Scheduling Multithreaded Computations by Work-Stealing
[Blumofe and Leiserson, 1999]

Vu Phan – COMP 522 (Rice University)

Thu 2019-03-07
work-stealing scheduling method: idle processors steal threads from busy processors

contribution: efficient randomized work-stealing algorithm for fully strict computations
Overview

challenge: efficiently executing a dynamic multithreaded computation on a MIMD computer

- parallelism not known in advance
  - dynamically growing and shrinking as computation unfolds
  - static scheduling: ill-suited
- threads depend on each other

scheduler goals:

- ensuring an appropriate number of threads are active at each step (keeping all processors busy)
- limiting memory usage of active threads
1 Introduction

2 A model of multithreaded computation

3 The busy-leaves property

4 A randomized work-stealing algorithm

5 Atomic accesses

6 Analysis of the work-stealing algorithm

7 Conclusion
Scheduling paradigms

**work-sharing:**
- scheduler migrates threads to underutilized processors (even if processors are busy)
- more thread migration

**work-stealing:**
- idle processors steal threads
- less thread migration
Fully strict computations

fully strict (well-structured) computations include:

- backtrack search
- divide-and-conquer
- data flow
efficient randomized work-stealing algorithm for scheduling fully strict multithreaded computations:

- **expected time**: $\frac{T_1}{P} + O(T_∞)$
  - $T_1$: serial time
  - $P$: number of processors
  - $T_∞$: time with $\infty$ processors

- **space**: $S_1 P$
  - $S_1$: serial space
1 Introduction

2 A model of multithreaded computation

3 The busy-leaves property

4 A randomized work-stealing algorithm

5 Atomic accesses

6 Analysis of the work-stealing algorithm

7 Conclusion
Multithreaded computation: continue-edges

- $v_1$: instruction
- $(v_1, v_2)$: continue-edge (horizontal)
- $\Gamma_6$: thread
  - activation frame
    - alive
    - dead
Multithreaded computation: spawn-edges

- $(v_2, v_3)$: **spawn-edge** (shaded)
- **spawn-tree**:
  - $\Gamma_1$: root thread
  - $\Gamma_3$: leaf thread
Multithreaded computation: join-edges

- \((v_5, v_{14})\): **join-edge** (curved)
- thread \(\Gamma_2\):
  - **ready** after \(v_2\)
  - **stalled** at \(v_{14}\)
    - **resolved join-dependency**
      - **enabled** by \(v_5\) and \(v_8\)
Multithreaded computation: execution schedule

2-processor **execution schedule**

<table>
<thead>
<tr>
<th>step</th>
<th>thread pool</th>
<th>processor activity</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>$\Gamma_1$: $v_1$</td>
<td>$p_1$: $v_1$</td>
</tr>
<tr>
<td>2</td>
<td>$v_2$</td>
<td>$p_2$: $v_2$</td>
</tr>
<tr>
<td>3</td>
<td>$\Gamma_2$: $v_3$</td>
<td>$\Gamma_1$: $v_{16}$</td>
</tr>
<tr>
<td>4</td>
<td>$\Gamma_3$: $v_4$</td>
<td>$\Gamma_1$: $v_{17}$</td>
</tr>
<tr>
<td>5</td>
<td>$\Gamma_2$: $v_5$</td>
<td>$\Gamma_6$: $v_{18}$</td>
</tr>
<tr>
<td>6</td>
<td>$\Gamma_2$: $v_6$</td>
<td>$\Gamma_4$: $v_{20}$</td>
</tr>
<tr>
<td>7</td>
<td>$\Gamma_1$: $v_7$</td>
<td>$\Gamma_1$: $v_{21}$</td>
</tr>
<tr>
<td>8</td>
<td>$v_8$</td>
<td>$\Gamma_2$: $v_9$</td>
</tr>
<tr>
<td>9</td>
<td>$\Gamma_1$: $v_{10}$</td>
<td>$\Gamma_2$: $v_{13}$</td>
</tr>
<tr>
<td>10</td>
<td>$\Gamma_1$: $v_{11}$</td>
<td>$\Gamma_1$: $v_{14}$</td>
</tr>
<tr>
<td>11</td>
<td>$\Gamma_2$: $v_{12}$</td>
<td>$\Gamma_1$: $v_{22}$</td>
</tr>
<tr>
<td>12</td>
<td>$\Gamma_1$: $v_{15}$</td>
<td>$\Gamma_2$: $v_{23}$</td>
</tr>
<tr>
<td>13</td>
<td>$\Gamma_1$: $v_{23}$</td>
<td></td>
</tr>
<tr>
<td>14</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Multithreaded computation: (full) strictness

- **strict**: each join-edge ends at an ancestor
- **fully strict** (well-structured): each join-edge ends at the parent
Multithreaded computation: work ($T_1$), span ($T_\infty$)

- **work**: number of instructions (23)
- **span (critical-path length)**: number of instructions in longest path (10)
Execution time

notations:

- $P$: number of processors
- $X$: $P$-processor execution schedule
- $T(X)$: execution time of $X$
- $T_P = \min_X T(X)$: least execution time with $P$ processors over all execution schedules $X$

observations:

1. $T_1 =$ work (number of instructions)
2. $T_\infty =$ span (length of longest path)
3. $T_P \geq T_1/P$
4. $T_P \geq T_\infty$
Greedy execution schedule

**Greedy** $P$-processor execution schedule:

- if at least $P$ instructions are ready, $P$ instructions are executed (**complete step**)
- otherwise, all ready instructions are executed (**incomplete step**)

**Theorem (1)**

*If a $P$-processor execution schedule $X$ is greedy, then $T(X) \leq T_1/P + T_\infty$.***

**Proof.**

\[
T(X) = \#\text{CompleteSteps} + \#\text{IncompleteSteps} \\
\leq T_1/P + T_\infty
\]
$P$-processor execution schedule $X$ achieves **linear speedup** when $T(X) = O(T_1/P)$

- if $X$ is greedy:
  - linear speedup is achieved when **parallelism** $T_1/T_\infty = \Omega(P)$
  - using Theorem 1: $T(X) \leq T_1/P + T_\infty$
Linear space expansion

- **stack depth of thread**: sum of sizes of activation frames of the thread and its ancestors

- **stack depth of computation**: max stack depth across all threads in the computation

- $S_1$: space usage with 1 processor (equal to stack depth of computation)

- $S(X)$: space usage of $P$-processor execution schedule $X$

- $X$ exhibits **linear space expansion** if $S(X) = O(S_1 P)$
Busy-leaves property

spawn-subtree at time step $t$: alive threads of spawn-tree

- given execution schedule $X$:
  - at time step $t$, leaf thread $\Gamma$ in the spawn-subtree is busy if some processor in $X$ is working on $\Gamma$
  - $X$ has busy-leaves property if: at every time step, all leaf threads in the spawn-subtree are busy
Busy-leaves property implying linear space expansion

Lemma (2)

If a $P$-processor execution schedule $X$ has busy-leaves property, then $S(X) \leq S_1 P$.
- $S(X)$: space usage of $X$
- $S_1$: serial space usage (stack depth of computation)

Proof.

1. by busy-leaves property: at every time step, the spawn-subtree has at most $P$ leaf threads
2. for each such leaf thread, the space used by the thread and its ancestors is $S_1$
3. at every time step, the total space used by all threads is $S_1 P$
Busy-leaves property implied by strict computation:

- after a thread $\Gamma$ is spawned and until $\Gamma$ dies, the subcomputation rooted at $\Gamma$ can be finished by 1 processor
- no leaf thread can stall

observation: if a computation is strict, then it has an execution schedule with busy-leaves property
Busy-leaves algorithm: linear speedup and linear space expansion

given a strict computation, the **busy-leaves algorithm** finds a \( P \)-processor execution schedule \( X \) such that:

- \( X \) is greedy
  - \( T(X) \leq T_1/P + T_\infty \) (Theorem 1)
    - excluding algorithm’s time to find schedule \( X \)
- \( X \) has busy-leaves property
  - \( S(X) \leq S_1 P \) (Lemma 2)
Busy-leaves algorithm: overview

- online algorithm:
  - only using information from the subcomputation revealed so far
  - no knowledge of:
    - instructions not yet executed
    - threads not yet spawned
- global pool of alive threads
  - processors take ready threads from this pool
  - processors return stalled threads to this pool
Busy-leaves algorithm: part 1

- root thread is put in global thread pool
- for each step:
  - each idle processor attempts to take a ready thread from the global thread pool
  - each busy processor executes the next instruction in a thread, until the thread:
    1. spawns
    2. stalls
    3. dies
Busy-leaves algorithm: part 2

each busy processor \( p \) executes the next instruction in a thread \( \Gamma_a \), until:

1. thread \( \Gamma_a \) spawns a child thread:
   - \( p \) returns \( \Gamma_a \) to the thread pool
   - \( p \) works on the child thread in the next step

2. thread \( \Gamma_a \) stalls:
   - \( p \) returns \( \Gamma_a \) to the thread pool
   - \( p \) becomes idle in the next step

3. thread \( \Gamma_a \) dies:
   - \( \Gamma_a \)'s parent is some thread \( \Gamma_b \)
   - if \( \Gamma_b \) has no alive child and no processor is working on \( \Gamma_b \), then \( p \) takes \( \Gamma_b \) from the thread pool and works on \( \Gamma_b \) in the next step
   - otherwise, \( p \) becomes idle in the next step
Busy-leaves algorithm: example

<table>
<thead>
<tr>
<th>step</th>
<th>thread pool</th>
<th>processor activity</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td>$p_1$: $v_1$</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>$v_2$</td>
</tr>
<tr>
<td>3</td>
<td>$\Gamma_1$</td>
<td>$\Gamma_2$: $v_3$</td>
</tr>
<tr>
<td>4</td>
<td>$\Gamma_2$</td>
<td>$\Gamma_1$: $v_4$</td>
</tr>
<tr>
<td>5</td>
<td>$\Gamma_1$</td>
<td>$\Gamma_2$: $v_5$</td>
</tr>
<tr>
<td>6</td>
<td></td>
<td>$\Gamma_6$: $v_6$</td>
</tr>
<tr>
<td>7</td>
<td>$\Gamma_1$</td>
<td>$\Gamma_2$: $v_7$</td>
</tr>
<tr>
<td>8</td>
<td>$\Gamma_2$</td>
<td>$v_8$</td>
</tr>
<tr>
<td>9</td>
<td>$\Gamma_1$</td>
<td>$\Gamma_2$: $v_9$</td>
</tr>
<tr>
<td>10</td>
<td>$\Gamma_1$</td>
<td>$\Gamma_2$: $v_{10}$</td>
</tr>
<tr>
<td>11</td>
<td>$\Gamma_1$</td>
<td>$v_{11}$</td>
</tr>
<tr>
<td>12</td>
<td>$\Gamma_2$</td>
<td>$v_{12}$</td>
</tr>
<tr>
<td>13</td>
<td>$\Gamma_1$</td>
<td>$\Gamma_2$: $v_{15}$</td>
</tr>
<tr>
<td>14</td>
<td></td>
<td>$\Gamma_1$: $v_{23}$</td>
</tr>
</tbody>
</table>

Thread pool:

- ready threads are in boldface
- stalled threads are not

Diagram showing thread pool with nodes labeled $v_1$ to $v_{23}$ and edges indicating dependencies.
for every strict computation, the busy-leaves algorithm computes a $P$-processor execution schedule $X$ such that:

- $X$ uses time $T(X) \leq T_1/P + T_\infty$
  - $T_1$: work
  - $T_\infty$: span (critical-path length)

  ($X$ is greedy)

- $X$ uses space $S(X) \leq S_1 P$
  - $S_1$: serial space

  ($X$ has busy-leaves property)
1 Introduction
2 A model of multithreaded computation
3 The busy-leaves property
4 A randomized work-stealing algorithm
5 Atomic accesses
6 Analysis of the work-stealing algorithm
7 Conclusion
each processor $p$ maintains a **ready deque** of threads

- other processors steal threads from the **top** of $p$’s ready deque
- $p$ inserts threads to the **bottom** of $p$’s ready deque
- $p$ removes threads from the **bottom** of $p$’s ready deque
Work-stealing algorithm

each processor $p$ works on a thread $\Gamma_a$, until:

1. $\Gamma_a$ spawns some thread $\Gamma_b$:
   $p$: inserts $\Gamma_a$ at the bottom of $p$’s ready deque, and starts working on $\Gamma_b$

2. $\Gamma_a$ stalls:
   - if $p$’s ready deque has some thread $\Gamma_b$:
     $p$: removes $\Gamma_b$ from $p$’s ready deque, and starts working on $\Gamma_b$
   - otherwise:
     $p$: steals the top-most thread $\Gamma_b$ of a randomly chosen processor, and starts working on $\Gamma_b$

3. $\Gamma_a$ dies: same as when $\Gamma_a$ stalls

4. $\Gamma_a$ enables some thread $\Gamma_b$: $\Gamma_b$ becomes the bottom-most thread in $p$’s ready deque
for every fully strict computation, the work-stealing algorithm needs at most $S_1 P$ space

- $S_1$: serial space
- $P$: number of processors

(the work-stealing algorithm find execution schedules with busy-leaves property)
1. Introduction
2. A model of multithreaded computation
3. The busy-leaves property
4. A randomized work-stealing algorithm
5. Atomic accesses
6. Analysis of the work-stealing algorithm
7. Conclusion
Atomic-access model

**atomic-access model:**

- parallel computer with $P$ processors
- concurrent accesses to the same data are serially queued by an adversary
  - the adversary tries to maximize the **total delay**
    (sum of numbers of outstanding access requests over all steps)
Total delay proportional to number of access requests

Lemma (6)

The total delay incurred by $M$ random access requests made by $P$ processors is:

1. $O(M + P \ln P - P \ln \epsilon)$, with probability at least $1 - \epsilon$, for every $0 < \epsilon < 1$
2. at most $M$ (expected)

very rough proof sketch:

1. tracking the **delay** of an access request
   (number of steps in which the request is waiting to be serviced)
2. linearity of expectation
1 Introduction

2 A model of multithreaded computation

3 The busy-leaves property

4 A randomized work-stealing algorithm

5 Atomic accesses

6 Analysis of the work-stealing algorithm

7 Conclusion
for every fully strict computation with work $T_1$ and span $T_∞$, the work-stealing algorithm has
time usage:

- $T_1/P + O(T_∞ + \ln P - \ln \epsilon)$, with probability at least $1 - \epsilon$, for every $0 < \epsilon < 1$
- $T_1/P + O(T_∞)$ (expected)

very rough proof sketch:

- summand $T_1/P$: $T_1$ instructions executed in parallel by $P$ processors
- summand $O(T_∞)$: scheduling overhead
  (time for steal attempts to wait before being satisfied)
  - overhead is high if many steal attempts are made
  - a large number of steal attempts can occur only with low probability
1 Introduction

2 A model of multithreaded computation

3 The busy-leaves property

4 A randomized work-stealing algorithm

5 Atomic accesses

6 Analysis of the work-stealing algorithm

7 Conclusion
Cilk

C-based language Cilk:

- runtime system employs work-stealing algorithm
- guaranteed performance to user applications
  - with high probability, linear speedup is achieved \( T_P = O(T_1/P) \), if parallel slackness \( T_1/(PT_\infty) \) is large

- applications:
  - protein folding
  - graphic rendering
  - backtrack search
  - chess
References


efficient randomized work-stealing algorithm for scheduling fully strict multithreaded computations:

- expected time: $T_1/P + O(T_\infty)$
  - $T_1$: serial time
  - $P$: number of processors
  - $T_\infty$: time with $\infty$ processors
- space: $S_1P$
  - $S_1$: serial space