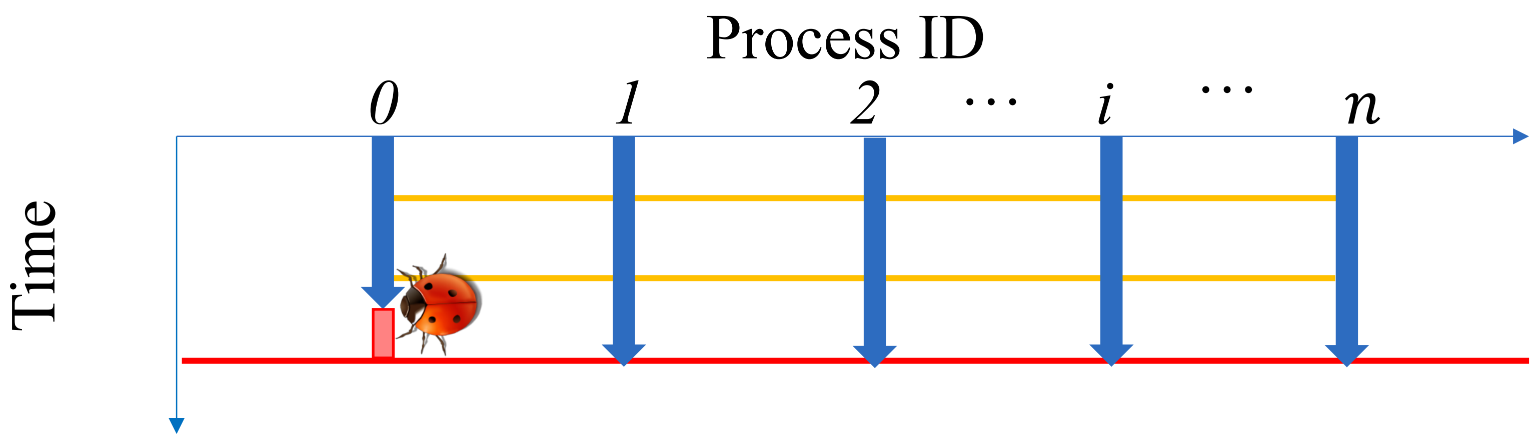


- T: occupied supercomputer time.
- Processes communicate via *message passing* (MPI).

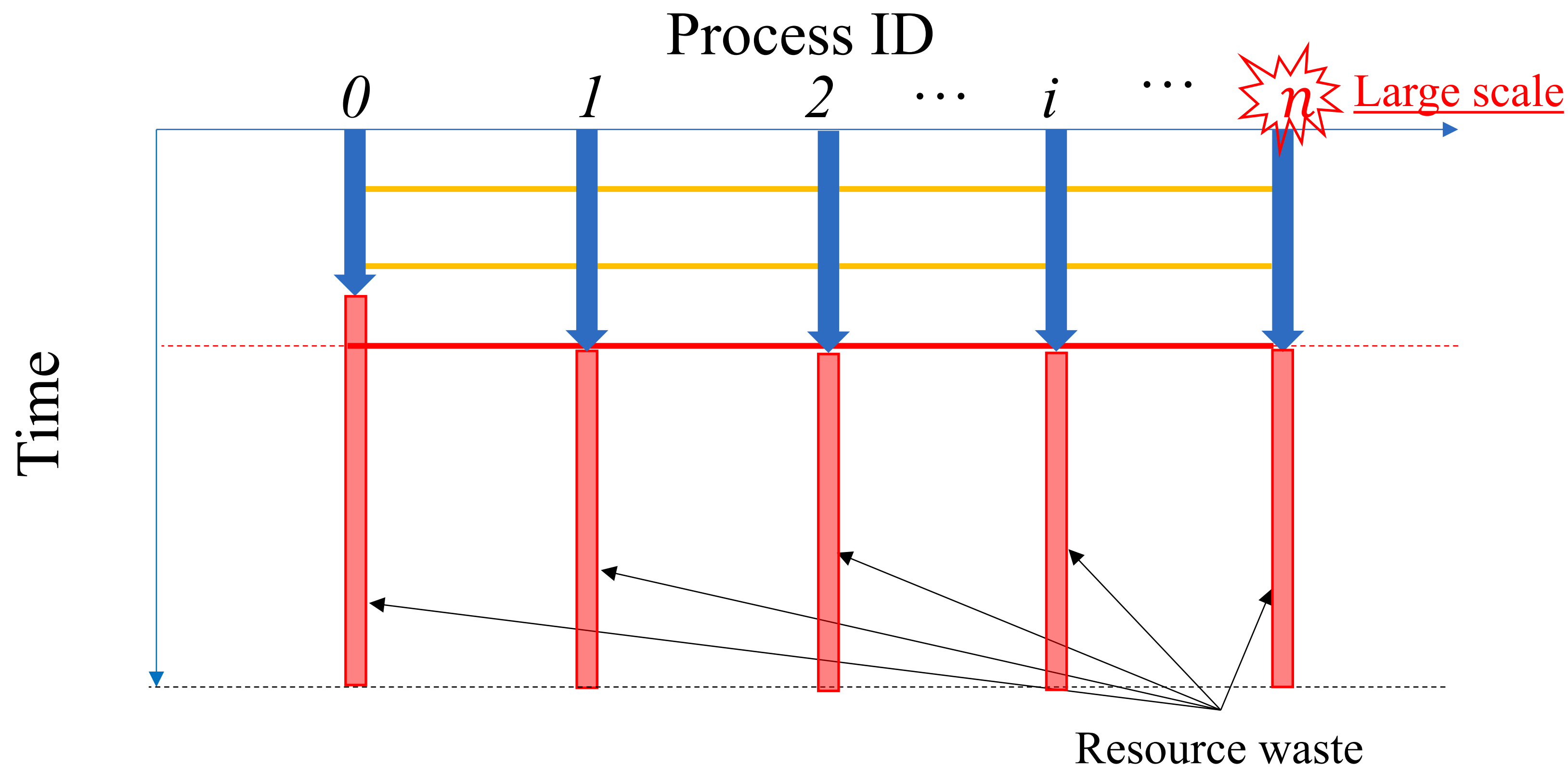
Program Hang Occurs

- **Program hang ---** a type of bug whose occurrence stalls the program's execution.

- **Root cause** can be in
  - one single process, e.g. process 0 --- Incorrect *thread-level synchronization* and *infinite loop,*
  - or all processes --- communication deadlock across all processes et.al.

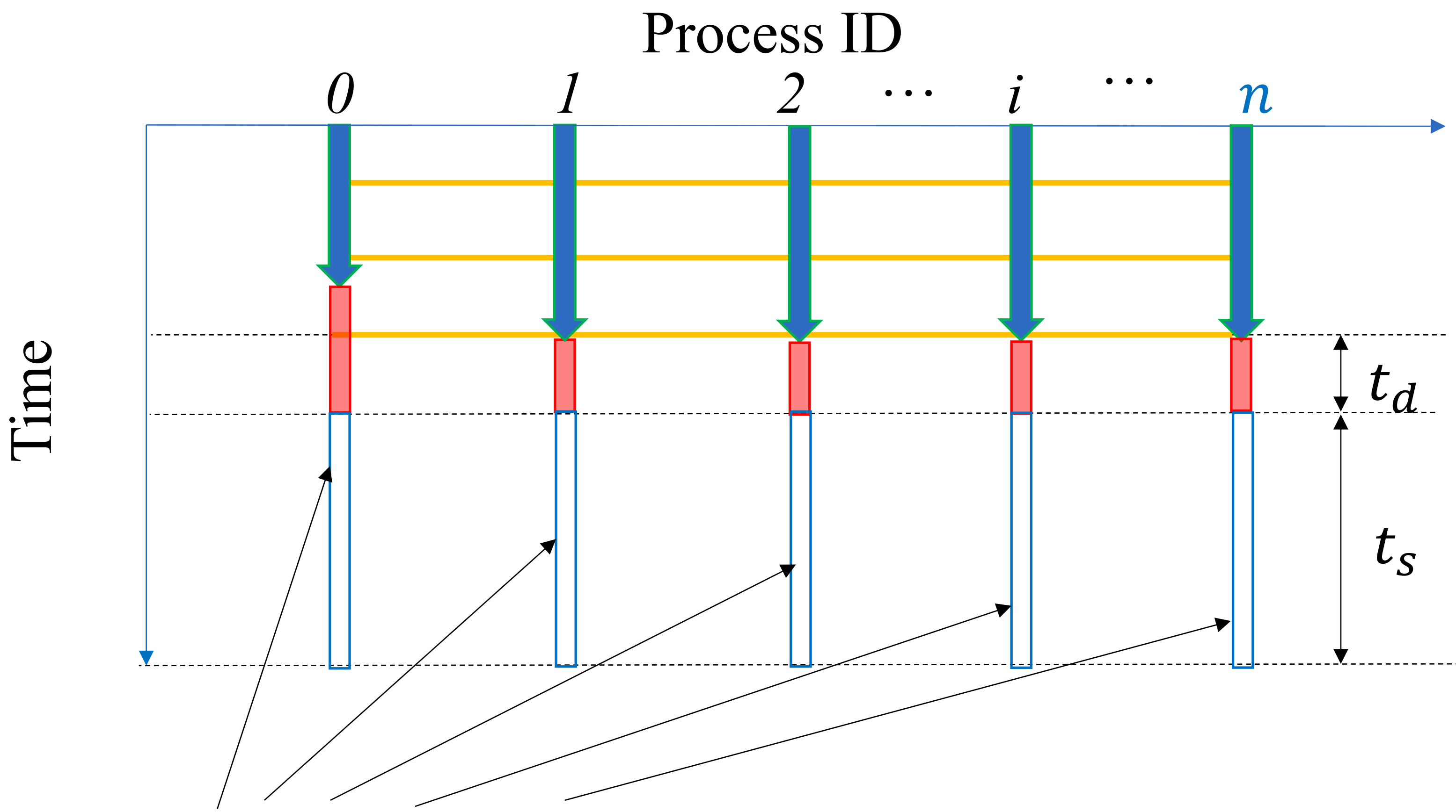


Hang Causes Resource Wastage



- **Negative ---** significant resource wastage at large scale.

Solution: Hang Detection



- *Release resources* when detecting a hang
- **Shorter detection delay (t_d) → Bigger saving (t_s)**

Traditional Detection Method 😟

- **Timeout** is a commonly used method based on various metrics, e.g., *IO-watchdog monitors how often a program writes*.

- **Setting a good timeout is hard** due to following two dilemmas:

  - Small timeout → Large Savings

    Too Small timeout → False Alarms

  - Large timeout → Avoid False Positives

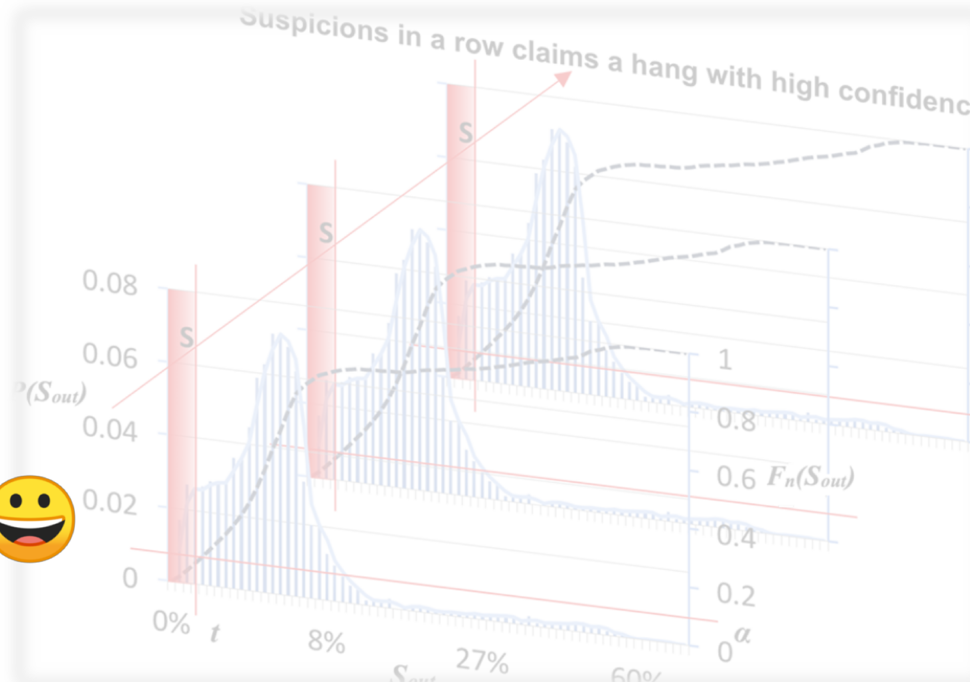
    Too Large timeout → Large Wastage

Question → Solution → Evaluation

Statistical Model

Two Problems

ParaStack 😀



- Does not guess based on *null* unlike timeout methods.
- Detects hangs based on runtime history.

Basic Concept

```
while (…) {
    user code
    MPI_Function ()
}
```

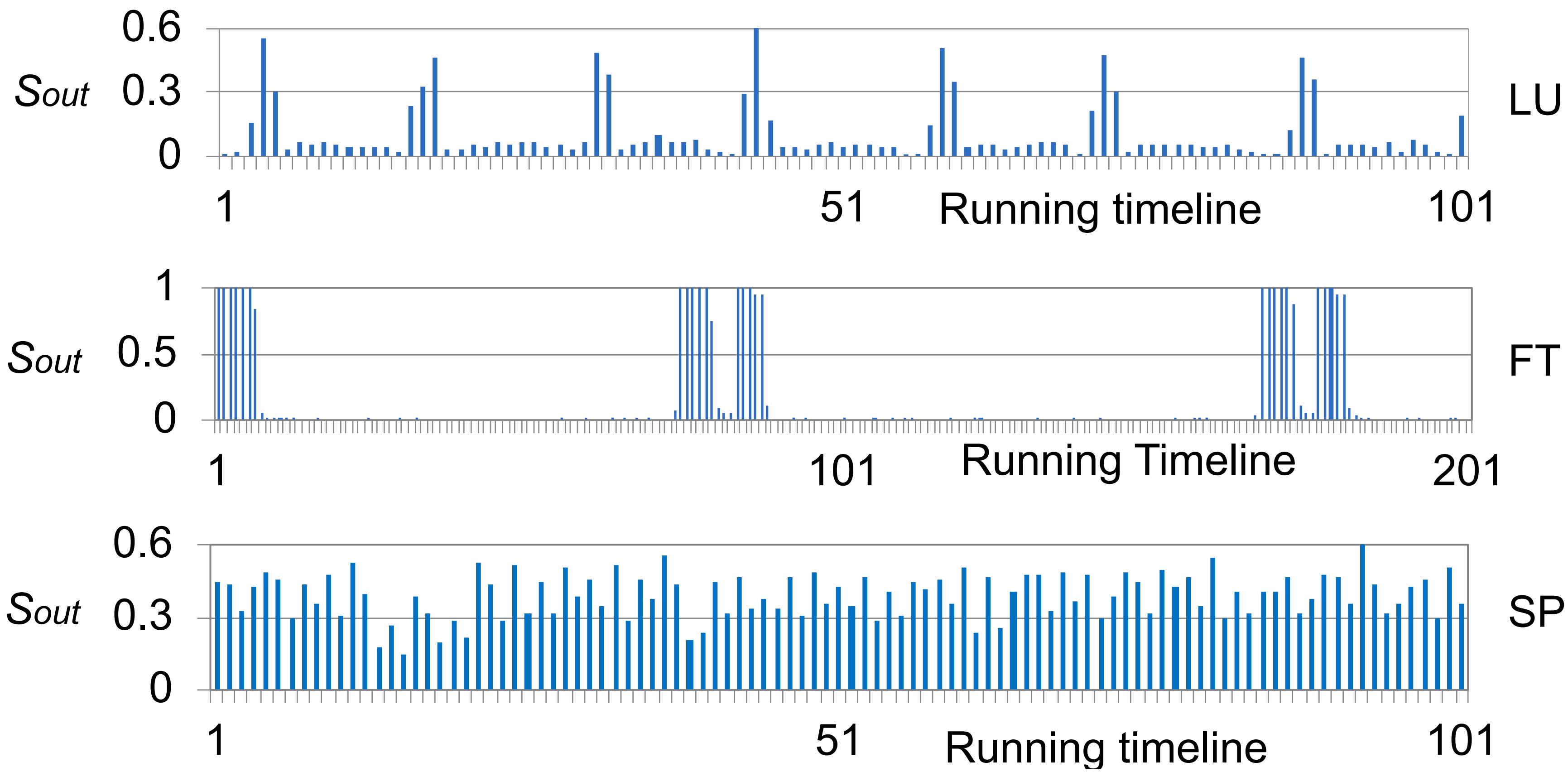
$\boldsymbol{S_{out}}$

- Definition:

$$\boldsymbol{S_{out} = \frac{N_{out}}{N_{total}}}$$

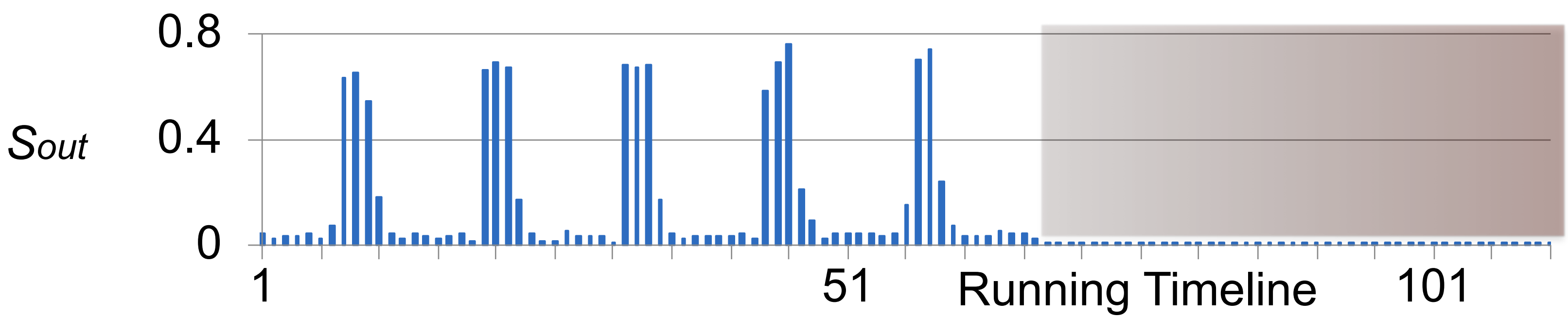
where N_{out} *denotes the number of processes executing inside user code and* N_{total} denotes the total number of processes employed in the run.

Dynamic Variation of *Sout*



A snippet of S_{out} variation obtained via sampling every 1 millisecond interval.

When a Hang Occurs



- S_{out} variation of a faulty LU run, where a fault is simulated by a very long *sleep* and injected on the left border of the red region.

- Program hang is characterized by **two features**: (1) very **small** S_{out} and (2) ***consecutive*** *observations of* (1).

Suspicion

- $\boldsymbol{F(S_{out})}$ is the *empirical cumulative distribution function* obtained from randomly sampling S_{out}.

- Given probability $\hat{p}$, we obtain $t = F^{-1}(\hat{p})$ and classify the observed value of S_{out} into **a pair of opposite random events**:

$$\begin{cases} A : \text{Suspicion} & \text{if } S_{out} \leq t, \\ \overline{A} : \text{Non-suspicion} & \text{if } S_{out} > t. \end{cases}$$

Feature 1: Small

Significance Test of Hang

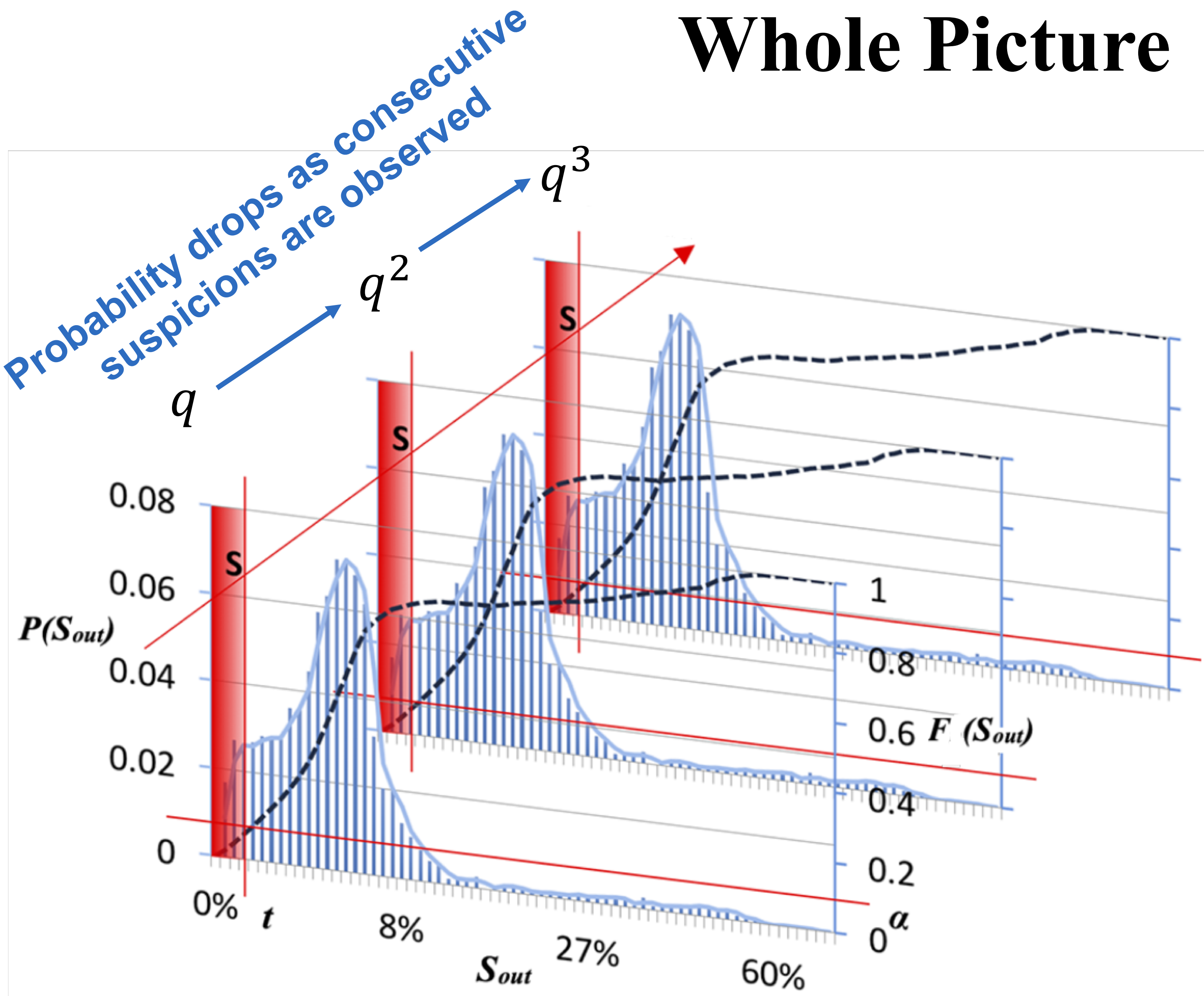
- **Geometric distribution**. The probability distribution of $Y = y$ times of suspicions before the first occurrence of non-suspicion is
$$P(Y = y) = q^y * (1 - q)$$
where q estimates the *true suspicion probability* p.

- Given the confidence level $1 - \alpha$, we **claim a hang** is detected if
$$P_{H_0}(Y \geq k) = \boldsymbol{q^k} \leq \boldsymbol{\alpha}.$$

- **Make it simple: something is very likely wrong when a very rare event occurs.**

Feature 1+2: Consecutively small

Whole Picture



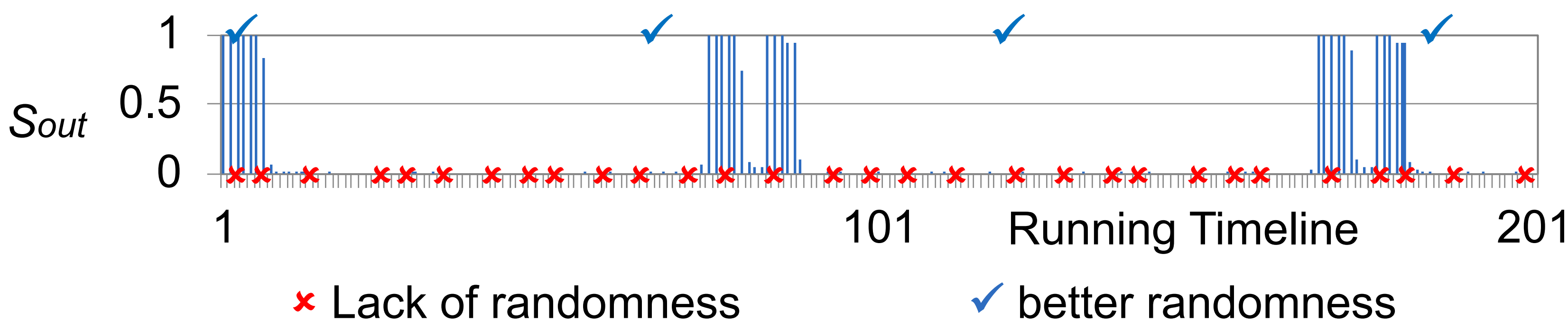
Two Problems with the Model

- (1) How to achieve random sampling?
- (2) The *observed suspicion probability* ($\hat{p}$) doesn't reflect the truth (p), i.e., $\boldsymbol{p} \neq \widehat{\boldsymbol{p}}$.

Random Sampling

- Insert between two consecutive samplings with **a random time step: $\boldsymbol{rand(I) + I/2}$**.

- Too small I → lack of randomness; Bigger I → better randomness.



- **Solution**: use **runs test** to check randomness of the sample sequence, and double I if it is found to be lack of randomness until randomness is assured.

Random Sampling (Cont.)

- Runs test --- a standard test that checks the randomness of a two-valued data sequence.

- Runs test's procedure**:**
  1) calculate the **average** of the sample sequence;
  2) denote values bigger than the average as (+) and those smaller than that as (-);
  3) check **the number of runs (*R*)** --- a run is defined as a series of consecutive (+) or (-);
  4) Too small or too large R → the sequence is lack of randomness **(significance test)**

Random Sampling (Cont.)

Example. We have a sample sequence as

$$\boxed{0.2\ 0.1\ 0.1\ 0.2\ \ 0.1\ 0.1\ 0.0\ 0.0}\ \boxed{0.8\ 0.9\ 1.0\ 0.8\ \ 0.9}\ \boxed{0.1}\ \boxed{0.9\ 0.9}$$

which can be transformed as below

$$\boxed{-\,-\,-\,-\quad -\,-\,-\,-}\ \boxed{+\,+\,+\,+\quad +}\boxed{-}\boxed{+\,+}\ .$$

Its average is 0.44375, the non-rejection region at 95% confidence is (4, 14), and $R = 4$. As $\boldsymbol{R}$ **is outside the non-rejection region**, we claim the **sampling is not random** and thus double I.

$\hat{\boldsymbol{p}} \neq \boldsymbol{p}$

- The **difference** ($\boldsymbol{d}$) between the observed *probability* ($\hat{\boldsymbol{p}}$) and the *true probability* ($\boldsymbol{p}$) is closely related to the **sample size $\boldsymbol{n}$**.

- **Solution**: Hence, we estimate $|\boldsymbol{p} - \hat{\boldsymbol{p}}| \leq \boldsymbol{d}$ at different sample size levels with high confidence (95%) :

$$\begin{cases} \hat{p} = 0.47 & d = 0.3 & \text{when } 11 \leq n < 19, \\ \hat{p} = 0.27 & d = 0.2 & \text{when } 19 \leq n < 42, \\ \hat{p} = 0.12 & d = 0.1 & \text{when } 42 \leq n < 86, \\ \hat{p} = 0.06 & d = 0.05 & \text{when } 86 \leq n. \end{cases}$$

  At each level, we use a **different credible $\hat{\boldsymbol{p}}$** to define what is a suspicion ($S_{out} \leq F^{-1}(\hat{p})$) .

- **Make it simple:** the difference gets smaller as sample size increases.

$\widehat{\boldsymbol{p}} \neq \boldsymbol{p}$ (Cont.)

- $|p - \hat{p}| \leq d$ is not enough as **underestimating** $\boldsymbol{p}$, i.e., $\widehat{\boldsymbol{p}} < \boldsymbol{p}$, lead to false positives.
  - Given $\hat{p} < p$, $\hat{p}^k$ --- the probability that a program is still healthy --- converges faster than p^k to the significance level α as k increases → more false positives.
- We **use** $\boldsymbol{q} = \widehat{\boldsymbol{p}} + \boldsymbol{d}$ **as an estimate of** $\boldsymbol{p}$ in the calculation of hangs' probability ($\boldsymbol{q}^{\boldsymbol{k}}$), which guarantees that $\boldsymbol{q} \geq \boldsymbol{p}$ with 97.5% confidence.

Question Solution Evaluation

Goal

- Trivial overhead
- High accuracy & Low false positive
- ParaStack > Timeout
- Short detection delay
- Enable resource saving when a hang occurs

Evaluation Setting

Fault injection

- A hang is simulated by injecting a long enough *sleep*() in either source code or binary.

Target Programs

- HPL, HPCG, NPB benchmark set

ParaStack's default setting

- 10 randomly selected processes are monitored.
- Significance level $\alpha = 0.1\%$.
- The initial maximal sampling interval is set as $I = 400$ ms.

Evaluation Setting (Cont.)

Number of hang-injected runs using default ParaStack

Scale	**Tardis**	**Tianhe-2**	**Stampede**
256	800+	20+	
1024		300+	100+
4096			50
8192			5
16384			3

Used notations

AC	Accuracy
FP	False positive rate
D	Average delay
S	Standard deviation of delays

Overhead, Accuracy & False Alarms

Overhead @ scale 1024 with **5 runs** on each program. We disable the automatic adaptation of I.

Benchmark	BT	CG	LU	SP	HPL	HPCG
I=100	2.44%	7.61%	3.35%	0.26%	0.12%	1.64%
I=400	-0.08%	0.55%	1.14%	0.04%	0.12%	0.35%

- **Average accuracy** → over 99% for *100 runs* of each program
- **No false alarm** reported in:
  - *39.7 hours of hang-free runs* at scale of 1024
  - *66 hours of hang-free runs* at scale of 256
  - all hang-injected runs

ParaStack v.s. **Timeout**

Platform →	Tianhe-2						Tardis					
Benchmark(Input size) →	FT(D)			FT(E)			FT(D)			LU(D)		
Metrics →	*AC*	*FP*	*D*	*AC*	*FP*	*D*	*AC*	*FP*	*D*	*AC*	*FP*	*D*
$I_1 = 400ms,\ K_1 = 5\ times$	1.0	0.0	3.3	0.0	1.0	–	0.0	1.0	–	0.0	1.0	–
$I_2 = 400ms,\ K_2 = 10\ times$	1.0	0.0	8.1	1.0	0.0	10.9	0.9	0.1	6.5	1.0	0.0	5.3
$I_3 = 800ms,\ K_3 = 5\ times$	1.0	0.0	7.2	1.0	0.0	11.7	0.8	0.2	7.0	1.0	0.0	3.9
$I_4 = 800ms,\ K_4 = 10\ times$	1.0	0.0	13.2	1.0	0.0	17.4	1.0	0.0	10.2	1.0	0.0	10.7

10 runs per setting & 256 processes

- Timeout baseline
  - Hang is claimed to be found upon ***K* consecutive observations** of $Sout \leq 0$ sampled at a **fixed interval *I***.
  - Like ParaStack, it only samples 10 processes to maintain the trivial overhead.

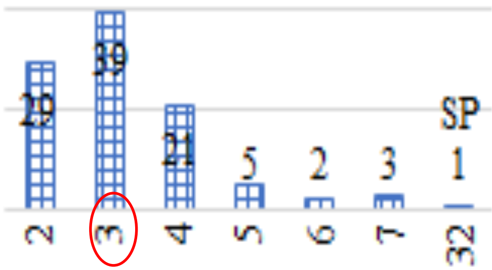
ParaStack v.s. Timeout (Cont.)

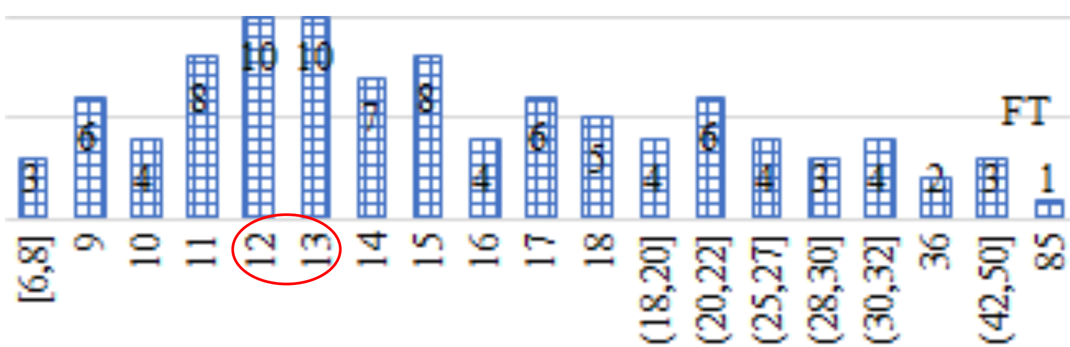
Platform	Bench.	P			P*		
		AC	*FP*	*D*	*AC*	*FP*	*D*
Tianhe-2	FT(D)	1.0	0.0	4.8	1.0	0.0	3.5
	FT(E)	1.0	0.0	29.4	1.0	0.0	14.9
Tardis	FT(D)	1.0	0.0	14.0	0.9	0.0	25.2
	LU(D)	1.0	0.0	4.5	1.0	0.0	1.1
	SP(D)	1.0	0.0	3.3	1.0	0.0	1.0

10 runs per setting & 256 processes

- Setting of ParaStack:
  - P: ParaStack initializing *I* as 400ms.
  - P*: ParaStack *initializing I as 10ms* which doesn't deliver random sampling.
- P* compares well with P as ParaStack is able to automatically adjust *I* to ensure a good model.

Detection Delay





The ***median* of detection delays** based on 100 runs per setting at scale 256.

BT	CG	LU	SP	FT	MG	HPL	HPCG
4	6	3	3	13	3	4	5

(Unit: *seconds*)

Detection Delay (Cont.)

Delay on Tianhe-2 with 50 runs per setting

Scale↓	Metric↓	BT	CG	FT	LU	SP	HPL
1024	*D*	7.2	18.8	8.8	9.0	4.8	6.8
	S	7.3	14.7	7.3	4.2	2.2	3.3

Delay on Stampede with 20 runs per setting @ scale 1024 and 10 runs per setting at scale 4096

Scale↓	BT		CG		LU		SP		HPL	
	D	*S*	*D*	*S*	*D*	*S*	*D*	*S*	*D*	*S*
1024	7.1	4.5	7.6	4.5	7.8	5.9	4.1	1.2	5.0	2.5
4096	5.4	3.6	24.1	13.1	4.3	1.3	3.7	2.0	5.6	4.7

- ParaStack detects hangs in ***a few seconds***, which is far less than the commonly used *1-minute timeout*.

Timesaving

- 10 faulty HPL runs with program hang's occurrence uniformly distributed over the program execution

- On average **35.5% time saving**

Thank you!

Any Question?