CS 598: Communication Cost Analysis of Algorithms

Lecture 11: Communication lower bounds for the FFT and sorting algorithms

Edgar Solomonik

University of Illinois at Urbana-Champaign

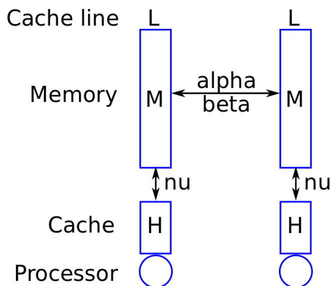
September 28, 2016

Horizontal and vertical communication cost

Lets review terminology, before we move forward with analysis:

- **interprocessor communication** - messages between processors executing in parallel: α, β
- **cache complexity** - loads and stores during sequential execution (memory bandwidth cost): ν

Below are two processors with M memory capacity and H cache size



(nu=ν)

Communication lower bounds

Communication lower bounds provide us with insight regarding algorithms and problems

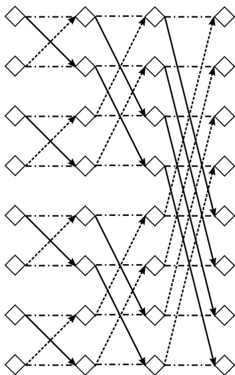
- provide optimality criterion
- entail rigorous definition of problem or space of algorithms
- achievable lower bounds typically carry algorithmic intuition
- most often obtained by upper-bounds on amount of useful work that can be done with a limited set of data
- often but not always, this strategy does not work for algorithms which have linear cost

Vertical and horizontal communication lower bounds

Given an upper bound $\mathcal{E}(s)$ on the amount of algorithmic operations that can be done with s data (inputs)

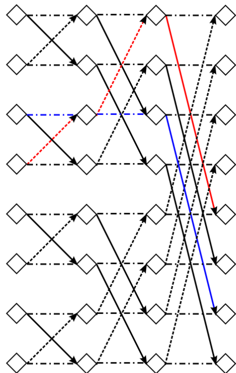
- we can derive lower bounds on cache complexity, since only $\mathcal{E}(2H)$ work can be done for every H inputs loaded to cache
- (for matrix multiplication, it was necessary to also consider outputs, $f_{\text{MM}}(s) = s^{3/2}$)
- for the FFT, unlike matrix multiplication, we can ignore outputs
- $\mathcal{E}$ yields lower bounds on interprocessor communication, since only $\mathcal{E}(N/P)$ work can be done without communication by each processor if the algorithm has N inputs
  - if algorithm requires total Z operations, communication necessary if $Z/P > \mathcal{E}(N/P)$
  - then need to communicate W data such that $\mathcal{E}(W) \geq Z/P - \mathcal{E}(N/P)$
  - tighter bounds sometimes attained by considering memory capacity M, $W = \Omega(M \cdot Z/P(\mathcal{E}(M)) - \mathcal{E}(N/P))$

Radix-2 FFT dependency graph



Paths in Radix-2 FFT dependency graph

Any two edge-disjoint paths in the FFT DAG intersect at no more than one vertex



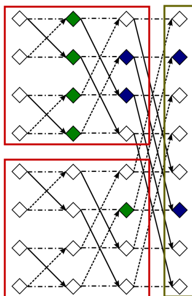
in other words, the FFT DAG has no cycles

Work bound for FFT

We prove that the work bound for the radix-2 FFT is $\mathcal{E}_{\mathsf{FFT}}(s) = s\log_2 s$

- in particular that with s inputs, at most $s\log_2 s$ work can be done
- we can do this by induction on s
- the base case, $s = 1$ holds trivially
- assume we have shown the bound for $s - 1$ inputs

Work bound for FFT, contd



- consider the last level in the FFT graph in which a vertex is computed
- if k vertices in the level were computed, we must know $k/2$ values in each of the left sub-FFTs
- moreover, each sub-FFT must have at least $k/2$ inputs
- conversely, if one of the sub-FFTs had t of the s inputs, we can at most $2\min(s-t,t)$ vertices at the last level may be computed
- $\mathcal{E}_{\mathrm{FFT}}(s) = \max_t(\mathcal{E}_{\mathrm{FFT}}(s-t) + \mathcal{E}_{\mathrm{FFT}}(t) + 2\min(s-t,t))$

Communication lower bound for the FFT

By induction the expression
$\mathcal{E}_{\mathsf{FFT}}(s) = \max_t(\mathcal{E}_{\mathsf{FFT}}(s-t) + \mathcal{E}_{\mathsf{FFT}}(t) + 2\min(s-t,t))$ implies

$$\mathcal{E}_{\mathsf{FFT}}(s) = \max_t((s-t)\log_2(s-t) + t\log(t) + 2\min(s-t,t))$$

which is maximized by picking $t = s/2$

$$\mathcal{E}_{\mathsf{FFT}}(s) = 2\mathcal{E}_{\mathsf{FFT}}(s/2) + s = s\log_2(s)$$

Given $\mathcal{E}_{\mathsf{FFT}}(s) = s\log_2(s)$, the cache complexity is

$$Q_{\text{seq-FFT}}(n,H) \geq n\log_2(n)H/\mathcal{E}_{\mathsf{FFT}}(2H) = n\frac{\log(n)}{2\log(2H)} = \Omega(n\log_H(n))$$

We showed that a radix-$\sqrt{n}$ FFT algorithm gets this cost.
Your homework includes showing that a careful schedule for the radix-2 FFT algorithm can attain the bound

Communication lower bound for the FFT

Given $\mathcal{E}_{\text{FFT}}(s) = s\log_2(s)$, the interprocessor communication cost is

$$W_{\text{par-FFT}}(n,P) \geq \mathcal{E}_{\text{FFT}}^{-1}(n\log_2(n)/P - \mathcal{E}_{\text{FFT}}(n/P))$$

$$W_{\text{par-FFT}}(n,P) \geq \mathcal{E}_{\text{FFT}}^{-1}(n(\log_2(n) - \log_2(n/P))/P)$$

$$W_{\text{par-FFT}}(n,P) \geq \mathcal{E}_{\text{FFT}}^{-1}(n\log_2(P)/P)$$

When $P = \Theta(n/P)$, $W_{\text{par-FFT}}(n,P) = \Omega(n/P)$
The tighter lower bound

$$W_{\text{par-FFT}}(n,P) = \Omega\left(\frac{n\log_2(n)}{P\log_2(n/P)}\right)$$

has been shown by other methods [Bilardi, Scquizzato, Silvestri 2012] and previously for other models (LPRAM) [Aggarwal, Chandra, Snir 1990]

Short pause

Sorting and parallel sorting

We first consider variants of sorting for sequential/shared-memory

- given n keys or n key-value pairs, order them in memory contiguously so that the ith smallest key (pair) is in the ith location
- if there are equivalent keys, a *stable* sort is one that preserves their original ordering
- depending on the type of key, we can work with their bit representation or only perform comparison operations

Sorting in distributed memory entails further problem variants

- now need to consider the splitting of keys among processors
- generally, we impose an order on the set of processors and distribute the target buffer in chunks according to this order
- exact splitting implies each processor should end up with the same number of elements
- approximate splitting may be sufficient with some imbalance threshold

Sorting algorithms

Most sorting algorithms can be classified as *merge-based* or *distribution-based*

- merge-based algorithms sort subsequences then merge them, e.g. mergesort, bitonic sort
  - sorting small subsequences is parallel and cache-efficient, but merging is challenging
- distribution-based algorithms partition the keys into buckets, then sort the buckets, e.g. quicksort, radix sort
  - sorting buckets is parallel and cache-efficient, but partitioning is challenging
- both merging and partitioning is easy when the keys are nicely-distributed/random
- as a result the dependency-graph and complexity of most sorting algorithms is input-dependent
- bitonic sort is a noteworthy exception, but has cost $O(n \log^2(n))$

Cache-complexity of mergesort

Mergesort sorts two subsequences of size $n/2$ recursively then merges

$$T_{\text{mergesort}}(n) = 2T_{\text{mergesort}}(n/2) + T_{\text{merge}}(n/2)$$

- a standard sequential merge has linear cache complexity $T_{\text{merge}}(s) = 2s \cdot \nu$ with any cache line size L
- since when $n \leq H$, no further cache transfers are required,

$$Q_{\text{seq-mergesort}}(n, H) = n\log_2(n/H) \cdot \nu$$

- this cache complexity is not bad and the algorithm is cache-oblivious, but for optimality we need

$$Q_{\text{sort}}(n, H) = \Theta(n\log_H(n) \cdot \nu)$$

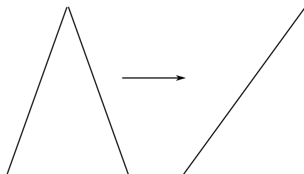
- furthermore, its not clear how to parallelize a standard merge
- Cole's mergesort (1988) parallelizes the merge by maintaining and combing "covering samplers" for each subsequence in mergesort

Sorting with optimal cache-complexity

Funnelsort: a cache-oblivious merge-based sort

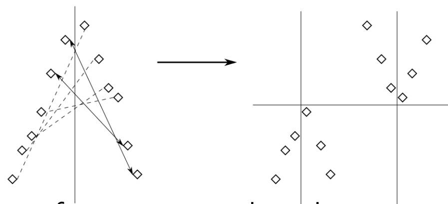
- introduced by Frigo, Leiserson, Prokop, and Ramachandran
- asymptotically optimal $Q_{\text{funnelsort}}(n, H) = \Theta(n \log_H(n) \cdot \nu)$
- partitions n keys into $n^{1/3}$ subsequences of length $n^{2/3}$ and sorts them recursively
- merges subsequences via a $k^{1/3} \times k^{2/3}$-recursively-defined merger
- the merger is complicated and not well-parallelizable
- some simplifications exist, e.g. "Lazy funnelsort", but still have polynomial depth

Bitonic sort



- bitonic sort recurses like mergesort, and uses a "bitonic merge" to combine subsequences
- a bitonic merge is itself recursive and costs $O(s \log_2 s)$ to merge to subsequences of size s
- the bitonic merge is typically defined with the second subsequence in reverse order from the first
- given a sequence like $(x_1 \leq \cdots \leq x_i \geq \cdots \geq x_{2s})$ or a shift of such a sequence, it produces an increasing sequence of size s

Bitonic merge



- the bitonic merge of two reverse order subsequences of size s works as follows
    - compare and swap the ith element in the first subsequence with the ith element in the second
    - the swaps result in two bitonic sequences, the second has elements greater than any of those in the other
    - perform two bitonic merges recursively to merge these subsequences
- input may be increasing then decreasing or decreasing then increasing, and we may want an increasing or decreasing output
- ‘increasing’ or ‘decreasing’ is a property of the buffer ordering, each step merges two ‘sorted’ subsequences

Bitonic sort

The second part of your homework is to analyze bitonic sort

Radix sort

Radix sort defines buckets based on bit subsequences of keys

- you should be familiar with it from the previous homework
- radix sort is typically defined to start with the least significant bits
- a key can transition between any pair of buckets for every subset of bits (in this sense, radix sort is not really a bucket sort)
- ordering within buckets must be preserved at the next step (exploits stability)
- Q: what would be different if we started with the most significant bits?
- A: then we would have a bucket sort and would need to sort each subsequence independently

Redistribution in radix sort

Bit operations allow us to determine which key belongs to which bucket

- the expensive part is moving the data to the appropriate destination
- the arbitrary rerouting is not cache-efficient
- in distributed memory, we have multiple ways of picking ‘destinations’
  - if we want to achieve an exact splitting, we need to assign each processor a set of destinations that is not aligned with bucket boundaries
  - for an approximate splitting it may suffice to assign buckets to processors, if there are a sufficient number of buckets and they are load balanced
- for an exact splitting we need to do a segmented scan, which bounds the number of buckets we can efficiently support (no more than n/P, so $\log_2(n/P)$ bits at a time)
- with an approximate splitting, we could potentially process more bits at a time, and move data less times

Quicksort

Quicksort allows us to partition arbitrary keys by selecting pivots

- if we select one pivot at a time, in the average case (or with a good pivot selection algorithm), we obtain

$$T_{\text{qsort}}(n) = 2T_{\text{qsort}}(n/2) + O(n)$$

  where the $O(n)$ work at each recursive step is more parallelizable than merging
- we can again use a segmented scan to determine the exact splitting/destinations
- however, we need to move data cross cache boundaries $\log_2(n/H)$ times
- in the parallel case we redistribute the data $\log_2(P)$ times
- we can increase the base of the logarithm by selecting more pivots (splitters)

Sample sort

The observation that a larger number of pivots reduces the number of redistributions motivates *splitter-based* sorting algorithms

- if we can find $n/H - 1$ or $P - 1$ splitters that partition the data evenly, it suffices to redistribute the data once and we can proceed with no communication
- sample sort algorithms select such splitters by collecting a sample and sorting it
- random sample sort: we can find $r - 1$ splitters by collecting a random sample of size r^2 sorting it and selecting every rth element
- regular sample sort: we can do better, by first sorting subsequences of n/r elements, selecting $r - 1$ splitters from each subsequence, sorting the $r(r - 1)$ total splitters and selecting a new $r - 1$ splitters
- regular sample sorting guarantees that no splitter interval will contain more than $2n/r$ elements
- in practice a random sample of size $\Theta(r \log r)$ might already obtain a very good splitting

Lowering the cost of finding splitters

Sorting a sample of size $\Theta(P^2)$ sequentially may be expensive on large distributed systems

- Q: how could we sort this sample more efficiently?
- A: apply sample sort recursively with all or a subset of processors
- an alternative technique to finding splitters is Histogram sort
- determines splitters iteratively by probing and computing histograms
- if any interval is imbalanced, adjust the splitters

Histogram sort

Histogram sort can work well in practice

- given a probe (set of splitter-guesses), computing histogram (how many elements fall between each pair of splitters) can be done by
  - bucketing unsorted subsequences
  - sorting subsequences and performing merging with probe
  - sorting subsequences and performing a binary sort with every element of the probe
- the probe can be initialized by a small random sample
- downside is that adjustment may take many iterations if the distribution is very unbalanced or there are many repeated keys

Splitter-based algorithms discussion

There are a plethora of parallel sorting algorithm variants

- overall the advantage of splitter-based algorithms in distributed-memory is that the full set of key-value pairs is redistributed only once
- to design cache-oblivious and/or shared memory algorithms, splitting may be done recursively and combined with other techniques
- it is less clear at this point what the best cache-efficient (parallel) sorting algorithms are than the best distributed-memory