Work-stealing

October 14, 2014
Context

- A parallel platform with $p$ processors
- A task-graph $G$ to be executed
- Non-clairvoyant setting: the structure of $G$ and/or the execution times of its constitutive tasks are discovered online
Work-sharing approach

Centralized scheduling
- A single list stores all ready tasks
- All processors retrieve work from that list

Advantage(s)
- Global view and knowledge

Drawback(s)
- Does not scale (contentions, etc.)
Work-stealing approach

Distributed scheduling

- Each processor owns a list of “its” ready tasks

Advantage(s)

- No contention problem
- Scalable solution

Drawback(s)

- Processors with empty lists do not know where to retrieve work from
Stealing policies 1/2

Global round-robin
- A global variable holds the identity of the next processor to steal from
- Variable incremented after each steal (successful or not)
- Advantage: eventual progress
- Drawback: centralized solution...

Local round-robin
- Each processor has its own variable indicating the next processor it should try to steal from
- Variable incremented after each steal (successful or not)
- Advantage: eventual progress; solution is scalable
- Drawback: all stealing processors may attempt to steal from the same processor; arbitrary notion of “distance” between processors
Random stealing (Blumofe and Leiserson)

- The processor to steal from is randomly and uniformly chosen
- Advantage: decentralized; scalable; no notion of “distance”; low probability of simultaneous steal from same processor
- Drawback: performance???
Execution time as a function of number of steals

Assumption

- A steal takes a unit time (whether it is successful or not) (Hence, contentions are taken into account)

Notation

- Overall work: $W$ (execution time on a single processor)
- Overall execution time: $T_p$
- Number of steal attempts: $S$

$$p \times T_p = W + S$$

(A processor is either working or doing a steal attempt)
The work-stealing algorithm

Principle
- Deepest-first order for execution
- Breadth-first order for steals

Specification
- Each processor stores ready tasks in a deque (double-ended queue)
- A new ready task is stored at the bottom of the deque
- The next task to be (locally) processed is taken from the bottom of the deque
- A task is stolen from the top of a randomly picked deque
Algorithm performance

Assumptions

- The DAG has a single entry node
- The DAG has a depth $D$
- The maximum out-degree of a node is 2
- A node has a unit execution time

Number of steal attempts

$$E[S] = O(p \times D)$$

With probability at least $1 - \epsilon$, the number of steals is bounded by

$$S = O\left(p \left(D + \log\left(\frac{1}{\epsilon}\right)\right)\right)$$
Enabling tree

- If execution of node $u$ enables node $v$
  - $(u, v)$ is an **enabling edge**
  - $u$ designated parent of $v$
  - Every node (except the root) has exactly one designated parent
- The enabling edges generate an **enabling tree**
- Node $u$ of depth $d(u)$ in the enabling tree has **weight** $w(u) = D - d(u)$
Theorem

The designated parents of the nodes in the deque lie on some root-to-leaf path in the enabling tree

Proof by induction

- Initial case: trivial when deque is empty
- Induction
  - Trivial in the case of a steal
  - Trivial when an execution complete when the deque was empty: at most two ready nodes, at most one in the deque
  - What about execution completion when the deque was not previously empty?
    If a single new ready node: no problem (this node is processed right away)
Execution completion when the deque was not previously empty

(a) Before. (b) After.

Figure 9: The deque of process orb before and after the execution of the assign node enables 2 children.

We consider various restrictions on kernel behavior in order to demonstrate environments in which the running time of the work stealer is optimal.

The following definitions will prove to be useful in our analysis. An instruction in the sequence executed by some process is a milestone if and only if one of the following two conditions holds: (i) execution of an odely process occurs at that instruction, or (ii) a popTop invocation completes. From the scheduling loop of Figure 3, we observe that a given process may execute a total of some constant number of instructions between successive milestones. Throughout this section, we elect to denote a sufficiently large constant such that in any sequence of consecutive instructions executed by a process, at least one milestone.

The remainder of this section is organized as follows. Section 4.1 reduces the analysis to bounding the number of “throws”. Section 4.2 defines a potential function that is central to all of our upper-bound arguments. Sections 4.3 and 4.4 present our upper bounds for dedicated and multiprogrammed environments.

4.1 Throws

In this section we show that the execution time of our work stealer is where is the number of “throws”, that is, steal attempts satisfying a technical condition stated below. This goal cannot be achieved without restricting the kernel, so in addition to proving this bound on execution time, we shall state and justify certain kernel restrictions.

One fundamental obstacle prevents us from proving the desired performance bound within the (unrestricted) multiprogramming model of Section 2. The problem is that the kernel may bias the random steal attempts towards the empty deques. In particular, consider the steal attempts initiated within some fixed interval of steps. The adversary can bias these steal attempts towards the empty deques by delaying those steal attempts that choose nonempty deques as victims so that they occur after the end of the interval. To address this issue, we restrict the kernel to schedule in rounds rather than steps. A process that is scheduled in a particular round executes between and instructions during the round, where is the constant defined at the beginning of Section 4. The precise number of instructions that a process executes during a round is determined by the kernel in an arbitrary manner. We assume that the process executes these to instructions in serial order, but we allow the instruction streams of different processes to be interleaved arbitrarily, as determined by the kernel. We claim that our requirement that processes be
Amortized analysis using a potential function

Potential function

- Let $R_i$ be the set of ready nodes at the beginning of round $i$
- Each ready node $u \in R_i$ has a potential $\phi_i(u)$:

$$\phi_i(u) = \begin{cases} 
3^{2w(u)-1} & \text{if } u \text{ is processed} \\
3^{2w(u)} & \text{otherwise (} u \text{ is in a deque)}
\end{cases}$$

- Potential at round $i$:

$$\Phi_i = \sum_{u \in R_i} \phi_i(u)$$

- Initial potential: $\Phi_0 = 3^{2D-1}$
- Final potential: 0
Potential never increases

Two potential-changing actions

- Node $u$ is removed from a deque (either through work-stealing or because the completion of the previous processing did not enable any node)
  \[ \phi_i(u) - \phi_{i+1}(u) = 3^{2w(u)} - 3^{2w(u)-1} = \frac{2}{3}3^{2w(u)} = \frac{2}{3}\phi_i(u) \]

- Completion of a processed node (enabling some nodes)
  Completion of node $u$ enables nodes $x$ and $y$: $x$ is processed and $y$ placed in the deque
  \[ \phi_i(u) - \phi_{i+1}(x) - \phi_{i+1}(y) = 3^{2w(u)-1} - 3^{2w(x)-1} - 3^{2w(y)} \]
  \[ = 3^{2w(u)-1} - 3^{2(w(u)-1)} - 3^{2(w(u)-1)} \]
  \[ = 3^{2w(u)-1} (1 - \frac{1}{9} - \frac{1}{3}) \]
  \[ = \frac{5}{9}\phi_i(u) > 0 \]
Partitioning processors

- $q$ a processor: $R_i(q)$ set of ready nodes in $q$’s deque plus the node it processes

$$\Phi(q) = \sum_{u \in R_i(q)} \phi_i(u)$$

- $A_i$: set of processors whose deque is empty
- $D_i$: set of other processors

$$\Phi_i = \Phi_i(A_i) + \Phi_i(D_i)$$

- Aim: prove that every $p$ steal attempts the potential decreases by a constant fraction with constant probability
Top-heavy deques

**Theorem**

Let $q$ be a processor in $D_i$. The topmost node $u$ in $q$’s deque is such that:

$$\phi_i(u) \geq \frac{3}{4} \Phi_i(q)$$

**Proof**

- Let $y$ be the node processed by $q$
- Suppose $u$ is the only node in the deque
- Suppose $u$ and $y$ have the same designated parent

\[
\Phi_i(q) = \phi_i(u) + \phi_i(y) = 3^{2w(u)} + 3^{2w(y)} - 1 = 3^{2w(u)} + 3^{2w(u)} - 1 = \frac{4}{3} \phi_i(u)
\]
Theorem

Suppose that $p$ balls are thrown independently and uniformly at random into $p$ bins, where bin $i$ has weight $W_i$, with $\sum_i W_i = W$. For each bin $i$ we define the random variable $X_i$ as:

$$X_i = \begin{cases} W_i & \text{if some ball lands in bin } i \\ 0 & \text{otherwise} \end{cases}$$

Let $X = \sum_i X_i$. For any $\beta$, $0 < \beta < 1$:

$$\Pr(X \geq \beta W) > 1 - \frac{1}{(1 - \beta)e}$$
Consider any round $i$ and a later round $j$ such that $p$ steal attempts occurred from round $i$ (included) to round $j$ (excluded). Then:

$$\Pr \left( \Phi_i - \Phi_j \geq \frac{1}{4} \Phi_i(D_i) \right) > \frac{1}{4}$$
Impact of $p$ steal attempts 2/3

- Let $q \in D_i$
- Let $u$ be the node at the top of $q$’s deque at round $i$
- We assume one of the $p$ steal attempts target $q$
- Cases
  1. $u$ is stolen
  2. Another node is stolen: therefore $u$ was assigned
  3. No node is stolen
     1. $u$ was previously stolen
     2. $q$ has started the processing of $u$

In any case, at the very least $u$ is processed and the potential decreased by at least $\frac{2}{3} \phi_i(u)$

$$\frac{2}{3} \phi_i(u) \geq \frac{2}{3} \frac{3}{4} \Phi_i(q) = \frac{1}{2} \Phi_i(q)$$
Impact of \( p \) steal attempts 3/3

We consider a series of \( p \) steal attempts

- If a steal attempt targets \( q \in D_i \), the potential decreases by \( \frac{1}{2} \Phi_i(q) \)
- \( \forall q \in D_i, \ W_q = \frac{1}{2} \Phi_i(q) \)
- \( \forall q \in A_i, \ W_q = 0 \)
- \( W = \frac{1}{2} \Phi_i(D_i) \)
- We use the “Balls and weighted bins theorem” with \( \beta = \frac{1}{2} \)

The potential decreases by \( \beta W = \frac{1}{4} \Phi_i(D_i) \) with a probability greater than
\[
1 - \frac{1}{(1 - \frac{1}{2})e} = 1 - \frac{2}{e} > \frac{1}{4}
\]
A phase is defined by a series of \( \Theta(P) \) steal attempts

Phase starting with round \( i \) and ending with round \( j \) (excluded)

\[ \Phi_i = \Phi_i(A_i) + \Phi_i(D_i) \]

Potential loss due to the steal attempts: at least \( \frac{1}{4} \Phi_i(D_i) \) with probability at least \( \frac{1}{4} \)

Potential loss due to task completion on \( A_i \)
If node \( u \) completes, potential drops by at least \( \frac{5}{9} \phi(u) > \frac{1}{4} \phi(u) \).
Overall: greater than \( \frac{1}{4} \Phi_i(A_i) \)

\[ Pr(\Phi_i - \Phi_j > \frac{1}{4} \Phi_i) > \frac{1}{4} \]
Phase is **successful** if potential drops by at least $\frac{1}{4}$

- Initial potential: $\Phi_0 = 3^{2D-1}$
- Final potential: 0
- Maximal number of successful phases: $S$
  
  $$
  \left(\frac{3}{4}\right)^S \times 3^{2D-1} < 1 \implies S \text{ is at most } (2D-1) \log_3(3) < 8D
  $$

  - The expected number of phases is then at most $32D$
  - The expected number of steal attempts is then $O(p \cdot D)$
  - The probability that the execution takes $64D + 16 \ln \left(\frac{1}{\epsilon}\right)$ phases or more is less than $\epsilon$
  - The number of steal attempts is $O\left(\left(D + \log \left(\frac{1}{\epsilon}\right)\right) p\right)$ with probability at least $1 - \epsilon$
Algorithm performance

Assumptions

- The DAG has a single entry node
- The DAG has a depth $D$
- The maximum out-degree of a node is 2
- A node has a unit execution time

Number of steal attempts: $\mathbb{E}[S] = O(p \times D)$

With probability at least $1 - \epsilon$, number of steals is bounded by

$$S = O\left(p \left(D + \log\left(\frac{1}{\epsilon}\right)\right)\right)$$

$$\mathbb{E}(T_p) = \frac{\mathcal{W}}{p} + O(D)$$

and

$$T_p = O\left(\frac{\mathcal{W}}{p} + D + \log\left(\frac{1}{\epsilon}\right)\right)$$

with probability $\geq 1 - \epsilon$
What about the assumptions?

- The DAG has a single entry node
  Transformation increases $D$ by $\lceil \log_2(I) \rceil$ where $I$ is the number of entry nodes.
- The maximum out-degree of a node is 2
  Transformation multiplies $D$ by $\lceil \log_2(\delta) \rceil$
- A node has a unit execution time
  In fact: maximum execution time is unit time
  Generalization: multiply number of steal attempts by the duration of the longest task...
Conclusion

- Not a list scheduling approach: because there are no centralized scheduler a processor may be left idle when there is ready nodes

\[ E(T_p) = \frac{\mathcal{W}}{\rho} + O(D) \Rightarrow E(T_p) = O(T_{opt}) \]

- Many existing variants of random work stealing: Try to take advantage of (data) locality, to avoid lengthy communications, etc.