CS 598: Communication Cost Analysis of Algorithms

Lecture 10: FFT algorithms and an introduction to communication lower bounds

Edgar Solomonik

University of Illinois at Urbana-Champaign

September 26, 2016

DFT matrix and convolutions

For any n, let $\omega_n = e^{-2\pi i/n}$, a DFT matrix of dimension n is given by

$$\forall j, k \in [0, n-1] \quad D_n(j, k) = \omega_n^{jk}$$

for example $D_4 = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & \omega & \omega^2 & \omega^3 \\ 1 & \omega^2 & \omega^4 & \omega^6 \\ 1 & \omega^3 & \omega^6 & \omega^9 \end{bmatrix}$

A convolution takes as input vectors a and b and computes vector c

$$\forall k \in [0, n-1] \quad c(k) = \sum_{j=0}^{k} a(j)b(k-j)$$

It can be computed via the DFT

$$c = D_n^{-1}[(D_n a) \odot (D_n b)]$$

where $\odot$ is an elementwise product

Radix-2 Fast Fourier Transform (FFT)

We now look at how to apply the DFT via the FFT algorithm

- intuitively, we can expect to compute the DFT quickly since D_n is so nicely structured, a single root of unity parameter ω_n can be used to represent it
- consider $b = D_n a$, we have

$$\forall j \in [0, n-1] \quad b(j) = \sum_{k=0}^{n-1} \omega_n^{jk} a(k)$$

- our goal is to find a recursive algorithm, that expresses the DFT as two DFTs of dimension $n/2$, with a different root of unity $\omega_{n/2}$
- $\omega_{n/2} = \omega_n^2$, so we separate the summands into odds and evens

$$\begin{aligned}\forall j \in [0, n-1] \quad b(j) &= \sum_{k=0}^{n/2-1} \omega_n^{j(2k)} a_{2k} + \sum_{k=0}^{n/2-1} \omega_n^{j(2k+1)} a(2k+1) \\ &= \sum_{k=0}^{n/2-1} \omega_{n/2}^{jk} a_{2k} + \omega_n^{j} \sum_{k=0}^{n/2-1} \omega_{n/2}^{jk} a(2k+1)\end{aligned}$$

Radix-2 Fast Fourier Transform (FFT), contd.

We can note that, given

$$\forall j \in [0, n-1] \quad b(j) = \sum_{k=0}^{n/2-1} \omega_{n/2}^{jk} a(2k) + \omega_n^j \sum_{k=0}^{n/2-1} \omega_{n/2}^{jk} a(2k+1)$$

the summations for $b(j)$ and $b(j+n/2)$ are closely related, $\forall j \in [0, n/2-1]$

$$b(j+n/2) = \sum_{k=0}^{n/2-1} \omega_{n/2}^{(j+n/2)k} a(2k) + \omega_n^{j+n/2} \sum_{k=0}^{n/2-1} \omega_{n/2}^{(j+n/2)k} a(2k+1)$$

we now note $\omega_{n/2}^{(j+n/2)k} = \omega_{n/2}^{jk}$ since $(\omega_{n/2}^{n/2})^k = 1^k = 1$, so

$$\forall j \in [0, n/2-1] \quad b(j+n/2) = \sum_{k=0}^{n/2-1} \omega_{n/2}^{jk} a(2k) - \omega_n^j \sum_{k=0}^{n/2-1} \omega_{n/2}^{jk} a(2k+1)$$

where we additionally use $\omega_n^{n/2} = -1$.

Radix-2 Fast Fourier Transform (FFT), contd.

Each of these two summation can be done recursively with an FFT

- lets vectors u and v be these two FFTs

$$\forall j \in [0, n/2-1] \quad u(j) = \sum_{k=0}^{n/2-1} \omega_{n/2}^{(j+n/2)k} a(2k)$$

$$\forall j \in [0, n/2-1] \quad v(j) = \sum_{k=0}^{n/2-1} \omega_{n/2}^{(j+n/2)k} a(2k+1)$$

- we can make these two recursive calls simultaneously and without any work
- we then scale using "twiddle factors" $z(j) = v(j) \cdot \omega_n^j$
- it then suffices to combine the vectors as follows

$$b = \begin{bmatrix} u+z \\ u-z \end{bmatrix}$$

- notice that the way we combine them can be seen as an FFT of dimension 2

$$b = \begin{bmatrix} b_1 \\ b_2 \end{bmatrix} = \text{vec}\left(\begin{bmatrix} b_1 & b_2 \end{bmatrix}\right) = \text{vec}\left(\begin{bmatrix} u & z \end{bmatrix} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}\right) = \text{vec}\left(\begin{bmatrix} u & z \end{bmatrix} D_2\right)$$

Cache complexity of radix-2 FFT

We can now analyze the cache complexity of this FFT algorithm

- lets consider γ to be the cost per operation, and ν to be the inverse memory bandwidth
- at every recursive level we have a linear cost of applying twiddle factors, yielding the recurrence

$$T_{\mathsf{FFT2}}(n, H) = 2T_{\mathsf{FFT2}}(n/2, H) + O(n \cdot \nu + n \cdot \gamma)$$

- once the problem fits in cache (size H), we incur no more bandwidth cost

$$T_{\mathsf{FFT2}}(n < H, H) = 2T_{\mathsf{FFT2}}(n/2, H) + O(n \cdot \gamma) = O(n \log(n) \cdot \gamma)$$

- therefore the total cost (assuming $n > H$) is

$$T_{\mathsf{FFT2}}(n, H) = O(n \log(n/H) \cdot \nu + n \log n \cdot \gamma)$$

- for $n \gg H$, this is flop to byte ratio approaches 1

Lowering the cost of twiddle factors

We can subdivide an FFT not just into two FFTs, but into many, then combine the result, with... more FFTs!

- consider any factorization $n_1 n_2 = n$
- we can subdivide the FFT into n_1 FFTs of dimension n_2 then combine them with n_2 FFTs of dimension n_1 as follows

$$c(i_2 n_1 + i_1) = \sum_{j_1=0}^{n_1} \omega_{n_1}^{i_1 j_1} \left[\left(\sum_{j_2=0}^{n_2} \omega_{n_2}^{i_2 j_2} a(j_1 n_2 + j_2) \right) \omega_n^{i_1 j_2} \right]$$

- essentially we have separated the columns of the DFT matrix with stride n_1 and expressed the sum in terms of the root of unity $\omega_{n/n_1} = \omega_{n_2}$
- the factors $\omega_n^{i_1 j_2}$ correspond to the twiddle factors by which we multiplied the FFT of the odd subsequence of a in the radix-2 algorithm

Correctness of Radix-n_1 FFT

Lets see why this equation is true

$$c(i_2 n_1 + i_1) = \sum_{j_1=0}^{n_1} \omega_{n_1}^{i_1 j_1} \left[\left(\sum_{j_2=0}^{n_2} \omega_{n_2}^{i_2 j_2} a(j_1 n_2 + j_2) \right) \omega_n^{i_1 j_2} \right]$$

we can show correctness by pushing the summations to the back

$$\begin{aligned} c(i_2 n_1 + i_1) &= \sum_{j_1=0}^{n_1} \sum_{j_2=0}^{n_2} \omega_{n_1}^{i_1 j_1} \omega_n^{i_1 j_2} \omega_{n_2}^{i_2 j_2} a(j_1 n_2 + j_2) \\ &= \sum_{j_1=0}^{n_1} \sum_{j_2=0}^{n_2} \omega_n^{i_1 j_1 n_2} \omega_n^{i_1 j_2} \omega_n^{i_2 j_2 n_1} a(j_1 n_2 + j_2) \\ &= \sum_{j_1=0}^{n_1} \sum_{j_2=0}^{n_2} \omega_n^{i_1 j_1 n_2 + i_1 j_2 + i_2 j_2 n_1} a(j_1 n_2 + j_2) \\ &= \sum_{j_1=0}^{n_1} \sum_{j_2=0}^{n_2} \omega_n^{(i_2 n_1 + i_1)(j_1 n_2 + j_2)} a(j_1 n_2 + j_2) \end{aligned}$$

Q: why is an extra factor of $\omega^{i_2 n_1 j_1 n_2}$ not a problem?

Recursive structure of Radix-n_1 FFT

Lets see how we can apply this equation

$$c(i_2 n_1 + i_1) = \sum_{j_1=0}^{n_1} \omega_{n_1}^{i_1 j_1} \left[\left(\sum_{j_2=0}^{n_2} \omega_{n_2}^{i_2 j_2} a(j_1 n_2 + j_2) \right) \omega_n^{i_1 j_2} \right]$$

- first lets decompose a into subvectors of length n_2, $a = \begin{bmatrix} a_1 \\ \vdots \\ a_{n_1} \end{bmatrix}$
- then we apply the FFT recursively to each of them, obtaining $v_{i_1} = D_{n_2} a_{i_1}$
- then we apply the twiddle factors to every element $u_{i_1}(j_2) = v_{i_1}(j_2)\omega_n^{i_1 j_2}$
- then we apply the FFT recursively on different subvectors

$$\begin{bmatrix} c_1 \\ \vdots \\ c_{n_1} \end{bmatrix} = \text{vec}\left(\begin{bmatrix} u_1 & \cdots & u_{n_1} \end{bmatrix} D_{n_1} \right)$$

- Q: sanity check, D_{n_1} is symmetric, so do we compute $D_{n_1} u_{i_1}$ recursively?
- A: no, we do $v_{i_2} D_{n_1} = (D_{n_1} v_{i_2}^T)^T$ where $\begin{bmatrix} v_1 & \cdots & v_{n_2} \end{bmatrix}^T = \begin{bmatrix} u_1 & \cdots & u_{n_1} \end{bmatrix}$

Cache oblivious FFT

We can get a cache-oblivious FFT algorithm by choosing $n_1 = n_2 = \sqrt{n}$

- we now get a recurrence

$$T_{\mathrm{FFT}}(n,H) = 2\sqrt{n}T_{\mathrm{FFT}}(\sqrt{n}) + O(n\cdot\gamma + n\cdot\nu)$$

- once $n < H$, we incur no more bandwidth cost, we get to this after $\log_H(n)$ recursive calls, obtaining a total cost of

$$T_{\mathrm{FFT}}(n,H) = O(n\log_H(n)\cdot\nu + n\log(n)\cdot\gamma)$$

- this improves over the radix-2 case, since

$$\log_H(n) = \log_2(n)/\log_2(H) \le \log_2(n/H) = \log_2(n) - \log_2(H)$$

FFT in BSP

Lets assume $n \geq P^2$, and again do radix-$\sqrt{n}$ FFT

- the assumption $n \geq P^2$ is similar to what our allgather algorithms assumed (each processor starts with $\geq P$ different elements)
- each processor computes $\sqrt{n}/P$ FFTs of dimension $\sqrt{n}$ with their local data
- the data is transposed (all-to-all)
- each processor computes $\sqrt{n}/P$ FFTs of dimension $\sqrt{n}$ with their local data
- $T_{\text{FFT}}^{\text{BSP}}(n, P) = \alpha + n/P \cdot \beta$
- Q: could we achieve the same cost if we allow only point-to-point messages?
- A: no, all-to-all has cost $T_{\text{FFT}}^{\alpha-\beta}(n, P) = \alpha \cdot \log_2(P) + n \log_2(P)/P \cdot \beta$ or $T_{\text{FFT}}^{\alpha-\beta}(n, P) = \alpha \cdot (P-1) + n/P \cdot \beta$

Short pause

Introduction to communication lower bounds

A brief history of pioneering work

- Floyd 1972: for large cache lines $L = \Theta(H)$, matrix transposition has cost $O(n^2 \log(n) \cdot \beta)$
- Jiawei and Kung 1981, pebbling lower bound
    - model communication as placing pebbles on a dependency graph of an algorithm
    - work with $L = 1$ (only consider H)
    - lower bounds for matrix-matrix multiplication, FFT, stencil computation, odd-even sort
- Aggarwal and Vitter 1988, lower bounds with any L, H
    - communication lower bounds for general permutation networks
    - lower bounds for transposition, FFT, and comparison-based sorting

Lower bounds by partitioning memory operations

Pebbling bounds employ the following general argument

- consider the sequence of loads and stores (memory-cache) transfers computed by a program
- the length of the sequence is the bandwidth cost Q
- partition the sequence into parts of size H
- upper-bound the amount of useful work that can be done between the beginning and end of this sequence
- H bounds the number of inputs we read from memory and outputs we write to cache
- Q: how many other inputs are available during the execution of this sequence?
- A: at the beginning of the sequence we have up to H inputs in cache, and at the end up to H outputs
- with partitioning, all we need is a bound $f_{\text{alg}}(H)$ on how much useful computation can be done with $3H$ inputs + outputs
- if the total amount of computation is F, $Q \geq FH/f_{\text{alg}}(H)$

Lower bounds by partitioning computation

We can also take the dual view

- we are given an algorithm that must perform F operations
- we need to prove that the given $3H$ inputs and outputs at most $f_{\text{alg}}(H)$ of the computation can be done
  - to prove this we generally need some assumptions to guarantee that outputs cannot be discarded
  - its typical to assume that the F operations are not recomputed (outputs are not regenerated)
  - we can also represent some algorithms with dependency graphs (DAGs) with F vertices
- consider any execution schedule (ordering) of the F operations
- for each subsequence of size $f_{\text{alg}}(H)$, we can show that H loads or stores are required
- we then get the desired bound $Q \geq FH/f_{\text{alg}}(H)$

Bounding work in matrix multiplication

Consider the $F = n^3$ products computed in square matrix multiplication

- additions are tricky, we don't want to impose specific summation trees
- consider any G of the products $C(i,j) \leftarrow A(i,k) \cdot B(k,j)$
- the $d = 3$ Loomis-Whitney theorem tells us that the number of unique (i,k), (k,j), and (i,j) indices in G: g_A, g_B, and g_C, satisfy

$$\sqrt{g_A \cdot g_B \cdot g_C} \geq G$$

- in other words, the inputs needed to compute the G entries include g_A values of A, g_B values of B, and they contribute to g_C different entries of C
- we can safely restrict the space of algorithms to those that do not sum products which contribute to different entries of C
- bound the size of G provided the number of inputs and outputs is at most H

$$f_{\mathrm{MM}}(H) = \max_{|g_A+g_B+g_C| \leq 3H} \sqrt{g_A \cdot g_B \cdot g_C} = H^{3/2}$$

Cache complexity lower bound for MM

Given $f_{\text{MM}}(H) = H^{3/2}$, we are essentially done

- we obtain the sequential memory bandwidth lower bound

$$Q_{\text{seq-MM}}(n, H) \geq n^3 H / f_{\text{MM}}(H) = \frac{n^3}{\sqrt{H}}$$

- in the parallel case, one of P processors needs to perform n^3 of the products, so

$$Q_{\text{par-MM}}(n, H, P) \geq \frac{n^3}{P\sqrt{H}}$$

Interprocessor communication lower bound for MM

We can also use f_{MM} to get lower bounds on interprocessor communication

- given that each processor has M memory, $f_{\text{MM}}(M)$ tells us how much computation can be done with M inputs/outputs
- we can assume no processor has more than $2n^2/P$ inputs at the start of execution and n^2/P outputs at the end, so

$$W_{\text{par-MM}}(n, H, M, P) \geq n^3 M / f_{\text{MM}}(M) - 3n^2/P = \frac{n^3}{P\sqrt{M}} - 3n^2/P$$

- for $c \in [1, P^{1/3}]$ we get

$$W_{\text{par-MM}}(n, H, cn^2/P, P) = \Omega\left(\frac{n^2}{\sqrt{cP}}\right)$$

- restricting the amount of work done to n^3/P, gets us

$$W_{\text{par-MM}}(n, H, P) = \Omega\left(\frac{n^2}{P^{2/3}}\right)$$