Fork-join model and work stealing
CRSPP reading group

Arthur Charguéraud and Mike Rainey

MPI-SWS

June 19, 2011
Outline

- Background: design and analysis of parallel algorithms
- Scheduling parallel algorithms on multiprocessor machines
- Scheduling by work stealing
- Implementation of work stealing
Implicit parallelism

Divide and conquer style is typical for parallel algorithms.

type tree =
  | Leaf of int
  | Node of tree * tree

let rec sum t =
  match t with
  | Leaf n -> n
  | Node (t1,t2) ->
    (* allow recursive calls to go in parallel *)
    let (n1, n2) = (| sum t1, sum t2 |) in
    n1 + n2
Visualizing the task graph

parallel tuple expression

\[
\text{sum (Node (Leaf 5, Node (Leaf 23, Leaf 8)))}
\]

series-parallel DAG
Series-Parallel DAGs

SP DAGs are built inductively from:

\[ G \]

\[ G_1 \]

\[ G_2 \]

\[ G_1 \parallel G_2 \]

task

serial composition

parallel composition

data / control

dependency

fork task

join task

Arthur Charguéraud and Mike Rainey
Work and span in SP DAGs

We use SP DAGs as basis for a cost model to estimate benefit of parallelism.

span = length of critical path
work = number of nodes

(convention: from hereon, edges will always point down implicitly)
The sum function applied to a balanced tree with 16 leaf nodes.

<table>
<thead>
<tr>
<th>Work</th>
<th>Span</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>8</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Total: 22

Total: 6
Defining costs

Let $T_P$ denote the time to execute a given SP DAG with $P$ processors.

- $T_1$ corresponds to the work.
- $T_\infty$ corresponds to the span.
- Speedup with $P$ processors is $\frac{T_1}{T_P}$.
  - Speedup $P$ means “perfect linear”.
  - Speedup 1 means adding more processors does not help.
The average parallelism

The *average parallelism* \( P_{\text{avg}} \) is \( \frac{T_1}{T_\infty} \), the ratio of work and span.

- Average parallelism represents the maximum speedup regardless of # processors.
  - Proof:
    - Speedup = \( \frac{T_1}{T_P} \)
    - \( T_P \geq T_\infty \), for any \( P \)
    - Therefore, speedup \( \leq \frac{T_1}{T_\infty} \)

- We can use \( P_{\text{avg}} \) to
  - estimate how parallel a given algorithm is
  - compare parallelism of different algorithms

- Examples:
  - `sum` (balanced tree): large \( P_{\text{avg}} \)
  - `sum` (unbalanced tree): small \( P_{\text{avg}} \)
  - list-based mergesort: small \( P_{\text{avg}} \)
  - tree-based quicksort: large \( P_{\text{avg}} \)
Complexity of $\text{sum}$

- Suppose the input tree contains $n$ leaves and has height $h$.
  - $T_1 = O(n)$
  - $T_\infty = O(h)$

- Example 1: the input tree is balanced ($h = \log_2 n$)
  - Large average parallelism $P_{avg} = O\left(\frac{n}{\log n}\right)$
  - For instance, if $n = 2^{20}$, then $P_{avg} = \frac{2^{20}}{20} \approx 100,000$.
  - That is more than enough parallelism to utilize many processors.

- Example 2: the input tree is not balanced ($h = n$, e.g., a list)
  - Small average parallelism $P_{avg} = 1$.
  - No benefit from having more than one processor.
Merging two sorted lists

let rec merge (xs : int list, ys : int list) =
    match (xs, ys) with
    | ([], ys) -> ys
    | (xs, []) -> xs
    | (x::xs, y::ys) ->
      if x < y then
        x :: merge (xs, y::ys)
      else
        y :: merge (x::xs, ys)
let rec mergesort (xs : int list) =
    match xs with
    | [] -> []
    | [x] -> [x]
    | xs ->
        let med = length xs / 2 in
        let (left, right) =
            (take med xs, drop med xs) in
        merge (| mergesort left, mergesort right |)
Complexity of mergesort

\[ T_\infty = 2cn \]

input list length = \( n \)

\[ T_1 = cn \log n \]

c is some constant

Arthur Charguéraud and Mike Rainey
Parallelism of mergesort

- Then the average parallelism $P_{avg} = \frac{cn \log n}{2cn} = \frac{\log n}{2}$.
- If $n = 2^{20}$, then $P_{avg} = \frac{\log 2^{20}}{2} = 10$.
- That is terrible: greatest speedup we can ever hope to achieve is 10x.
- Can we do better?
  - There exists a parallel functional mergesort with
    - $T_1 = O(n \log n)$
    - $T_\infty = O(\log^3 n)$
    - $P_{avg} = O\left(\frac{n}{\log^2 n}\right)$
  - Basic ideas:
    - balanced trees (or arrays) instead of linked lists
    - parallelize the merging phase
  - See Blelloch & Greiner 1995 for more details.
Tree-based quicksort

type tree = Empty | Leaf of int | Node of tree * tree

let append (xs, ys) =  
  match (xs, ys) with  
  | (Empty, ys) -> ys  
  | (xs, Empty) -> xs  
  | _ -> Node (xs, ys)

let rec filter (f : int -> bool) (xs : tree) =  
  match xs with  
  | Empty -> Empty  
  | Leaf x -> if f x then Leaf x else Empty  
  | Node (xs, ys) ->  
    let (xs', ys') =  
      (| filter f xs, filter f ys |) in  
    append (xs', ys')
Tree-based quicksort

```
let rec quicksort (xs : tree) =
  match xs with
  | Empty -> Empty
  | Leaf x -> Leaf x
  | _ ->
    let pivot = first xs in
    let less = filter (fun x -> x < pivot) xs in
    let greater = filter (fun x -> x > pivot) xs in
    let equal = filter (fun x -> x = pivot) xs in
    let (left, right) =
      (| quicksort less, quicksort greater |) in
    append (left, append (equal, right))
```
Complexity of **quicksort** (assuming the input tree is balanced)

\[ T_\infty = O(\log^2 n) \]

\[ P_{avg} = O((n \log n) / \log n) = O(n / \log n) \quad \text{(good!)} \]
Summary

<table>
<thead>
<tr>
<th></th>
<th>( T_1 )</th>
<th>( T_\infty )</th>
<th>( P_{avg} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>sum (balanced trees)</td>
<td>( O(n) )</td>
<td>( O(\log n) )</td>
<td>( O\left( \frac{n}{\log n} \right) )</td>
</tr>
<tr>
<td>sum (unbalanced trees)</td>
<td>( O(n) )</td>
<td>( O(n) )</td>
<td>( O(1) )</td>
</tr>
<tr>
<td>mergesort</td>
<td>( O(n \log n) )</td>
<td>( O(n) )</td>
<td>( O(\log n) )</td>
</tr>
<tr>
<td>quicksort</td>
<td>( O(n \log n) )</td>
<td>( O(\log^2 n) )</td>
<td>( O\left( \frac{n}{\log n} \right) )</td>
</tr>
</tbody>
</table>

- Lists are bad.
- Trees are good.
- List-to-tree adaptation gives good results for a number of algorithms.
- Sometimes algorithms need to be redesigned.
Scheduling SP DAGs on multiprocessor machines

- Scheduling is mapping suparts of SP DAGs to finitely-many processors.
- Goal: to minimize execution time.
- Scheduler discovers the structure of the SP DAG as it goes.
  - *i.e.*, online scheduling
- The scheduling policy
  - determines order in which tasks are executed
  - mappings from tasks to processors
Greedy scheduling policies

A *greedy scheduler* is a scheduler in which no processor is idle if there are ready tasks.

![Diagram of a greedy scheduler with nodes indicating executed, ready, waiting, and not yet created tasks. The number of processors $P = 2$.](image)
What’s good about the greedy scheduler

Recall: $T_P$ denotes execution time on $P$ processors

- Observation 1: $T_P \geq T_\infty$
  - can go no faster than length of critical path
- Observation 2: $T_P \geq \frac{T_1}{P}$
  - can go no faster than having all processors always busy
- Brent’s Theorem: $T_P \leq \frac{T_1}{P} + T_\infty$
- Theorem says that we can get close to optimal execution time (within factor of two).
When to expect linear speedups

Recall:

▷ $P_{avg} = \frac{T_1}{T_\infty}$.
▷ Brent’s Theorem: $T_P \leq \frac{T_1}{P} + T_\infty$

Therefore:

▷ Suppose that $P_{avg} \gg P \iff \frac{T_1}{T_\infty} \gg P \iff \frac{T_1}{P} \gg T_\infty$.
  ▷ This case is often called “parallel slackness”.
▷ With parallel slackness, the first term in Brent’s Theorem dominates.
▷ So, we have $T_P \approx \frac{T_1}{P}$.
  ▷ i.e., linear speedup
▷ Observation: This prediction is valid for our model, where scheduling costs are not reflected.
Designing a greedy scheduling policy

First idea: maintain ready tasks in a shared queue.

- When a processor needs a new task, it grabs one from the queue.
- When a processor forks a task, it puts the task on the shared queue.
Problem with the shared queue

- Benefits of parallelism are obliterated because processors spend a lot of time waiting to access the queue.
Work stealing

- Each processor maintains the ready tasks that it has created in what is called a deque.
- Processors usually push and pop on their own deques.
- If a given processor’s deque is empty, then the processor pops from the non-empty deque of another processor (if any).
  - called “stealing”
- There is an extensive literature on work stealing.
Work stealing deques

processors

deques

$P_1$ $P_2$ $P_3$

push pop push pop

deques

Arthur Charguéraud and Mike Rainey
Work stealing steal

processors

$P_1$

push pop

deques

$P_2$

push pop

$P_3$

push pop
Example of work stealing

number of processors $P = 2$
Example of work stealing

number of processors $P = 2$

Arthur Charguéraud and Mike Rainey
Example of work stealing

\[ P_1 \]

number of processors \( P = 2 \)

Arthur Charguéraud and Mike Rainey
Example of work stealing

number of processors $P = 2$

Arthur Charguéraud and Mike Rainey
Example of work stealing

\[ P_1 \quad P_2 \]

number of processors \( P = 2 \)

executing
executed
ready
waiting

Arthur Charguéraud and Mike Rainey
Example of work stealing

Number of processors $P = 2$

$P_1$ $P_2$

- executing
- executed
- ready
- waiting

Arthur Charguéraud and Mike Rainey
Example of work stealing

number of processors $P = 2$

Arthur Charguéraud and Mike Rainey
Example of work stealing

number of processors $P = 2$
Example of work stealing

number of processors $P = 2$
Example of work stealing

number of processors $P = 2$
Example of work stealing: handling join tasks

number of processors $P = 2$
Example of work stealing: handling join tasks

number of processors $P = 2$

$P_1$ gets to the join task first
Example of work stealing: handling join tasks

$P_1$ goes idle
(starts trying to steal)

number of processors $P = 2$
Example of work stealing: handling join tasks

number of processors $P = 2$

$P_2$ executes the continuation

$P_2$ executes

executing

executed

ready

waiting

Arthur Charguéraud and Mike Rainey
Example of work stealing: handling join tasks

number of processors \( P = 2 \)

\( P_2 \) executes the continuation

number of processors \( P = 2 \)
Example of work stealing: handling join tasks

number of processors $P = 2$

Arthur Charguéraud and Mike Rainey
The benefit of work stealing

Recall:
- Brent’s theorem $T_P \leq \frac{T_1}{P} + T_\infty$
- When $\frac{T_1}{T_\infty} \gg P$, we can expect linear speedup: $T_P \approx \frac{T_1}{P}$.

But!
- We have to assume that ready tasks can be found efficiently.
- Work stealing achieves this because most of the time the ready task is in the local deque.
- Rarely does a processor have to steal.
  - Suppose we have the thief processor always pick its victim uniformly at random.
  - Blumofe and Leiserson (1995) show that, with high probability, expected total # of steals $\leq O(T_\infty P)$.
  - So, we want $T_\infty P \ll T_1$, which is equivalent to $\frac{T_1}{T_\infty} \gg P$. 

Arthur Charguéraud and Mike Rainey
The work-first principle for designing efficient implementations of work stealing

\[
\text{scheduling costs} = \text{stealing costs} + \text{non stealing costs} \\
\quad \text{(non local)} \quad \text{(local)} \\
\quad \text{(rare)} \quad \text{(common)}
\]

Work-first principle (Frigo et al. 1998): minimize the second term in the sum above because it represents the common case.
Slow clone (represents rare case)

let \((x_1, x_2) = (|t_1, t_2|)\) in
... (* continuation \(k\) *)

\((t_2\text{ was stolen})\)
Slow clone (represents rare case)

let \((x_1, x_2) = (|t_1, t_2|)\) in
...
(* continuation \(k\) *)

expensive atomic add to prevent race
let \((x_1, x_2) = (|t_1, t_2|)\) in
...
(* continuation \(k\) *)

expensive atomic
add to prevent race
let \((x_1, x_2) = (|t_1, t_2|)\) in

... (* continuation \(k\) *)

start evaluating

\(k(x_1, x_2)\)
let \((x_1, x_2) = (|t_1, t_2|)\) in
... (* continuation \(k\) *)
Fast clone (represents common case)

let \((x_1, x_2) = (|t_1, t_2|)\) in

... (* continuation \(k\) *)
Fast clone (represents common case)

```latex
let (x_1, x_2) = (|t_1, t_2|) in
... (* continuation k *)

start evaluating $t_2$:

(fun x_2 -> k (x_1, x_2)) t_2
```
Making deques more efficient than using a lock

- There is a potential race in stealing because both thief and victim can try to pop from deque simultaneously.
- Using locks would be too expensive.
- There are some better approaches:
  - Private deques
  - Shared deques
Private deques

- Each processor has sole read / write access to its own deque.
  - Stealing is handled by message passing.
  - Designs investigated in Multilisp, ADM, and Manticore.
- Local deque access is cheap.
- Protecting deques from races is trivial: just delay handling a message while deque is in inconsistent state.
- Message-passing can be implemented on top of software polling or OS interrupts.
- In any case, busy processors always pay a cost for handling signals.
- We need to avoid the case where busy processors are sent too many messages.
- We can avoid sending unnecessary messages by having each processor maintain a flag indicating if its deque is empty.
Shared deques

- Each processor has exclusive read / write access to the top of its own deque; all processors have read / write access to the bottom of the deque.
  - Synchronization is handled by Dijkstra-style mutual exclusion protocol (Frigo et. al. 1998).
  - Designs investigated in Multilisp, Cilk, and Hood.
- Non-blocking deques are crucial in the setting where processors are shared between work stealing and other processes.
  - Several papers investigate non-blocking deques.
    - Blumofe et. al. 1998
    - Nir & Shavit 2005
    - Tang et. al. 2010
- Presentations of concurrent deque algorithms assume sequential consistency.
  - Modern multicore machines usually have relaxed memory consistency models.
  - For such machines, expensive memory fences are required to prevent race condition.
Comparing shared and private deques

- In private-deque approach, we can easily avoid costly memory fences, whereas we cannot in existing shared-deque approaches.
- Stealing is more expensive with with private deques because of message-passing overhead.
  - Stealing costs are arguably of minor importance, because we expect the common case is that # steals is negligible.
- Handling deque overflow is trivial with private deques; special concurrency protocol is necessary for shared deques (Nir & Shavit 2005).
Summary

- We introduced SP DAGs to model performance.
- We compared the parallelism of two tree-based and one list-based algorithms.
- We found that the tree-based algorithms naturally exhibit more parallelism than list-based ones.
- We studied the class of greedy schedulers and found that:
  - Execution time $T_P \leq \frac{T_1}{P} + T_\infty$
  - When $P_{avg} \gg P$, we achieve linear speedup.
- In practice, work stealing gets close to the bound above because it minimizes costs of managing ready tasks in common case.
Parallel merge

Find $p$ by binary search.

Key fact: if the number of elements in both arrays is $n$, then the number of elements in the larger of the two recursive merges is no greater than $\frac{3}{4}n$. 

Arthur Charguéraud and Mike Rainey