CS 598: Provably Efficient Algorithms for Numerical and Combinatorial Problems

Part 4: Communication Cost in Algorithms

Edgar Solomonik

University of Illinois at Urbana-Champaign

Algorithmic cache management

Consider a computer with unlimited memory and a cache of size H

- we can design algorithms by manually managing cache transfers
  - *minimize amount of data moved from memory to cache (bandwidth cost)*
  - *minimize number of synchronous memory-to-cache transfers (latency cost)*
- generally, efficient algorithms in this model try to select blocks of computation that minimize the surface-to-volume ratio
  - *i.e., do as much computation with the cache-resident data as possible*
  - *in other words, exploit temporal and spatial locality*

Cache-efficient matrix multiplication

Consider multiplication of $n \times n$ matrices $\boldsymbol{C} = \boldsymbol{A} \cdot \boldsymbol{B}$

For $i \in [1, n/s], j \in [1, n/t]$, $k \in [1, n/v]$, define blocks $\boldsymbol{C}[i,j]$, $\boldsymbol{A}[i,k]$, $\boldsymbol{B}[k,j]$ with dimensions $s \times t$, $s \times v$, and $v \times t$, respectively

```
for (i = 1 to n/s)
  for (j = 1 to n/t)
    initialize C[i,j] = 0 in cache
    for (k = 1 to n/v)
      load A[i,k] into cache
      load B[k,j] into cache
      C[i,j] = C[i,j] + A[i,k]*B[k,j]
    end
    write C[i,j] to memory
  end
end
```

Memory-bandwidth analysis of matrix multiplication

- Lets consider bandwidth and latency cost if each matrix multiplication has dimensions s, t, v
  - *there are a total of* $(n/s)(n/t)(n/v)$ *inner loop iterations*
  - *the memory latency cost of the algorithm is the number of inner loop iterations,* $O(n^3/(stv))$
  - *since each block of* $\boldsymbol{C}$ *stays resident in the innermost loop, we write each element of* $\boldsymbol{C}$ *to memory only once*
  - *we read each block* $s \times v$ *block of* $\boldsymbol{A}$ *and* $v \times t$ *block of* $\boldsymbol{B}$ *in each innermost loop*
  - *therefore, the bandwidth cost is* $Q = n^2 + (n/s + n/t)n^2 = n^2 + n^3/s + n^3/t$
- Given the constraint, $st + sv + vt \leq H$, we can derive the optimal block sizes
  - *if we pick* $s = t = v = \sqrt{H/3}$, *we satisfy the constraint and obtain* $Q \approx 2n^3/\sqrt{H/3}$, *with* $n^3/H^{3/2}$ *memory latency cost*
  - *if we pick* $s = t = \sqrt{H - 2\sqrt{H}}$ *and* $v = 1$, *we obtain* $Q \approx 2n^3/\sqrt{H}$ *with* n^3/H *memory latency cost*

Ideal cache model

- A more accurate model is to consider a cache line size L in addition to the cache size H
  - *each memory-to-cache transfer has size* L
  - *new unified metric: cache misses (number of cache lines transferred)*
  - *the bandwidth cost is the number of cache misses multiplied by* L
  - *the (old) latency cost (number of transfers) is disregarded*
  - *assume 'tall' cache,* $L \leq \sqrt{H}$ *(more convenient,* $H = \Omega(L^2)$*)*
- We can now consider different caching protocols
  - *an ideal cache model corresponds to the assumption that the protocol always makes the best decision*
  - *this ideal cache model is in a sense equivalent to a manually orchestrated cache protocol*
  - *arbitrary manual orchestration can be achieved with an LRU (lest-recently-used protocol)*

Matrix transposition in the ideal cache model

- Matrix multiplication bandwidth cost with a tall cache is not affected by L
  - *if we read square blocks into cache they have dimension* $\Theta(L)$
  - *if we compute outer products, just need to transpose* $\boldsymbol{B}$ *initially*
- $n \times n$ matrix transposition becomes non-trivial
  - *when* $L = 1$ *(original model), there is no notion of how a matrix is laid out in memory*
  - *for general* L*, we should read* $\sqrt{H} \times \sqrt{H}$ *blocks into cache, transpose them, then write them to memory to get linear bandwidth cost* $O(n^2)$
  - *matrix transposition is a very useful subroutine when we need to ensure contiguous access to cache lines*

Cache obliviousness

- Introduced by Frigo, Leiserson, Prokop, Ramachadran
  - *basic idea: algorithms should not be parameterized by architectural parameters*
  - *good ideas in computer science are most often good abstractions*
  - *designing an algorithm obliviously of cache size makes it portable and efficient for all levels of a cache hierarchy*
- cache oblivious algorithms are stated without explicit control of data movement
  - *their communication cost is derived by assuming an ideal cache model*
  - *ideal caches can be simulated by an LRU cache protocol for most (regular) algorithms*

Cache oblivious matrix transposition

Given $m \times n$ matrix $\boldsymbol{A}$, compute $\boldsymbol{B} = \boldsymbol{A}^T$

- *if* $m \leq n$ *subdivide* $\boldsymbol{A} = [\boldsymbol{A}_1 \ \boldsymbol{A}_2]$ *and* $\boldsymbol{B} = \begin{bmatrix} \boldsymbol{B}_1 \\ \boldsymbol{B}_2 \end{bmatrix}$ *and compute recursively,* $\boldsymbol{B}_1 = \boldsymbol{A}_1^T$, $\boldsymbol{B}_2 = \boldsymbol{A}_2^T$
- *if* $m > n$ *subdivide* $\boldsymbol{A} = \begin{bmatrix} \boldsymbol{A}_1 \\ \boldsymbol{A}_2 \end{bmatrix}$ *and* $\boldsymbol{B} = [\boldsymbol{B}_1 \ \boldsymbol{B}_2]$ *and compute recursively,* $\boldsymbol{B}_1 = \boldsymbol{A}_1^T$, $\boldsymbol{B}_2 = \boldsymbol{A}_2^T$

obtains linear bandwidth cost $T(mn) = 2T(mn/2)$, $T(1) = O(1)$, *so* $T(mn) = O(mn)$

Cache oblivious matrix multiplication

Given $m \times k$ matrix $\boldsymbol{A}$ and $k \times n$ matrix $\boldsymbol{B}$, compute $m \times n$ matrix $\boldsymbol{C} = \boldsymbol{A}\boldsymbol{B}$

- *if* $k \geq m$ *and* $k \geq m$ *subdivide* $\boldsymbol{A} = \begin{bmatrix} \boldsymbol{A}_1 & \boldsymbol{A}_2 \end{bmatrix}$ *and* $\boldsymbol{B} = \begin{bmatrix} \boldsymbol{B}_1 \\ \boldsymbol{B}_2 \end{bmatrix}$ *and compute recursively,* $\bar{\boldsymbol{C}} = \boldsymbol{A}_1\boldsymbol{B}_1$, $\hat{\boldsymbol{C}} = \boldsymbol{A}_2\boldsymbol{B}_2$, *then* $\boldsymbol{C} = \bar{\boldsymbol{C}} + \hat{\boldsymbol{C}}$
- *if* $n > k$ *and* $n \geq m$ *subdivide* $\boldsymbol{C} = \begin{bmatrix} \boldsymbol{C}_1 & \boldsymbol{C}_2 \end{bmatrix}$ *and* $\boldsymbol{B} = \begin{bmatrix} \boldsymbol{B}_1 & \boldsymbol{B}_2 \end{bmatrix}$ *and compute recursively,* $\boldsymbol{C}_1 = \boldsymbol{A}\boldsymbol{B}_1$, $\boldsymbol{C}_2 = \boldsymbol{A}\boldsymbol{B}_2$
- *if* $m > k$ *and* $m > n$ *subdivide* $\boldsymbol{C} = \begin{bmatrix} \boldsymbol{C}_1 \\ \boldsymbol{C}_2 \end{bmatrix}$ *and* $\boldsymbol{A} = \begin{bmatrix} \boldsymbol{A}_1 \\ \boldsymbol{A}_2 \end{bmatrix}$ *and compute recursively,* $\boldsymbol{C}_1 = \boldsymbol{A}_1\boldsymbol{B}$, $\boldsymbol{C}_2 = \boldsymbol{A}_2\boldsymbol{B}$

Cache oblivious fast Fourier transform (FFT)

- The Fourier transform computes $\boldsymbol{y} = \boldsymbol{D}^{(n)}\boldsymbol{x}$, where $d_{ij}^{(n)} = \omega_n^{ij}$ and ω_n is the nth complex root of identity
  - *$\boldsymbol{D}^{(n)}$ is complex and symmetric (not Hermitian)*
  - *$\boldsymbol{D}^{(n)}$ is unitary modulo scaling (ignored here)*
- A cache-oblivious algorithm for the FFT can be derived by folding $\boldsymbol{y}$ and $\boldsymbol{x}$ into matrices $\boldsymbol{Y}$ and $\boldsymbol{X}$ of dimensions $\sqrt{n} \times \sqrt{n}$
  - *Using the fact that $\omega_n^m = \omega_{n/m}$, and assuming $m = \sqrt{n}$ is an integer, observe*

$$\begin{aligned}\omega_n^{ij} &= \omega_n^{(i_1+mi_2)(j_1+mj_2)} = \omega_n^{i_1j_1}\omega_n^{mi_1j_2}\omega_n^{mi_2j_1}\omega_n^{nj_1j_2} \\ &= \omega_n^{i_1j_1}\omega_m^{i_1j_2}\omega_m^{i_2j_1}\end{aligned}$$

  - *Consequently, $d_{i_1+mi_2,j_1+mj_2}^{(n)} = d_{i_1j_1}^{(n)}d_{i_1j_2}^{(m)}d_{i_2j_1}^{(m)}$, so we can compute $\boldsymbol{Y}$ via*

$$y_{i_1i_2} = \sum_{j_1,j_2}[d_{i_1j_1}^{(n)}(d_{i_1j_2}^{(m)}x_{j_1j_2})]d_{i_2j_1}^{(m)}$$

  - *In matrix form, $\boldsymbol{Y} = (((\boldsymbol{D}^{(m)}\boldsymbol{X}) \odot \boldsymbol{F})\boldsymbol{D}^{(m)})^T$ where $f_{ij} = \omega_n^{ij}$*

Cache oblivious fast Fourier transform (FFT)

- Lets now analyze the cost of the cache oblivious algorithm based on

$$\boldsymbol{Y} = (((\boldsymbol{D}^{(m)}\boldsymbol{X}) \odot \boldsymbol{F})\boldsymbol{D}^{(m)})^T$$

  - *There are $2\sqrt{n}$ recursive calls and $O(n)$ work to apply the Hadamard product at each level, so the work is*

$$W(n) = 2\sqrt{n}W(\sqrt{n}) + O(n) = O(n \log n)$$

  - *The Hadamard product can be applied with depth 1, but half the recursive calls are dependent on the other half, so the depth is*

$$D(n) = 2D(\sqrt{n}) + O(1) = O(\log n)$$

  - *After $\log n / \log H = \log_H n$ recursive calls, $n < H$ and the computation can be done in cache. $O(n)$ memory traffic is required otherwise, so*

$$Q(n) = 2\sqrt{n}Q(\sqrt{n}) + O(n) = O(n \log_H n)$$

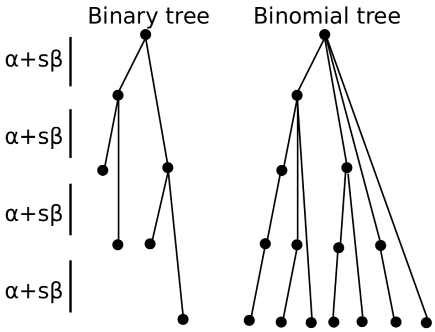
A simple model for point-to-point messages

The time to send or receive a message of s words is $\alpha + s \cdot \beta$

- α – **latency/synchronization cost** *per message*
- β – **bandwidth cost** *per word*
- *each processor can send and/or receive one message at a time*

Consider the cost of a broadcast of s words

- *using a* **binary tree** *of height*
  $h_r = 2(\log_2(p+1) - 1) \approx 2\log_2(p)$
  $$T = h_r \cdot (\alpha + s \cdot \beta)$$
- *using a* **binomial tree** *of height*
  $h_m = \log_2(p+1) \approx \log_2(p)$
  $$T = h_m \cdot (\alpha + s \cdot \beta)$$



Bulk Synchronous Parallel (BSP) Model

- *Bulk Synchronous Parallel (BSP) model* (Valiant 1990)
  - *execution is subdivided into* **supersteps***, each associated with local work and a single round of communication*
  - *within each superstep each processor can send and receive up to h messages (called an h-***relation***)*
  - *in original model, messages were restricted to one-word length*
  - *for modern architectures makes sense to allow each processor to send arbitrary-sized messages to at most h other processors*
- The cost of a BSP algorithm is a sum over supersteps of the maximum costs incurred in that superstep
  - *given S supersteps, where processor i sends and receives W_{ij} a total of words in superstep j and performs F_{ij} local work, we have*

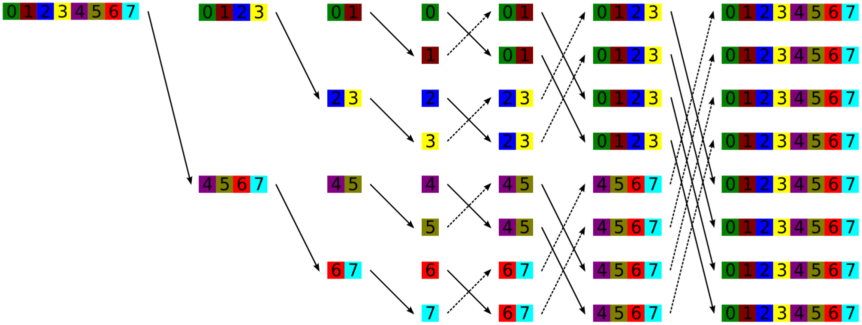
$$T = \sum_{j=1}^{S} \alpha + \beta \max_i W_{ij} + \gamma \max_i F_{ij}$$

  - *recursive definition of algorithms permits extension to asynchronous algorithms (synchronization over subsets of processors)*

Collective communication in BSP

- When $h = p$, most collective communication routines involving s words of data per processor can be done with BSP cost $O(\alpha + s \cdot \beta)$
  - *Scatter: root sends each s/p-sized message to its target (root incurs $s \cdot \beta$ send bandwidth)*
  - *Reduce-Scatter: each processor sends s/p summands to every other processor (every processor incurs $s \cdot \beta$ send and receive bandwidth)*
  - *Gather: send each message of size s/p to root (root incurs $s \cdot \beta$ receive bandwidth)*
  - *Allgather: each processor sends s/p words portion to every other processor (every processor incurs $s \cdot \beta$ send and receive bandwidth)*
  - *Broadcast done by Scatter then Allgather*
  - *Reduce done by Reduce-Scatter then Gather*
  - *Allreduce done by Reduce-Scatter then Allgather*
  - *All-to-all can be done by sending messages directly in one round*
  - *For $h < p$, $O(\log_h p)$ supersteps required, but bandwidth cost same for all except all-to-all (higher by $O(\log_h p)$ via h-ary butterfly protocols*

Butterfly Broadcast



Matrix-vector Product

- Lets design a cache-efficient algorithm for a matrix-vector product
  - *Each processor owns n^2/p matrix data*
  - *Given $O(H)$ matrix data in cache, can perform $O(H)$ work, input/output $O(H)$ vector entries*
  - *Row-wise partitioning avoids write conflicts, achieves $Q = O(n^2/p)$ cache traffic*
- Lets design a BSP algorithm for a matrix-vector product
  - *Each processor starts with matrix and vector data, which it may not need to communicate*
  - *Need to pick initial distribution, assume initial data is not replicated*
  - *First consider row-wise (1D) distribution, communicate $O(n)$ input vector data per processor (all-gather)*
  - *With 2D blocked or cyclic distribution, require $O(n/\sqrt{p})$ input vector data per processor (broadcast or all-gather in processor columns) and contribute $O(n/\sqrt{p})$ partial sums per processor (reduce or reduce-scatter)*

Sparse matrix-vector Product

- 1D distribution is effective for BSP algorithm for SpMV
  - *Each processor assigned n/p input/output vector entries and n/p matrix rows*

$$\begin{bmatrix} \boldsymbol{y}_1 \\ \vdots \\ \boldsymbol{y}_p \end{bmatrix} = \begin{bmatrix} \boldsymbol{A}_1 \\ \vdots \\ \boldsymbol{A}_p \end{bmatrix} \begin{bmatrix} \boldsymbol{x}_1 \\ \vdots \\ \boldsymbol{x}_p \end{bmatrix}$$

  - *Let $\boldsymbol{A}_i = \boldsymbol{A}_i^{\mathit{local}} + \boldsymbol{A}_i^{\mathit{remote}}$ where $\boldsymbol{A}_i^{\mathit{local}}$ is on the block-diagonal*
  - *Can perform local SpMV $\boldsymbol{A}_i^{\mathit{local}}\boldsymbol{x}$ without communication, since only $\boldsymbol{x}_i$ is needed*
  - *ith processor needs to receive entries for each nonzero column in $\boldsymbol{A}_i^{\mathit{remote}}$*
  - *ith processor needs to send entries for each nonzero column in $\boldsymbol{A}_i^{\mathit{remote}}$ assuming $\boldsymbol{A}$ is symmetric (otherwise consider nonzero rows of block-column)*
  - *Algorithm can be more efficient than 2D if the number of such nonzero columns is less than $n/\sqrt{p}$*
  - *Appropriate reorderings of rows and columns (e.g. via graph partitioning), minimize communication*

Massively Parallel Computing (MPC) Model

- Massively Parallel Computing (MPC) model
  - *Given input size* N*, allow each processor to have* $S = N^\alpha$ *memory for* $0 < \alpha < 1$
  - *Let the number of processors be* N/S *so that the input fits into global memory or* $(N/S)\operatorname{polylog} B$
  - *Each round corresponds to local computation followed by arbitrary global communication*
  - *Aim to minimize number of rounds, can simulate PRAM and BSP*
- Aim to achieve $O(\log N)$ or $O(\log\log N)$ rounds with minimal memory per processor
  - *strongly superlinear if* $S \geq N^{1+\epsilon}$
  - *nearly linear if* $S = O(N \operatorname{polylog} N)$
  - *strongly sublinear (scalable) if* $S = N^\gamma$ *for* $\gamma < 1$

Graph Algorithms in MPC

- Graph algorithms in the MPC model for graphs with n vertices
  - *With* $S = \Theta(n)$ *memory can assign a vertex and its incident edges to a processor*
  - *Can perform SpMV with* $O(1)$ *rounds, many graph algorithms in constant or* $O(\mathrm{polylog}(\log n))$ *rounds*
  - *Very active area of current research*

Communication lower bounds

- Given an algorithm (e.g. radix-2 FFT, bitonic sort) or family of algorithms (e.g. radix-k FFT, comparison based sorting algorithms), how much communication is necessary?
  - *How much data or cache lines must be moved between memory and cache?*
  - *How much data must processes communicate in BSP, assuming work or input is load balanced?*
- Communication lower bounds ascertain optimality of communication schedules
  - *For numerical problems, full space of potential algorithms is often too difficult to consider for communication lower bounds*
  - *Communication cost lower bounds consequently focus on a particular set of algorithms*
  - *Often leverage volumetric inequalities to assert surface-area-to-volume bounds*

Classical results in communication lower bounds

- Floyd 1972: for large cache lines $L = \Theta(H)$ *matrix transposition has cost* $O(n^2 \log(n) \cdot \beta)$
- Hong and Kung 1981, pebbling lower bound
  - *model communication as placing pebbles on a dependency graph of an algorithm*
  - *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*
- *with partitioning, all we need is a bound* $f_{alg}(H)$ *on how much useful computation can be done with* $3H$ *inputs + outputs*
- *if the total amount of computation is* F, $Q \geq FH/f_{alg}(H)$

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 $f_{alg}(H)$, we can show that H loads or stores are required*
  - *we then get the desired bound $Q \geq FH/f_{alg}(H)$*

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*

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

Cache complexity lower bound for MM

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

- *we obtain the sequential memory bandwidth lower bound*

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

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

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

Interprocessor communication lower bound for MM

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

- *given that each processor has M memory, $f_{\textsf{MM}}(M)$ tells us how much computation can be done with $3M$ 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*

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

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

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

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

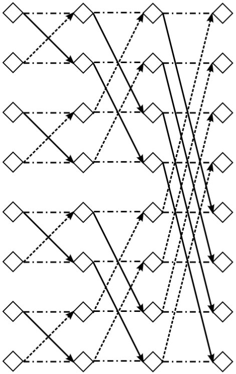
$$W_{\textsf{par-MM}}(n, H, P) = \Omega\left(\frac{n^2}{P^{2/3}}\right)$$

Latency/synchronization lower bounds

From f_{MM} to get lower bounds on the number of messages

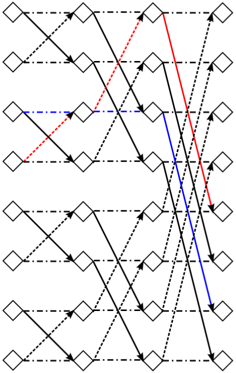
- *Given a cache of size H, $\Omega(n^3/f_{\mathsf{MM}}(H))$ blocks must be transfered between memory and cache*
- *Given $M = 3cn^2/p$ memory, $\Omega((n^3/p)/f_{\mathsf{MM}}(M)) = \Omega(\sqrt{p/c^3})$ messages must be sent or received by some processor*
- *Given $M = 3cn^2/p$ memory, $\Omega((n^3/p)/f_{\mathsf{MM}}(M)) = \Omega(\sqrt{p/c^3})$ BSP supersteps are required*

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



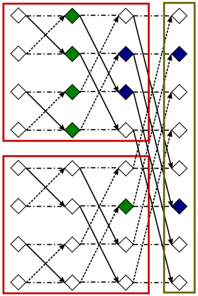
in other words, the FFT DAG has no cycles

Work bound for FFT

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

- *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*
- $f_{FFT}(s) = \max_t(f_{FFT}(s-t) + f_{FFT}(t) + 2\min(s-t,t))$

Communication lower bound for the FFT

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

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

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

$$f_{\mathit{FFT}}(s) = 2f_{\mathit{FFT}}(s/2) + s = s\log_2(s)$$

Given $f_{\mathit{FFT}}(s) = s\log_2(s)$, *the cache complexity is*

$$Q_{\textit{seq-FFT}}(n,H) \geq n\log_2(n)H/f_{\mathit{FFT}}(2H) = n\frac{\log(n)}{2\log(2H)} = \Omega(n\log_H(n))$$

We showed that a radix-$\sqrt{n}$ *FFT algorithm gets this cost.*

Lower bounds via graph partitioning

- Given a DAG representation of an algorithm, graph partitioning properties can provide communication lower bounds
  - *Consider 2-processor load-balanced parallelization to get balanced two-way partitioning of graph*
  - *Vertices with outgoing edges to vertices in the other part must be communicated*
  - *These vertices define a separator between the two parts, since their removal disconnects the graph*
  - *Lower bound on vertex separator size yields lower bound on communication*
- Consideration of expansion of subgraphs can yield better bounds
  - *Two-processor view can yield communication volume lower bound by considering data movement between first half and second half of processors*
  - *Can get better lower bounds like before by obtaining function $f(s)$ on how much useful computation can happen with $3s$ data*
  - *If for any subset of vertices $S \subset V$ with $|S| \leq k \ll |V|$, a separator of size $\Omega(r(k))$ is needed to disconnect S from $V \setminus S$, then $f(s) = O(r^{-1}(s))$*
  - *For irregular graphs, can obtain yet better bounds, by considering best possible partitioning where each subset to has boundary of at most $3H$*

Dependency interval expansion

Consider an algorithm that computes a set of operations V with a partial ordering, we denote a dependency interval between $a, b \in V$ as

$$[a, b] = \{a, b\} \cup \{c : a < c < b, c \in V\}$$

If there exists $\{v_1, \ldots, v_n\} \in V$ *with* $v_i < v_{i+1}$ *and* $\left|[v_{i+1}, v_{i+k}]\right| = \Theta(k^d)$ *for all* $k \in \mathbb{N}$*, then*

$$F \cdot S^{d-1} = \Omega(n^d)$$

where F *is the computation cost and* S *is the synchronization cost*

Dependency interval expansion

Consider an algorithm that computes a set of operations V with a partial ordering, we denote a dependency interval between $a, b \in V$ as

$$[a, b] = \{a, b\} \cup \{c : a < c < b, c \in V\}$$

If there exists $\{v_1, \ldots, v_n\} \in V$ *with* $v_i < v_{i+1}$ *and* $\big|[v_{i+1}, v_{i+k}]\big| = \Theta(k^d)$ *for all* $k \in \mathbb{N}$*, then*

$$F \cdot S^{d-1} = \Omega(n^d)$$

where F *is the computation cost and* S *is the synchronization cost*

Dependency interval expansion

Consider an algorithm that computes a set of operations V with a partial ordering, we denote a dependency interval between $a, b \in V$ as

$$[a, b] = \{a, b\} \cup \{c : a < c < b, c \in V\}$$

If there exists $\{v_1, \ldots, v_n\} \in V$ *with* $v_i < v_{i+1}$ *and* $\left|[v_{i+1}, v_{i+k}]\right| = \Theta(k^d)$ *for all* $k \in \mathbb{N}$*, then*

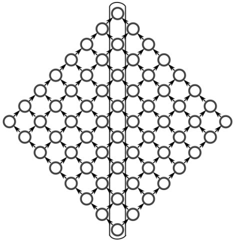
$$F \cdot S^{d-1} = \Omega(n^d)$$

where F *is the computation cost and* S *is the synchronization cost*

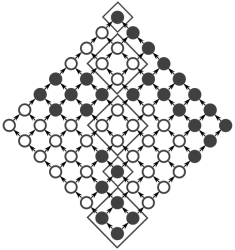
Further, if the algorithm has a work bound $f(H) = \Omega(H^{\frac{d}{d-1}})$, then

$$W \cdot S^{d-2} = \Omega(n^{d-1})$$

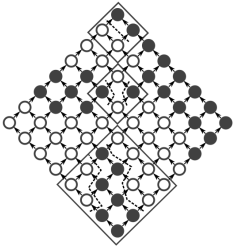
Example: diamond DAG



Dependency chain P



Monochrome dependency intervals



Multicolored dependency intervals

For the $n \times n$ diamond DAG ($d = 2$),

$$
\begin{aligned}
F \cdot S^{d-1} &= F \cdot S = \Omega((n/b)b^2) \cdot \Omega(n/b) = \Omega(n^2) \\
W \cdot S^{d-2} &= W \quad\; = \Omega((n/b)b) \qquad\qquad\quad\; = \Omega(n)
\end{aligned}
$$

idea of $F \cdot S$ tradeoff goes back to Papadimitriou and Ullman, 1987

Tradeoffs involving synchronization

For triangular solve with an $n \times n$ matrix

$$F_{\mathrm{TRSV}} \cdot S_{\mathrm{TRSV}} = \Omega\left(n^2\right)$$

For Cholesky of an $n \times n$ matrix

$$F_{\mathrm{CHOL}} \cdot S_{\mathrm{CHOL}}^2 = \Omega\left(n^3\right) \qquad W_{\mathrm{CHOL}} \cdot S_{\mathrm{CHOL}} = \Omega\left(n^2\right)$$

For computing s applications of a $(2m+1)^d$-point stencil

$$F_{\mathrm{St}} \cdot S_{\mathrm{St}}^d = \Omega\left(m^{2d} \cdot s^{d+1}\right) \qquad W_{\mathrm{St}} \cdot S_{\mathrm{St}}^{d-1} = \Omega\left(m^d \cdot s^d\right)$$