Scheduling Multithreaded Computations by Work Stealing

Robert D. Blumofe
The University of Texas at Austin

Work done in collaboration with
Nimar Arora, Charles Leiserson, and Greg Plaxton

Threads and Processes

A program partitions the work into (user-level) threads to expose all of the parallelism. A computation may create millions of threads. Threads are dynamically scheduled through two levels.

Each computation has a (user-level) thread scheduler that maps its threads to its processes.

The kernel maps all processes to all processors.

Example: Cilk

Cilk programs spawn threads to express parallelism.

```cilk
int fib (int n) {
    int x, y;
    if (n < 2)
        return n;
    x = spawn fib (n-1);
    y = spawn fib (n-2);
    sync;
    return x+y;
}
```

Work Stealing

Each process maintains a “pool” of ready threads organized as a deque (double-ended queue) with a top and a bottom.

A process obtains work by popping the bottom-most thread from its deque and executing that thread.

- If the thread blocks or terminates, then the process pops another thread.
- If the thread creates or enables another thread, then the process pushes one thread on the bottom of its deque and continues executing the other.

If a process finds that its deque is empty, then it becomes a thief and steals the top-most thread from the deque of a randomly chosen victim process.
Our Results

We show that for the case of a dedicated machine with $P$ processes executing on $P$ processors, the execution time $T$ of the work-stealing algorithm satisfies the following bound.

$$E[T] \leq O(T_{1} \cdot P \cdot T_{1}).$$

- $T_{1}$ is the work, the execution time with 1 processor.
- $T_{1}$ is the critical-path length, the theoretical minimum execution time with infinitely many processors.
- This bound is optimal to within a constant factor.
- For any $\alpha \geq 0$, we have $T \leq O(T_{1} \cdot P \cdot T_{1} \cdot \lg(1 + \gamma))$ with probability at least $1 - \epsilon$.

(Blumofe & Leiserson, FOCS 1994)

Outline

- The dag model
  - The model
  - Simple bounds
  - Dag scheduling
    - Structural Lemma
    - Time analysis
    - Conclusion

Introduction to Dag Model

A multithreaded computation is modeled as a dag (directed acyclic graph).

- The dag models the execution of a multithreaded program.
- The nodes represent executed instructions.
- The edges define a partial order on the instructions.

Dag Model: Example I

Cilk procedures spawn children and then sync, waiting for the children to terminate.
Dag Model: Example II

Threads may use synchronization variables such as locks, condition variables, and semaphores.

semaphore V

semaphore P

• Each thread is a chain of nodes.
• Inter-thread edges arise from spawning and synchronizing.

Simple Bounds

Let $T_P$ be the minimum possible execution time with $P$ processors.

Lower bounds:

• $T_P \geq T_1 P$. Each processor can execute at most 1 node per time step.
• $T_P \geq T_1$. A node cannot be executed until after all of its predecessors.

Upper bound:

• $T_P \leq T_1 P \cdot T_1$. “Brent schedules” and “greedy schedules” meet this bound.

Dag Model

• Each node represents one unit of work and takes one time step to execute.
• We assume a single source node and out-degree at most 2.
• The work $T_1$ is the number of nodes. The critical-path length $T_1$ is the length of a longest (directed) path.
• A node is ready if all of its ancestors have been executed. Only ready nodes can be executed.

Scheduling Dags by Work Stealing

We ignore threads and view the algorithm as scheduling the nodes of the dag.

• We replace each ready thread with its unique ready node.
• For any process, the thread currently being executed is its assigned thread.
• The ready node of the assigned thread is the assigned node.
Dag-Scheduling Loop

```java
while (!computationDone) {
    while (!assignedNode)
        assignedNode = randomProcess().popTop();
    numChild, child = execute (assignedNode);
    if (numChild == 0)
        assignedNode = popBottom();
    else if (numChild == 1)
        assignedNode = child[0];
    else if (numChild == 2) {
        pushBottom (child[0]);
        assignedNode = child[1];
    }
}
```

Simplifying Assumptions

To simplify this presentation, we make the following assumptions:

- Execution is step-by-step synchronous.
- At each step, each process executes one iteration of the scheduling loop.
- If multiple processes try to pop the same node from the same deque at the same step, then exactly one (arbitrarily chosen) succeeds and the others fail (returning 0).

Outline

- The dag model
- Structural Lemma
  - Enabling tree
  - Structural Lemma
  - Structural Corollary
- Time analysis
- Conclusion

Enabling Tree

- For any (non-root) node $v$, suppose node $u$ is the last of $v$’s parents to be executed.
  - The execution of node $u$ enables node $v$.
  - Node $u$ is the designated parent of $v$.
  - Edge $(u,v)$ is an enabling edge.
  - The enabling edges form an enabling tree.
**Structural Lemma**

*Structural Lemma*: For any deque, at all times during the execution of the work-stealing algorithm, the designated parents of the nodes in the deque lie on a root-to-leaf path in the enabling tree.

Consider any process at any time during the execution.
- $v_0$ is its assigned node.
- $v_1, v_2, \ldots, v_k$ are the ready nodes in its deque ordered from bottom to top.
- For $i \mod 0, 1, \ldots, k$, node $u_i$ is the designated parent of $v_i$.

Then:
- For $i \mod 1, 2, \ldots, k$, node $u_i$ is an ancestor of $u_{i+1}$ in the enabling tree.
- For $i \mod 2, \ldots, k$, we have $u_i + u_{i+1}$.

**Structural Lemma: Proof**

*Proof*: By induction on the number of steals and assigned-node executions since the deque was last empty.

- *Base case*: If the deque is empty, then the lemma holds vacuously.
- *Induction hypothesis*: Consider a steal or an assigned-node execution, and assume that the lemma holds beforehand.
- *Induction step*: Show that the lemma holds afterwards.
  4 cases: 
  - **(S)** Top node is stolen.
  - **(E0)** Assigned node enables 0 children.
  - **(E1)** Assigned node enables 1 child.
  - **(E2)** Assigned node enables 2 children.

**Structural Lemma: Proof Case (S)**

The top node $v_k$ is stolen.

**Structural Lemma: Proof Case (E0)**

Execution of assigned node $v_0$ enables 0 children.
**Structural Lemma: Proof Case (E1)**

Execution of assigned node $v_0$ enables 1 child $v_a$.

$v_a$ is assigned.

$v_0$ is the designated parent of $v_a$.

---

**Structural Lemma: Proof Case (E2)**

Execution of assigned node $v_0$ enables 2 children $v_a$ and $v_b$.

$v_a$ is pushed on bottom; $v_b$ is assigned.

$v_0$ is the designated parent of both $v_a$ and $v_b$.

---

**Structural Corollary**

Each node $u$ has weight $w(u) \% T_1(d(u))$, where $d(u)$ is the depth of $u$ in the enabling tree.

*Structural Corollary:* For any deque, at all times during the execution of the work-stealing algorithm, the weights of the nodes in the deque increase from bottom to top.

Consider any process at any time during the execution.

* $v_0$ is its assigned node.
* $v_1, v_2, \ldots, v_k$ are the ready nodes in its deque ordered from bottom to top.

Then:

* $w(v_0) \% w(v_1) \% \ldots \% w(v_k)$.

---

**Outline**

- The dag model
- Structural Lemma
- Time analysis
  - Accounting
  - Analysis of steals
  - Analysis of work stealing
- Conclusion
Accounting I

To analyze the work-stealing algorithm we use an accounting argument. At each time step, each process pays one token.

- If the process executes a node of the dag, then it places a token in the work bucket. Execution ends with $T_1$ tokens in the work bucket.

- If the process makes a steal attempt, then it places a token in the steal bucket. Let $S$ denote the number of tokens in the steal bucket when execution ends.

Accounting II

- At each step, at least $P$ tokens are collected and each step takes constant time, so the execution time is $T \% O(T_1P')$.

- We will prove $E[S] \% O(T_1P)$ by an amortization argument based on a potential function.

We will conclude $E[T] \% O(T_1P' T_1)$.

Potential Function

*We use a potential function to bound the number of steal attempts.*

Each ready node $u$ has potential $-(u) \% 3^{w(u)}$. (Recall weight is $w(u) \% T_1$ (d(u)) where d(u) is depth of $u$ in enabling tree.)

The potential $\cdot i$ at step $i$ is the sum of all ready node potentials.

- The initial potential is $\cdot 0 \% 3^T_1$.
- The final potential is $\cdot T \% 0$.
- Execution of a node $u$ causes potential decrease.

Potential decrease:

$$-(u) \% 3^{w(u)} - (v_1) \% 3^{w(u)(1)} - (v_2) \% 3^{w(u)(1)}$$

Top-Heavy Deques

At each step $i$, we think of the total potential $\cdot i$ as being partitioned among the $P$ processes.

The potential $\cdot (q)$ associated with process $q$ is the sum of the potentials of all of the nodes in $q$’s deque and $q$’s assigned node.

*Top-Heavy-Deques Lemma:* For any process at any time step during the execution of the work-stealing algorithm, the potential of the topmost node in the deque contributes at least $142$ of the potential associated with the process.

- $-(u) \% (142) \cdot (q)$, where $u$ is the topmost node in $q$’s deque at step $i$.

*Proof:* From structural corollary. Potential of nodes below $u$ decreases geometrically.
Balls and Weighted Bins

Consider throwing $P$ balls at random into $P$ weighted bins.

- For each bin $i = 1, 2, \ldots, P$, bin $i$ has weight $W_i$. Let $W% / \% W_i$.
- Random variable $X_i$ is $W_i$ if a ball lands in bin $i$ and 0 otherwise. Let $X% / \% X_i$.

**Balls-and-Weighted-Bins Lemma:** \( \Pr\{X \neq W\} \geq 1 - (1 - e^{-1})P \).

---

Analysis of Steal Attempts I

**Steal-Attempts Lemma:** $P$ steal attempts cause the potential to decrease by a factor of at least $112$ with probability at least $14$.  

**Proof:** Consider a step $i$ and $P$ subsequent steal attempts. Partition the potential as $\% D_i ' E_i$, where $D_i$ is the potential associated with processes whose deque is non-empty and $E_i$ is the potential associated with processes whose deque is empty.

- If $q$’s deque is empty, then execution of $q$’s assigned node $u$ causes potential decrease of at least $((13) - (u)) \% (13) \cdot (q)$.
- Thus, the potential decreases by at least $((13)E_i)$.

- If $q$’s deque is not empty, then if a steal attempt chooses $q$ as the victim, the topmost node $u$ will be stolen and executed, causing potential decrease of at least $(13) - (u)$, which by the Top-Heavy-Deques Lemma is at least $(13)(112) \cdot (q) \% (116) \cdot (q)$.

---

Analysis of Steal Attempts II

- Consider the $P$ processes as bins and the $P$ steal attempts as ball throws. For each process $q$, if its deque is non-empty, then it is given weight $(116) \cdot (q)$, otherwise it is given weight 0. The total weight is $W% (116)D_i$.
- Thus, from the Balls-and-Weighted-Bins Lemma with $\% (112)$, the potential decreases by at least $\% (112)(116)D_i \% (1112)D_i$ with probability at least $1 - (1 - e^{-1})P \# 114$.

- Since $\% (112) ' E_i$, the potential decreases by at least $(1112) \% (q)$ with probability at least $14$.

---

Work-Stealing Theorem I

**Work-Stealing Theorem:** For any number $P$ of (dedicated) processors and any multithreaded computation with work $T_1$ and critical-path length $T_i$, the work-stealing algorithm runs in expected time $E[T] \% O(T_1P ' T_i)$.

**Proof:** It remains only to show that the expected number of tokens in the steal bucket is $E[S] \% O(T_1P)$. We divide the execution into phases of $P$ consecutive steal attempts, and we show that the expected number of phases is $O(T_i)$.

- A phase is **successful** if the potential decreases by a factor of at least $112$.
- By the Steal-Attempts Lemma, a phase is successful with probability at least $14$.  

---
Work-Stealing Theorem II

- After $k$ successful phases, the potential is at most $(1112)^k \cdot \theta$ \% $(1112)^{3T_1}$.
- When the potential drops below 1, the execution is complete, so the number of successful phases is at most $k \% (\log_{12}(3)T_1)$.
- The expected number of phases before $(\log_{12}(3)T_1)$ successes, is $4(\log_{12}(3)T_1) \% O(T_1)$.

Summary of Results

Work stealing is a user-level thread-scheduling algorithm that is efficient in theory and in practice.

- **Theory:** $E[T] \% O(T_1P \cdot T_1)$.
- **Practice:** $T_2 T_1 P \cdot T_1$.

With a “non-blocking” implementation of work stealing, this result can be generalized to the case when the number $P$ of processes exceeds the number of processors or when the number of processors grows and shrinks over time. $P_A$ is the time-average number of processors.

- **Theory:** $E[T] \% O(T_1 P_A \cdot T_1 P A_p P_A)$.
- **Practice:** $T_2 T_1 P A \cdot T_1 P A_P A$.