ParaStack: Efficient Hang Detection for MPI Programs at Large Scale

Hongbo Li
Zizhong Chen & Rajiv Gupta
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.

![Diagram showing process ID and time](image-url)
Hang Causes Resource Wastage

- Negative --- significant resource wastage at large scale.
Solution: Hang Detection

- Release resources when detecting a hang
- Shorter detection delay ($t_d$) $\Rightarrow$ 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
Statistical Model

Two Problems
ParaStack

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

\[
S_{out} = \frac{N_{out}}{N_{total}}
\]

Definition:

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.

while (...) {
    user code
    MPI_Function ();
}

\[\]
Dynamic Variation of $S_{out}$

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

$F(S_{out})$ 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:

\[
\begin{cases} 
A : \text{Suspicion} & \text{if } S_{out} \leq t, \\
\bar{A} : \text{Non-suspicion} & \text{if } S_{out} > t.
\end{cases}
\]
Significance Test of Hang

› **Geometric distribution.** The probability distribution of $Y = y$ times of suspicions before the first occurrence of non-suspicion is

$$P(Y = y) = q^y \times (1 - q)$$

where $q$ estimates the true suspicion probability $p$.

› Given the confidence level $1 - \alpha$, we claim a hang is detected if

$$P_{H_0}(Y \geq k) = q^k \leq \alpha.$$  

› Make it simple: something is very likely wrong when a very rare event occurs.
Probability drops as consecutive suspicions are observed.
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., $p \neq \hat{p}$.
Random Sampling

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

- Too small $I \rightarrow$ lack of randomness; Bigger $I \rightarrow$ 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 \(R\) \(\rightarrow\) the sequence is lack of randomness (**significance test**)
Random Sampling (Cont.)

Example. We have a sample sequence as

\[0.2 \, 0.1 \, 0.1 \, 0.2 \, 0.1 \, 0.1 \, 0.0 \, 0.0 \, 0.8 \, 0.9 \, 1.0 \, 0.8 \, 0.9 \, 0.1 \, 0.9 \, 0.9,\]

which can be transformed as below

\[- - - - - - - - + + + + + +.\]

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

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

- **Solution:** Hence, we estimate \(|p - \hat{p}| \leq d\) at different sample size levels with high confidence (95%) :
  \[
  \begin{align*}
  \hat{p} = 0.47 & \quad d = 0.3 & \text{when } 11 \leq n < 19, \\
  \hat{p} = 0.27 & \quad d = 0.2 & \text{when } 19 \leq n < 42, \\
  \hat{p} = 0.12 & \quad d = 0.1 & \text{when } 42 \leq n < 86, \\
  \hat{p} = 0.06 & \quad d = 0.05 & \text{when } 86 \leq n.
  \end{align*}
  \]

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

- **Make it simple:** the difference gets smaller as sample size increases.
\( \hat{p} \neq p \) (Cont.)

| \( |p - \hat{p}| \leq d \) is not enough as **underestimating** \( p \), i.e., \( \hat{p} < p \), 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 \( \alpha \) as \( k \) increases \( \rightarrow \) more false positives.

- We use \( q = \hat{p} + d \) as an **estimate of** \( p \) in the calculation of hangs’ probability \( (q^k) \), which guarantees that \( q \geq p \) with 97.5% confidence.
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.)

<table>
<thead>
<tr>
<th>Scale</th>
<th>Tardis</th>
<th>Tianhe-2</th>
<th>Stampede</th>
</tr>
</thead>
<tbody>
<tr>
<td>256</td>
<td>800+</td>
<td>20+</td>
<td></td>
</tr>
<tr>
<td>1024</td>
<td>300+</td>
<td>100+</td>
<td></td>
</tr>
<tr>
<td>4096</td>
<td></td>
<td>50</td>
<td></td>
</tr>
<tr>
<td>8192</td>
<td></td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>16384</td>
<td></td>
<td>3</td>
<td></td>
</tr>
</tbody>
</table>

**Used notations**

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

Number of hang-injected runs using default ParaStack
### Overhead, Accuracy & False Alarms

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

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>BT</th>
<th>CG</th>
<th>LU</th>
<th>SP</th>
<th>HPL</th>
<th>HPCG</th>
</tr>
</thead>
<tbody>
<tr>
<td>$I=100$</td>
<td>2.44%</td>
<td>7.61%</td>
<td>3.35%</td>
<td>0.26%</td>
<td>0.12%</td>
<td>1.64%</td>
</tr>
<tr>
<td>$I=400$</td>
<td>-0.08%</td>
<td>0.55%</td>
<td>1.14%</td>
<td>0.04%</td>
<td>0.12%</td>
<td>0.35%</td>
</tr>
</tbody>
</table>

- **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

Timeout baseline

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

<table>
<thead>
<tr>
<th>Platform → Benchmark(Input size) →</th>
<th>Tianhe-2</th>
<th>Tardis</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>FT(D)</td>
<td>FT(E)</td>
</tr>
<tr>
<td>Metrics →</td>
<td>AC</td>
<td>FP</td>
</tr>
<tr>
<td></td>
<td>AC</td>
<td>FP</td>
</tr>
<tr>
<td></td>
<td>AC</td>
<td>FP</td>
</tr>
</tbody>
</table>

| $I_1 = 400ms, K_1 = 5$ times      | 1.0 0.0 3.3 | 0.0 1.0 –     |
| $I_2 = 400ms, K_2 = 10$ times    | 1.0 0.0 8.1 | 1.0 0.0 10.9 |
| $I_3 = 800ms, K_3 = 5$ times     | 1.0 0.0 7.2 | 1.0 0.0 11.7 |
| $I_4 = 800ms, K_4 = 10$ times    | 1.0 0.0 13.2| 1.0 0.0 17.4 |

10 runs per setting & 256 processes
ParaStack v.s. Timeout (Cont.)

<table>
<thead>
<tr>
<th>Platform</th>
<th>Bench.</th>
<th>P</th>
<th>P*</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>AC</td>
<td>FP</td>
</tr>
<tr>
<td>Tianhe-2</td>
<td>FT(D)</td>
<td>1.0</td>
<td>0.0</td>
</tr>
<tr>
<td></td>
<td>FT(E)</td>
<td>1.0</td>
<td>0.0</td>
</tr>
<tr>
<td>Tardis</td>
<td>FT(D)</td>
<td>1.0</td>
<td>0.0</td>
</tr>
<tr>
<td></td>
<td>LU(D)</td>
<td>1.0</td>
<td>0.0</td>
</tr>
<tr>
<td></td>
<td>SP(D)</td>
<td>1.0</td>
<td>0.0</td>
</tr>
</tbody>
</table>

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.

10 runs per setting & 256 processes
Detection Delay

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

<table>
<thead>
<tr>
<th>BT</th>
<th>CG</th>
<th>LU</th>
<th>SP</th>
<th>FT</th>
<th>MG</th>
<th>HPL</th>
<th>HPCG</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>6</td>
<td>3</td>
<td>3</td>
<td>13</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
</tbody>
</table>

(Unit: seconds)
Detection Delay (Cont.)

**Delay on Tianhe-2** with 50 runs per setting

<table>
<thead>
<tr>
<th>Scale</th>
<th>Metric</th>
<th>BT</th>
<th>CG</th>
<th>FT</th>
<th>LU</th>
<th>SP</th>
<th>HPL</th>
</tr>
</thead>
<tbody>
<tr>
<td>1024</td>
<td>D</td>
<td>7.2</td>
<td>18.8</td>
<td>8.8</td>
<td>9.0</td>
<td>4.8</td>
<td>6.8</td>
</tr>
<tr>
<td></td>
<td>S</td>
<td>7.3</td>
<td>14.7</td>
<td>7.3</td>
<td>4.2</td>
<td>2.2</td>
<td>3.3</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Scale</th>
<th>BT</th>
<th>CG</th>
<th>LU</th>
<th>SP</th>
<th>HPL</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>D</td>
<td>S</td>
<td>D</td>
<td>S</td>
<td>D</td>
</tr>
<tr>
<td>1024</td>
<td>7.1</td>
<td>4.5</td>
<td>7.6</td>
<td>4.5</td>
<td>7.8</td>
</tr>
<tr>
<td>4096</td>
<td>5.4</td>
<td>3.6</td>
<td>24.1</td>
<td>13.1</td>
<td>4.3</td>
</tr>
</tbody>
</table>

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