CSE 6220 High-Performance Computing Lecture Notes

Jie Wu, Jack

Spring 2022

1 Introduction to High Performance Computing

The course covers three machine models:

parallel RAM model: one memory + multiple processors, processors communicate via shared variables, $\text{cost}(n; P) = O(n \log n/P)$

distributed memory model: a network of memory-processor modules i.e. RAM computers, no processor can read or write the memory of any other processors, $\text{cost}(n; P) = \alpha f(n; P) + \beta g(n; P)$, the second term accounts for the volume of communication

two-level I/O model: one or more processors connected to a single main memory, but there is a level of fast memory (e.g. cache, virtual memory) that sits in between the processor(s) and the slow main memory

2 Basic Model of Locality

2.1 A First Basic Model

von Neumann model: a processor connects to a slow main memory through a fast but small memory with size Z

rule: there are two

local data rule: processor may only compute on data in fast memory

block transfer rule: slow-fast transfers in blocks of size L words, so that loading a word from slow memory to a fast memory requires loading the adjacent $L-1$ words, thus data alignment may become an issue

cost: two components

work: $W(n) \equiv \#$ of computation operations

transfer: $Q(n; Z, L) \equiv \#$ of L-sized slow-fast transfers

e.g. sum the elements of an array of size n

work: $W(n) \geq n - 1$ additions $= \Omega(n)$

transfer: $Q(n; Z, L) \geq \lceil n/L \rceil = \Omega(n/L)$, note that it doesn't depend on Z

2.2 Alignment Quiz

Q: how many transfers are necessary in the worst case?

A: $\lceil n/L \rceil + 1$

2.3 Minimum Transfers to Sort Quiz

Q: to sort an array of size n, $W(n) = \Omega(n \log n)$ for a comparison-based sort, what is the lower bound on the asymptotic number of transfers?

A: an obvious guess is $Q(n; Z, L) = \Omega(\lceil n/L \rceil)$, actual is $Q(n; Z, L) = \Omega((n/L) \log(n/L) / \log(Z/L))$

2.4 Minimum Transfers to Multiply Matrices

Q: $C = A \cdot B$ where A, B, C are $n \times n$ matrices, what is the lower bound on the asymptotic number of transfers?

A: an obvious guess is $Q(n; Z, L) = n^2/L$ (number of elements/L), actual is $Q(n; Z, L) = \Omega(n^3/L\sqrt{Z})$

2.5 I/O Example Reduction

To explicitly address the existence of cache and slow-fast transfer, we break the array summation loop into two

$$\begin{array}{l} \text{for i} \leftarrow 0 \text{ to } n-1 \text{ do} \\ \quad \text{sum} \leftarrow \text{sum} + \text{array}[i] \end{array} \quad \Longrightarrow \quad \begin{array}{l} \text{for i} \leftarrow 0 \text{ to } n-1 \text{ by } L \text{ do} \\ \quad \text{cache}[0 : L-1] \leftarrow \text{array}[i : (i+L-1)] \\ \quad \text{for j} \leftarrow 0 \text{ to } L-1 \text{ do} \\ \quad\quad \text{sum} \leftarrow \text{sum} + \text{cache}[j] \end{array}$$

2.6 Matrix Vector Multiply Quiz

If the matrix is stored column wise, then one column follows previous column in memory. As a result for a matrix multiplication $y = A \cdot x$ (assuming column fits cache size and alignment)

- $Q(n; Z, L) = 3n/L + n^2$ for outer loop iteration over rows + inner loop iteration over columns, because loading the next element in a row will require loading the entire next column
- $Q(n; Z, L) = 3n/L + n^2/L$ for outer loop iteration over columns + inner loop iteration over rows, because the inner loop can be carried out over the column that is transferred to cache

where $3n$ is for loading x and y into fast memory and storing updated y back to slow memory.

2.7 Algorithmic Design Goals

work optimality: the two-level algorithm should do the same work as the best asymptotic/RAM algorithm $W(n) = \Theta(W_*(n))$

high computational intensity: denoted by $I(Q; Z, L)$ **intensity** is defined as

$$I(n; Z, L) = \frac{W(n)}{L \cdot Q(n; Z, L)}$$

it has a unit of operation/word transfer and measures the data reuse of the algorithm, we want to maximize the reuse as long as the work is optimized

2.8 Intensity, Balance, and Time

Let τ be time per operation so that time to compute $T_{\text{comp}} = \tau W$, and α (amortized) time per word of slow-fast transfer so that time to execute Q transfers $T_{\text{mem}} = \alpha LQ$, assuming perfect overlap between operation and transfer gives a lower bound to the execution time

$$T \geq \max(T_{\text{comp}}, T_{\text{mem}}) = \tau W \cdot \max\left(1, \frac{\alpha/\tau}{W/LQ}\right) = \tau W \cdot \max\left(1, \frac{B}{I}\right)$$

where B measuring operation per word transfer is called **machine balance**. Note that B is machine specific whereas I is algorithm specific. Assuming no overlap between operation and transfer gives an upper bound to the execution time

$$\tau W \cdot \max\left(1, \frac{B}{I}\right) \leq T \leq \tau W \cdot \left(1 + \frac{B}{I}\right)$$

normalized performance is defined as

$$R \equiv \frac{\tau W_*}{T} \leq \frac{W_*}{W} \cdot \underbrace{\min\left(1, \frac{I}{B}\right)}_{\text{communication penalty}}$$

2.9 Roofline Plots

Plotting $R_{\max}$ against I we get a line that increases linearly with I and then levels off at $I = B$. The value of the plateau is $R_{\max} = W_*/W$. Conventionally we say an algorithm that lies to the right of $I = B$ as *compute-bound* and that lies to the left of $I = B$ as *memory-bound*.

2.10 Intensity of Conventional Matrix Multiply

If we operate on element by element

$$\begin{aligned} W(n) &= n^2(\text{elements in } C) \times n(\text{sum over } n \text{ products of } A \text{ and } B \text{ for each element in } C) \\ Q(n; Z) &= n^2(\text{elements in } A) + 2n^2(\text{read \& write elements in } C) \\ &\quad + n^3(\text{loop over } n \text{ columns of } B \text{ for each element in } A) \end{aligned}$$

the intensity is $\Theta(1)$.
If we operate on block by block, each block of size $b \times b$, $W(n)$ is still the same, so is the transfer of A and C because number of reads per block $\times$ number of blocks to read $= b^2 \times (n/b)^2 = n^2$. But for the transfer of B it becomes $b^2 \times (n/b)^3 = n^3/b$. Hence the intensity becomes $I = \Theta(b) = \Theta(\sqrt{Z})$ (we require fast memory to have a size $Z = 3b^2 + O(1)$ to store one block of A and two blocks of C—one for read and one for write).

2.11 Informing the Architecture Quiz

If the machine balance B doubles, since $I = \sqrt{Z}$ the size of fast memory Z should be increased by four fold in order to keep the communication penalty the same.

2.12 Conclusion

The main technique in this lesson is to organize data access to increase data reuse. The general rule of thumb if you want your algorithm to scale well to future memory hierarchies is that, you want your algorithm to at least match and preferably exceed machine balance.

3 I/O-Avoiding Algorithms

3.1 External Memory Mergesort

Assume total n items, fast memory can store Z items, and L items per slow-fast transfer. The merge sort consists of two phases:

phase 1: partition into n/Z chunks, for each chunk $i \leftarrow 1$ to n/Z do: (1) read chunk i (2) sort it into a sorted chunk called "run" (3) write run i

phase 2: merge the n/Z runs into a single run

3.2 Partition Sorting Step Analysis

The number of asymptotic transfers for each of the three steps of phase 1 are

read chunk: Z/L transfers per chunk $\times$ n/Z chunks $= O(n/L)$

sort chunk: $Z \log_2 Z$ comparisons per chunk $\times$ n/Z chunks $= O(n \log_2 Z)$

write run: same as read chunk i.e. $O(n/L)$

3.3 Two Way External Memory Merging

Assume 2^k runs of size s so that $n = 2^k s$. The conventional method is to merge pairs of runs and then pairs of pairs so that the run size grows from s to $2s$, $2^2 s$, $\ldots$, $2^{k-1}s$, $2^k s$. To merge two runs of size $2^{k-1}s$

number of transfers: (read) $2^{k-1}s/L + 2^{k-1}s/L$ + (write) $2^k s/L = 2^{k+1}s/L$ transfers per pair $\times$ $\log_2(n/s)$ number of levels $= (2^{k+1}s/L)\log_2(n/s) = (2n/L)\log_2(n/s)$

number of comparison: $\Theta(2^k s) = \Theta(n)$ number of comparisons per level (sorted thus is $\Theta(n)$ not $\Theta(n \log_2 n)$) $\times$ $\log_2(n/s)$ number of levels $= \Theta(n \log_2(n/s))$

3.4 External Memory Mergesort Question

Combine the results from the previous two subsections

phase 1: comparison $= O(n \log_2 Z)$, transfer $= O(n/L)$

phase 2: comparison $= O(n \log_2(n/Z))$, transfer $= O(n/L \log_2(n/Z))$ ($\Rightarrow s = L$ and $Z = 3L$?)

we have comparison $= O(n \log_2 n)$, transfer $= O(n/L \log_2(n/Z))$.

3.5 What's Wrong with 2 Way Merging

The number of transfers in external memory mergesort with 2-way merging is

$$Q(n; Z, L) = O\left(\frac{n}{L}\log_2\frac{n}{Z}\right) = O\left(\frac{n}{L}\left[\log_2\frac{n}{L} - \log_2\frac{Z}{L}\right]\right)$$

The lower bound is

$$Q(n; Z, L) = \Omega\left(\frac{n}{L}\log_{Z/L}\frac{n}{L}\right) = \Omega\left(\frac{n}{L}\frac{\log_2\frac{n}{L}}{\log_2\frac{Z}{L}}\right)$$

Their ratio, which is also the remaining factor of potential improvement, is

$$O\left(\log_2\frac{Z}{L}\left[1 - \frac{\log_2\frac{Z}{L}}{\log_2\frac{n}{L}}\right]\right)$$

3.6 Multiway Merging

To fully utilize fast memory we should do multiway merging so that the M runs to be merged (the input) and the merged block (the output) fully occupy the fast memory i.e. $(M+1)L \approx Z$ (when the output block is filled, it is flushed by writing it to slow memory) ($\Rightarrow$ hinted by the Z/L base). To find the smallest value of all the input runs, a linear scan can be used if k is small, or a priority queue or min heap can be used, which involves the following operations

build the heap: $O(M)$ operations

extract min: $O(\log M)$ operations

insert: $O(\log M)$ operations

These are all fast memory operations so we can count them as comparisons. Hence the cost of M-way merge is

transfer: Ms items read + Ms items write divided by L gives $2Ms/L$

comparison: $O(M)$ build + $O(\log M)$ either extraction or insertion for each of the Ms items gives $O(M + Ms \log M)$

3.7 Cost of Multiway Merge Quiz

Mimic the analysis of two-way external memory merging. Assume M^k runs of size s so that $n = M^k s$. The merged block size grows from s to Ms, $M^2 s$, ..., $M^{k-1}s$, $M^k s$. To merge M runs at level $i-1$ into one block at level i, the number of transfer per merge is $\Theta(M^i s/L)$ as there are $M^{i-1}s \times M = M^i s$ items, and the number of merges at level i is $n/M^i s$. Therefore the total number of transfers at level i is $\Theta(n/L)$ which is independent of level i. Hence the total number of transfers is

$$\Theta(n; Z, L) = \underbrace{\Theta\left(\frac{n}{L}\right)}_{\#\text{ transfers per level}} \times \underbrace{\log_M \frac{n}{s}}_{\#\text{ levels}} = \Theta\left(\frac{n}{L} \log_{\frac{Z}{L}} \frac{n}{L}\right)$$

where in the last equality we have used the fact that $(M+1)L \approx Z$ ($\Rightarrow$ and $s = L$?).

3.8 A Lower Bound on External Memory Sorting

Let $f(t-1)$ denote the number of possible orderings remains after $t-1$ transfers. Thus $f(0)$ = number of all possible orderings of n items which is $n!$. The maximum number of reductions in the remaining possible orderings going from $t-1$ transfers to t transfers from slow memory occurs when all of the L items in the tth transfer are new ($\Rightarrow$ since the position referencing ruler is expanded to the maximum extent). Because these L new items induces $\binom{Z}{L} L!$ possible orderings relative to the ordered block in fast memory after the transfer, where $\binom{Z}{L}$ refers to the number of possible positions for the L new items in fast memory, we have the following inequality regarding the remaining possible orderings after t transfers

$$f(t) \geq \frac{f(t-1)}{\binom{Z}{L} L!}$$

Apply this recursive inequality starting from $f(0) = n!$ we have

$$f(t) \geq \frac{n!}{\left[\binom{Z}{L} L!\right]^t} = \frac{n!}{\left[\binom{Z}{L}\right]^t (L!)^t} \geq \frac{n!}{\left[\binom{Z}{L}\right]^t (L!)^{n/L}}$$

The last inequality stems from the fact that the maximum number of L-sized reads, each of which contains only new items, is n/L. Then the smallest t that results in $f(t) \leq 1$ i.e. final ordered state is given by

$$\log n! \leq t \log \binom{Z}{L} + (n/L) \log L! \implies t \geq \frac{\log n! - (n/L) \log L!}{\log \binom{Z}{L}}$$

Apply Stirling's approximation $\log x! \approx x \log x$ and the approximation $\log \binom{a}{b} \approx b \log(a/b)$ we have

$$t \geq \frac{n \log n - n \log L}{L \log(Z/L)} = \frac{n}{L} \frac{\log(n/L)}{\log(Z/L)} = \frac{n}{L} \log_{\frac{Z}{L}} \frac{n}{L}$$

3.9 How Many Transfers in Binary Search

We use the following recursive relation

$$Q(n; Z, L) = \begin{cases} 1 + Q(n/2; Z, L) & \text{if } n > L \\ 1 & \text{if } n \leq L \end{cases}$$

Thus the number of transfers is increased by 1 as we go one level up. Hence the number of transfers $Q(n; Z, L) = O(\log_2 n/L)$.

3.10 Lower Bounds for Search

To find the largest index i such that $A[i] \leq v$, think of the problem in binary. The index i searched for has $\lfloor \log n \rfloor + 1 = \log n$ bits. Let $x(L)$ be the maximum number of bits that can be learned from a L-sized read. The maximum bit learning occurs when it is determined that the index i searched for lies to the left or right of the L-block read, thus $x(L) = \log L$ ($\Rightarrow$ number of bits to represent L). Therefore the lower bound of the number of transfers for search is $O(\log n / \log L)$ or $O(\log_L n)$.

3.11 I/O Efficient Data Structures Quiz

Out of the following search algorithms (1) doubly-linked list (2) binary search tree (3) red-black tree (4) skip list (5) B-tree, only B-tree can attain the lower bound of the slow-fast memory transfer $\log_L n$. In a B-tree because the key value of a node must lie between the key values of its children, it can be shown that the height of a B-tree is $\log_B n$. It therefore can be made IO optimal, but only if you choose the right branching factor $B = \Theta(L)$.

3.12 Conclusion

In the above analysis we have assumed that the time spent moving data dominates, hence we should look for ways to reduce I/Os.

4 Cache-Oblivious Algorithms (Optional)

4.1 An Introductory Example

Oblivious to fast memory algorithms are those that make no reference to fast memory Z or its transfer parameter L. From now on any automatic fast memory is referred to as "cache".

4.2 The Ideal-Cache Model

Assume that cache is divided into blocks of size L words (i.e. equal to the transfer size), each of which is referred to as "cache line". The assumptions regarding the operations of ideal cache are

1. the program issues loads (from slow memory) and stores (into slow memory) sequentially
2. the hardware has to transfer the entire cache line for loads and stores
3. value found in cache is called "cache hit", otherwise it is a "cache miss" which triggers loading from slow memory
4. cache is fully associative i.e. a block of size L can go into any cache line
5. **optimal replacement** if some cache line needs to be evicted (which triggers storing into slow memory) to make room for next storage, i.e. cache knows future access and evicts the one that will be accessed the most distantly in the future

Under these assumptions number of slow memory-cache transfers is

$$Q(n; Z, L) = \text{number of cache misses} + \text{number of store-evictions}$$

4.3 How Ideal is the Ideal Cache?

The lemma (where LRU = least recently used, OPT = optimal replacement)

$$Q_{\text{LRU}}(n; Z, L) \leq 2 \cdot Q_{\text{OPT}}(n; Z/2, L)$$

together with the **regularity condition**—doubling cache size doesn't change the asymptotic number of transfers

$$Q_{\text{OPT}}(n; Z, L) = O\left(Q_{\text{OPT}}(n; 2Z, L)\right)$$

will lead to

$$Q_{\text{LRU}}(n; Z, L) = \Theta\left(Q_{\text{OPT}}(n; Z, L)\right)$$

which asserts that Q_{LRU} and Q_{OPT} are asymptotically close, in other words optimal replacement isn't as strong an assumption as one might think.

4.4 Proof of the LRU-OPT Competitiveness Lemma

Matrix multiplication satisfies the regularity condition, for which $Q_{\text{OPT}}(n; Z) = \Theta(n^3/\sqrt{Z})$ assuming $L = 1$ and the multiplication is done on blocks of $\sqrt{Z} \times \sqrt{Z}$. The plausibility of the lemma can be seen by comparing the maximum number of evictions for $Q_{\text{LRU}}(Z)$ with the minimum number of evictions for $Q_{\text{OPT}}(Z/2)$ on Z unique addresses

$$Q_{\text{LRU}}(Z) \leq Z \leq 2 \cdot Q_{\text{OPT}}(Z/2)$$

4.5 LRU Sorting Quiz

A sorting algorithm has on an ideal (Z, L)-cache

$$Q_{\text{OPT}}(n; Z, L) = \Theta\left(\frac{n}{L}\frac{\log\frac{n}{L}}{\log\frac{Z}{L}}\right)$$

satisfies the regularity condition (since $\log(2Z/L) = \log(Z/L) + \log 2$). Therefore an LRU cache will perform just as well as the optimal cache asymptotically.

4.6 The Tall-Cache Assumption

To fit a $b \times b = \Theta(Z)$ submatrix in cache we must have $Z \geq L^2$ or $Z = \Omega(L^2)$. This is the tall-cache assumption. It requires that the cache should be taller than its width, or equivalently number of lines in cache $\geq$ number of words per line i.e. $Z/L \geq L$.

4.7 Cache-Oblivious Matrix Multiply

Divide into four quadrants and perform matrix multiplication. Divide down till scalar case $c \leftarrow c + a \cdot b$. The number of flops i.e. scalar multiplications and additions is

$$F(n) = \begin{cases} 8 \cdot F(n/2) & \text{if } n > 1 \\ 2 & \text{if } n = 1 \end{cases} \implies F(n) = 2n^3$$

The recursive relation implies that in the recursion tree each node of function call has 8 branches accompanied by a matrix size reduction of 2. The branching carries out until the matrix size $n_l = n/2^l$ fits into the cache $3n_l^2 \leq Z$ or equivalently $n_l \leq f \cdot \sqrt{Z}$. The number of cache misses then satisfies the following recursive relation (the base case $n \leq f \cdot \sqrt{Z}$ is when the operands fit in cache)

$$Q(n; Z, L) = \begin{cases} \Theta(n^2/L) & \text{if } n \leq f \cdot \sqrt{Z} \\ 8 \cdot Q(n/2; Z, L) + O(1) & \text{if } n > f \cdot \sqrt{Z} \end{cases} \implies Q(n; Z, L) = O\left(\frac{n^3}{L\sqrt{Z}}\right)$$

which matches the cache-aware (cache size used) algorithm as well as the lower bound.

4.8 Cache-Oblivious Binary Search

Similar to matrix multiplication the number of cache misses satisfies the following recursive relation

$$Q(n; Z, L) = \begin{cases} 1 & \text{if } n \leq L \\ 1 + Q(n/2; Z, L) & \text{if } n > L \end{cases} \implies Q(n; Z, L) = O\left(\log\frac{n}{L}\right)$$

which is a factor of $\log L$ larger than the lower bound $\Omega(\log_L n) = \Omega(\log n / \log L)$.

4.9 van Emde Boas Layout Question

One way to achieve the lower bound $\Omega(\log n / \log L)$ is to use van Emde Boas layout. The idea is to partition binary search tree and store the upper sub-tree in cache instead of linear partition, so that the maximum height of the sub-tree cache line is $\log L$ (the denominator of the lower bound) because the sub-trees are of size L. Cache miss is generated only when you hit the root of one of the sub-trees. This layout reflects the fact that data structure matters.

5 Intro to the Work-Span Model

5.1 The Multithreaded DAG Model

In the multithreaded DAG model you represent a parallel computation by a directed acyclic graph or DAG, in which

vertex: operation (e.g. addition, multiplication)

directed edge: how an operation depends on others where the sinks depend on the sources

For simplicity we will assume that there is exactly one starting vertex and one exit vertex in a DAG.

5.2 Example Sample Reduction

The execution time of summing an array of n elements is (where P = number of processors)

$$T_P(n) \geq \max\left(\underbrace{\lceil\frac{n}{P}\rceil}_{\text{load}}, \underbrace{n}_{\text{addition}}\right) = n$$

Note that there is an implicit dependence called control dependence since the summation is executed sequentially. But for now we are going to ignore the control dependence.

5.3 A Reduction Tree Question

Since we assume sufficient number of processors $P \geq n$, load time becomes $O\left(\lceil n/P \rceil\right) = O(1)$ and the addition at each level takes $O(1)$ time to complete. As a result the process time is just the number of levels in the binary tree which is $O(\log n)$.

5.4 Work and Span

work: total number of vertices in a DAG, denoted by $W(n)$

span: the number of vertices on the longest path, historically called depth, denoted by $D(n)$

If all the operations have unit cost, then the execution time for one processor $T_1(n) = W(n)$. With an infinite number of processors the execution time is determined by the length of the longest path $T_\infty(n) = D(n)$.

5.5 Basic Work Span Laws

The ratio $W(n)/D(n)$ basically measures the amount of work per critical path vertex. Thus $W(n)/D(n)$ basically tells the **average available parallelism**, and suggests the optimal number of processors for execution. The execution time is bounded below by

span law: $T_P(n) \geq D(n)$

work law: $T_P(n) \geq \lceil W(n)/P \rceil$

Combined we have the following **work-span law**

$$T_P(n) \geq \max\left\{D(n), \left\lceil\frac{W(n)}{P}\right\rceil\right\}$$

5.6 Brent's Theorem

step #1: break execution into phases

- each phase has one critical path vertex, thus there will be $D(n)$ phases
- non-critical path vertices in each phase are independent i.e. no edges in between, which is always achievable (proof by contradiction with the fact that they are non-critical), this implies that the execution time for phase k is $t_k = \lceil W_k/P \rceil$ ($\Rightarrow$ = or $\leq$?), hence total execution time is

$$T_P = \sum_{k=1}^{D} \left\lceil \frac{W_k}{P} \right\rceil$$

- every vertex must belong to one and only one phase, this implies that $W = \sum_{k=1}^{D} W_k$

step #2: convert ceiling to floor using identity, which provides a natural upper bound $\lfloor x \rfloor \leq x$

$$\left\lceil \frac{a}{b} \right\rceil \equiv \left\lfloor \frac{a-1}{b} \right\rfloor + 1 \implies$$

$$T_P = \sum_{k=1}^{D} \left\lceil \frac{W_k}{P} \right\rceil = \sum_{k=1}^{D} \left(\left\lfloor \frac{W_k - 1}{P} \right\rfloor + 1 \right) \leq \sum_{k=1}^{D} \left(\frac{W_k - 1}{P} + 1 \right) = \frac{W - D}{P} + D$$

Combine it with the work-span law above we have

$$\max \left\{ D, \left\lceil \frac{W}{P} \right\rceil \right\} \leq T_P \leq \frac{W - D}{P} + D$$

Note that the lower bound is within a factor of two of the upper bound.

5.7 Desiderata - Speedup, Work-Optimality, and Weak-Scalability

Define **speedup** as the best sequential time divided by the parallel time

$$S_P(n) \equiv \frac{T_*(n)}{T_P(n)} = \frac{W_*(n)}{T_P(n)}$$

ideally we want the parallel algorithm running on P processors to be P times faster than the best sequential algorithm $S_P(n) = \Theta(P)$, this condition is called ideal speedup or linear speedup or ideal scaling. Apply Brent's theorem we have

$$S_P(n) = \frac{W_*(n)}{T_P(n)} \geq \frac{W_*}{\frac{W - D}{P} + D} = \frac{P}{\frac{W}{W_*} + \frac{P-1}{W_*/D}}$$

It shows that to get ideal scaling $S_P(n) = \Theta(P)$, the denominator should be a constant i.e. $W/W_* + (P-1)/(W_*/D) = O(1)$, which requires

work optimality $W = O(W_*)$ i.e. the work of the parallel algorithm has to match that of the best sequential algorithm

weak scalability: $P = O(W_*/D)$ or equivalently $W_*/P = \Omega(D)$ i.e. the work per processor has to grow in proportion to the span, which in turn depends on the problem size n

Work optimality and weak scalability are two fundamental principles of good parallel algorithm design.

5.8 Basic Concurrency Primitives

spawn: a signal to either the compiler or the runtime system that the target is an independent unit of work, inserting the keyword spawn effectively indicates that the target may be executed asynchronously from the caller any time a processor is available

sync: wait for any spawn that has occurred so far within the same stack frame, there will always be an implicit sync at the return immediately before going back to the caller, the implicit syncs will constrain the kind of DAGs that this programming model can produce, the style of parallelism in such DAGs is sometimes called *nested parallelism*

5.9 A Subtle Point about Spawn Quiz

One of the following spawns can be eliminated without increasing the span asymptotically

```
if n ≥ 2 then {
    a ← spawn reduce (A[0 : n/2 − 1])
    b ← ~~spawn~~ reduce (A[n/2 : n − 1])
    sync
    return a + b
} else n = 1
    return A[0]
```

5.10 Basic Analysis of Work and Span

Assume that each spawn and sync is a constant time operation i.e. $O(1)$, then the recurrence for work is analogous to the recurrence for sequential execution time and we will get linear work

$$W(n) = \begin{cases} 2 \cdot W(n/2) + O(1) & \text{if } n \geq 2 \\ O(1) & \text{if } n \leq 1 \end{cases} \implies W(n) = O(n)$$

As for span, since the span only depends on the problem size, and we can always divide and conquer the problem into equal size half for each, the recurrence for span is as follows

$$D(n) = \begin{cases} D(n/2) + O(1) & \text{if } n \geq 2 \\ O(1) & \text{if } n \leq 1 \end{cases} \implies D(n) = O(\log n)$$

5.11 Desiderata for Work and Span

$$\left.\begin{array}{r} \text{work optimality: } W(n) = W_*(n) = O(n) \text{ if possible} \\ \text{low span: } D(n) = O(\log^k n) \text{ i.e. poly-logarithmic} \end{array}\right\} \implies \frac{W}{D} = O\left(\frac{n}{\log^k n}\right)$$

so that the average available parallelism grows with n, close to linearly.

5.12 Concurrency Primitive - Parallel For

"par-for" means all iterations are independent of one another so that they can be executed in any order. The end of a par-for loop will include an implicit sync point, which will force all the independent iterations to join. Suppose $W_i = O(1)$ then $W_{\text{parfor}}(n) = O(n)$.

5.13 Implementing par-for

The binary tree recursion implementation of par-for, parfor(fcn,a,b), is as follows

$n \leftarrow b - a + 1$
if $n \geq 2$ then {
    $m \leftarrow a + \lfloor n/2 \rfloor$
    spawn parfor(fcn,a,$m-1$)
    parfor(fcn,m,b)
    sync
} else if $n = 1$
    return fcn(a)

for which $D(n) = O(\log n)$. We will assume this implementation from now on.

5.14 Data Races and Race Conditions

In the matrix multiplication $y \leftarrow y + A \cdot x$

parfor $i \leftarrow 1$ to n do
    for $j \leftarrow 1$ to n do
        $y[i] \leftarrow y[i] + A[i,j] \cdot x[j]$

all inner loop iterations j write to the same location $y[i]$. This situation is called **data race**, which is formally defined as at least one read and one write may happen at the same memory location at the same time. Data race that leads to an error is called a **race condition**, although a data race does not always lead to a race condition.

5.15 Putting it All Together

The outer par-for loop gives $\log n$ levels whereas the inner loop remains sequential thus has a span of n. Combined the span of the matrix multiplication is $O(\log n + n) = O(n)$. Since $W(n) = O(n^2)$ as we have two loops each of n iterations, the average available parallelism is $O(n)$.
A better implementation is to use a temporary storage to avoid data race so as to enable par-for in the inner loop as well:

parfor $i \leftarrow 1$ to n do
    let $t[1:n]$ be a temporary array
    parfor $j \leftarrow 1$ to n do
        $t[j] \leftarrow A[i,j] \cdot x[j]$
    $y[i] \leftarrow y[i] + \text{reduce}(t)$

for which the span is $O(\log n + \log n) = O(\log n)$ and the average available parallelism is $O(n^2/\log n)$.

5.16 Vector Notation

The use of temporary array above can be denoted by $t[:] \leftarrow A[i,:] \cdot x[:]$, which can be easily converted into a par-for loop (implicit par-for) thus has $W = O(n)$ and $D = O(\log n)$.

5.17 Conclusion

The three ideas of parallel algorithm design are

1. good algorithms are work optimal and have low span and weak scalability (cf. Amdahl's law)
2. divide and conquer should often be the first thing to try
3. separate how to express concurrency from how to execute it

6 Comparison Sort

6.1 Bitonic Sequences

A sequence $(a_0, a_1, \ldots, a_{n-1})$ is **botonic** if $a_0 \leq a_1 \leq \cdots \leq a_i$ and $a_{i+1} \geq \cdots \geq a_{n-1}$. Note that this definition presumes that necessary circular shift is already done for the non-increasing subsequence.

6.2 Bitonic Splits

The pairing (convolution-like) and taking minimum/maximum of the pairs split one bitonic sequence into two non-overlap bitonic sequences, paving the way for divide-and-conquer.

$$
\begin{aligned}
&\text{parfor } i \leftarrow 1 \text{ to } n/2 - 1 \text{ do} \\
&\quad a \leftarrow A[i] \\
&\quad b \leftarrow A[i + n/2] \\
&\quad A[i] \leftarrow \min(a, b) \\
&\quad A[i + n/2] \leftarrow \max(a, b)
\end{aligned}
$$

The work is $O(n)$ and the span is $O(\log n)$ (for par-for). The subtle point here is that in the work-span model the fix sized circuit (our comparator network) has a constant span.

6.3 Generate a Bitonic Sequence

The process is opposite to bitonic merge in the sense that

bitonic merge: start with bitonic sequence and split into smaller and smaller bitonic sequences

$$
\begin{aligned}
&\text{bitonic split } A[0 : n-1] \\
&\text{spawn bitonic merge}_+ \ A[0 : n/2 - 1] \\
&\text{bitonic merge}_+ \ A[n/2 : n-1]
\end{aligned}
$$

The work is $O(n \log n)$ and the span is $O(\log^2 n)$, because the work and span of bitonic split are $O(n)$ and $O(\log n)$ respectively and there are $\log n$ levels of bitonic splits.

bitonic sequence: start with bitonic sequences of size 2 (apply comparators to adjacent pairs) and merge in to larger and larger bitonic sequence (subscript $+/-$ indicates ascending/descending)

$$
\begin{aligned}
&\text{spawn generate bitonic } A[0 : n/2 - 1] \\
&\text{generate bitonic } A[n/2 : n-1] \\
&\text{sync} \\
&\text{spawn bitonic merge}_+ \ A[0 : n/2 - 1] \\
&\text{bitonic merge}_- \ A[n/2 : n-1]
\end{aligned}
$$

6.4 Bitonic Sort

Bitonic sort consists of (1) generating a bitonic sequence followed by (2) bitonic merge. The span of bitonic sort is optimal but the work of bitonic sort is not

$$W_{bs}(n) = \Theta(n \log^2 n) \qquad vs. \qquad D_{bs}(n) = \Theta(\log^3 n)$$

7 Scans and List Ranking

7.1 A Naive Parallel Scan

The span for a parallelized scan is $O(\log n)$ because both par-for and reduce have logarithmic span. The work is $O(n^2)$ because each reduce costs $O(i)$ work so the total work is the sum $\sum_1^n O(i) = O(n^2/2)$, which is worse than the sequential operation $O(n)$.

7.2 Parallel Scan Analysis

Let I_E be the even indices and I_O the odd indices, the recursive steps are

$$\begin{aligned} A[I_E] &\leftarrow A[I_E] + A[I_O] && n/2 \text{ additions} \\ A[I_E] &\leftarrow \text{addScan}(A[I_E]) \\ A[I_O] &\leftarrow A[I_E[1:]] + A[I_O[2:]] && n/2 - 1 \text{ additions} \end{aligned}$$

hence the work satisfies the following recurrence

$$W(n) = W(n/2) + (n-1) \implies W(n) = O(n)$$

which has a hiding constant 2 i.e. $O(2n)$. In contrast the sequential algorithm has a constant 1.
For span since we can use par-for for the $n/2$ and $n/2 - 1$ additions, the span satisfies the following recurrence

$$D(n) = D(n/2) + O(\log n) \implies D(n) = O(\log^2 n)$$

7.3 Parallel Partitioning Quiz

The par-for loop is not safe because the assignment of the next index of the buffer $k \leftarrow k + 1$. Recall mutex in CS 6200! Hence the key to make quick sort parallel is to find a parallel way to determine the buffer index k.

7.4 Conditional Gathers Using Scan

The idea of making buffer index k determination parallel comes from the observation that, if we use boolean flag 1 to represent an element smaller than the pivot and 0 to represent an element larger than the pivot, a scan through the boolean flags would yield the order the smaller elements would be copied to the buffer, thus we can parallelize scan so as to parallelize quick sort.

$$\begin{aligned} &F[:] \leftarrow (A[:] \le \text{pivot}) \\ &K[:] \leftarrow \text{addScan}\,(F[:]) \\ &\text{parfor } i \leftarrow 1 \text{ to } n \text{ do} \\ &\qquad \text{if } F[i] = 1 \text{ then } L[K[i]] \leftarrow A[i] \end{aligned}$$

Since the flagging operation is one time with work n and span 1, the work and span of this conditional gather if algorithm are that of parallel span i.e. $W(n) = O(n)$ and $D(n) = O(\log^2 n)$. Because there are $\log n$ levels of conditional gather if, the work of parallel quick sort is $W(n) = O(n \log n)$ and the span is $D(n) = O(\log^3 n)$.

7.5 Segmented Scans

The idea is to define a 2-tuple so as to absorb the segment-start condition check into a scan operation $(a_i, f_i) \circ (a_j, f_j) = (a_i + a_j, f_i \vee f_j)$ (where $\vee$ is logical OR)

$$\begin{aligned}&\text{if } f_j\text{: return } (a_i + a_j, f_i \vee f_j);\\ &\text{else: return } (a_j, f_j);\end{aligned}$$

this $\circ$-scan operation satisfies associativity—the only condition to qualify as scan operation (see below).

7.6 Scan Ingredients Quiz

The only requirement to qualify an operation as a scan operation is associativity. Commutativity is a stronger condition than associativity which is not required for a scan operation, and the cost of the scan operation being $O(1)$ only affects the work and span of the scan.

7.7 List Ranking Quiz

The idea is to use another flag-like list (rank increments) to make the list ranking into a prefix sum.

7.8 Linked Lists as Array Pools

The essential difference between linked list and array is that, the former has only a single entry point and there is no way to get other elements without traversing them one by one, whereas the later allows random access to any element. The idea to make a linked list random accessible is to use two arrays, one for storing the value at current node and the other for storing the index of next node ($\Rightarrow$ linked list nodes need these two pieces of information to record anyway).

7.9 A Parallel List Ranker

Combine the three ideas

1. store list as array pool to enable random access
2. use 2-tuple to convert list ranking into +-scan operation
3. employ jump primitive to divide and conquer

However jump will disturb the rank in the 2-tuple used for list ranking conversions. To solve this problem we need to update the ranks by pushing the rank of the jumped-out node to the next node (as jump-out effectively cuts off the node from the sublist). These updates can be done in parallel.

7.10 How Many Jumps Quiz

A jump step reduces the list into two sublists, which is analogous to binary tree recursion. Thus the maximum number of jumps is the same as the binary tree depth $O(\log n)$

7.11 A Parallel List Ranker Quiz

Wyllie's algorithm for parallel list ranking is

parfor $i \leftarrow 1$ to $\lceil \log n \rceil$ do
    updateRanks(rank_1[:], rank_2[:], next_1[:])
    jumpList(next_1[:], next_2[:])
    swap(rank_1[:], rank_2[:]); swap(next_1[:], next_2[:])

The span is $O(\log^2 n)$ because the par-for outer loop does logarithmic sequential steps, and each of the update ranks and jump list operations consist of par-for which have logarithmic span.
The work is $O(n \log n)$ because the par-for outer loop contributes the logarithmic part, and since the update ranks and jump list operations visit every node in every iteration they have linear cost. Hence this parallel list ranker is not work optimal because the naive sequential algorithm only has linear cost.

8 Tree Computations

8.1 Tree Warm Up

The running time is $O(n)$ as the extremely unbalanced tree is a single line. A tree can be parallelized by pointing to grandparent instead of parent ($\Rightarrow$ analogous to the jumpt list for linked list ranking).

8.2 Parallel Root Finder Quiz

The parallel root finder algorithm

parfor $l \leftarrow 1$ to $\lceil \log n \rceil$ do
    adopt $(P_{\text{cur}}[:], P_{\text{next}}[:])$
    $P_{\text{cur}}[:] \leftarrow P_{\text{next}}[:]$

where the adopt step implements the parent-to-grandparent pointer jumping

parfor $i \leftarrow 1$ to n do
    if hasGrandparent$(i, P[:])$
    then $G[i] \leftarrow P[P[i]]$
    else $G[i] \leftarrow P[i]$

Since both the outer sequential loop and the inner par-for loop of the adopt step are polylogarithmic, the span is polylogarithmic ($\Rightarrow O(\log^2 n)$?). However because of the outer sequential loop this algorithm is not work optimal ($\Rightarrow O(n \log n)$?).

8.3 Work-Optimal List Scan/Prefix-Sum - Part 1

The trick of improved work-suboptimal Wyllie $O(n \log n)$ towards sequential $O(n)$ is to use shrink list to size $m < n$ (consisting of sublists instead of nodes) and apply Wyllie $O(m \log m)$. Assume that this intermediate result can be extended to full list, with $m = n/\log n$ the work is $O(n)$.

8.4 Parallel Independent Sets - Part 1

To shrink a list in parallel a handy trick is to use **independent set**—the set I of vertices such that next$[i] \notin I \ \forall i \in I$

8.5 Parallel Independent Sets - Part 3

For the randomized parallel independent set generation, the work is $O(n)$ because both the coin flipping loop and the double head removing loop goes linearly from 1 to n, and both of them have constant amount of work per iteration, the span is $O(\log n)$ by the standard par-for parallelization of the two loops—a par-for loop with a constant amount of work per iteration has logarithmic span $\Rightarrow O(\log n + \log n)$.

8.6 Parallel Independent Sets - Part 4

The average number of vertices that end up in the independent set is $n/4$, because there are four possibilities (HH, HT, TH, TT) after the coin flipping loop, and a correction is made to HH so the possibilities are now (TH, HT, TH, TT). Hence there is only one out of four possible configurations that has the first node of the pair as H.

8.7 Work-Optimal List Scan/Prefix-Sum - Part 2

The trick for a work-optimal list ranking algorithm consists of three steps: (1) shrink the list to size $m = n/\log n$ (2) run Wyllie algorithm on the shrunk list $O(m \log m) = O(n)$ (3) restore the full list and ranks.
The shrinking is done via removing independent set as follows:

1. initialize rank as in Wyllie algorithm (i.e. 0 for head and 1 for others)
2. identify an independent set via coin flipping + double head removing loops
3. remove the independent set via jumping over it and pushing its rank to the next node
4. repeat step #2 and #3 until the size is smaller than $n/\log n$

The rank pushing ensures that the ranks of the remaining nodes will be the same as that produced by the sequential algorithm.

8.8 Work-Optimal List Scan/Prefix-Sum - Part 3

Recall the previous analysis that the average number of vertices that end up in the independent set is $n/4$. So to shrink the list size to $n/\log n$ the number of times we need to execute the shrinking step k must satisfy

$$\left(\frac{3}{4}\right)^k n \leq \frac{n}{\log n} \implies \left(\frac{4}{3}\right)^k \geq \log n \implies k \geq \frac{\log\log n}{\log(4/3)}$$

Therefore the number of times to remove independent set to shrink the list is $O(\log\log n)$ and as a result the work and span of this algorithm is not $O(n)$ and $O(\log n)$ but $O(n \log\log n)$ and $O(\log n \cdot \log\log n)$ ($\Rightarrow$ cf. the work and span of Wyllie on the full list is $O(n \log n)$ and $O(\log n \cdot \log n)$).

8.9 The Euler Tour Technique

The analogy of and representation by a linked list is grounded on the fact that every node has equal number of inbound edges as that of outbound edges (the "Euler" node). To convert the tree to a linked list replace the undirected edges with the directional traversals.
Assigning 0 to every parent-to-child sink and 1 to every child-to-parent sink and then perform a scan, the resulted prefix sum is exactly the same as the post-order numbers.

8.10 The Span of an Euler Tour Quiz

Suppose the prefix sum operation is work-optimal i.e. $W(n) = O(n)$. Then the work of an Euler tour is linear i.e. the work is $O(n)$ as well. In addition we know that the span of work-optimal prefix sum is $O(\log n)$. Thus is the span of an Euler tour also $O(\log n)$? The answer is yes, and it stems from the fact that the Euler tour operates on a linked list. Furthermore once the conversion is done the tree shape no longer matters. However all this hinges on being able to convert a computation into a Euler tour linked list, which is not always possible.

8.11 Computing Levels Quiz

To make the prefix sum equal the tree level, notice that an edge that goes from parent to child represents one level increment in tree whereas an edge that goes to child to parent represents one level decrease in tree. Hence it is natural to assign a node value $+1$ to every parent-to-child sink and -1 to every child-to-parent sink. Then the prefix sum at each child-to-parent source node is the tree level.

8.12 Implementing Euler Tours

The two key tools to implement Euler tours are

adjacency list: it is the set of all outgoing neighbors, let v be the node of interest, denote u_i to be its outgoing neighbors ordered by tree traversal, the adjacency list of v is denoted by $\text{adj}(v) = \{u_0, u_1, \ldots, u_{d_v-1}\}$ where d_v is the number of v's outgoing neighbors

successor function: defined as $s(u_i, v) = (v, u_{(i+1) \bmod d_v})$, the modulo in the subscript effectively makes the adjacency list circular, the significance of this function is that you keep applying the successor function you will end up traversing the entire Euler circuit. This is at least plausible by observing that the successor function flips the relative position of u and v, which models the two directed edges that point to opposite directions between a pair of nodes ($\Rightarrow (u_0, v) \to (v, u_1) \to (u_2, v) \to (v, u_0)$ if $d_v = 3$)

The remaining important implementation detail is to show that the cost of applying the successor function is constant $O(1)$. To show this we need to augment the adjacency lists with all cross edges.

8.13 Conclusion

There are two ways to build parallel tree algorithms: (1) build on top of work optimal lists (2) build on top of compressed framework for evaluating expression trees. The common idea behind these two methods is linearization. Linearizing the tree helps to achieve work balance which is needed to make a parallel algorithm scale.

9 Shared Memory Parallel BFS (Optional)

9.1 BFS 101

Breadth-first search (BFS) is useful to finding the fastest route through a network. Each successive advancement vertices being examined is called a frontier. The number of iterations of the while loop over the set of visited nodes F should be no more than the number of vertices $|V|$. And since each vertex appears in F at most once, we will visit each edge at most once if the graph is directed or twice if the graph is undirected. It means that the inner for loop over the neighbors of a given node will be executed at most $O(|E|)$ times. Hence the work of the sequential BFS is $O(|V| + |E|)$.

9.2 Analysis - Is BFS Inherently Sequential?

The problem is associated with the while loop—each time we pull a vertex out of the set of visited nodes F to examine its neighbors we insert vertices back to F. As such the span is $O(|V|)$. This makes the available parallelism $W/D = O(1 + |E|/|V|)$, which is bad because in real life we expect graphs to be sparse i.e. $|E| = O(|V|)$.

9.3 Intuition - Why We Might Do Better

The fact that BFS visits graphs in waves ($\Rightarrow$ wavefronts) has two important implications

- the upper bound on the span should be the number of waves called **levels** instead of the number of vertices, a level is all the vertices equidistant to the source
  the **diameter** of a graph is the maximum shortest distance between any pair of vertices, it is a property of the whole graph and an upper bound on the number of levels of any starting vertex, level-synchronous traversal is visiting the nodes level by level
- during a level-synchronous traversal the order of visiting the nodes at a given level should not matter, in fact they can be visited simultaneously (parallelism!)

9.4 The Diameter of a Graph Quiz

Note that diameter might be greater than the number of levels ($\Rightarrow$ just imagine the distance from the leftmost point of level x to the the rightmost point of level y is roughly $x + y$).

9.5 High-Level Approach to Parallel BFS

Two key ideas

- BFS should be carried out level by lever rather than vertex by vertex
- BFS should process the entire level in parallel

Now since the while loop iteration is over levels, the span as defined by the while loop is bounded by the diameter of the graph. However to implement this level-synchronous traversal we need a new data structure called bag.

9.6 Bags - Key Properties

A **bag** has the following data properties

- it is an unordered collection
- it allows repetition, which permits redundant insertion of the same vertex if necessary

A bag should allow the following operations

- fast enumeration of elements, which permits fast traversal in a level
- fast and logically associative union and split (i.e. $A \cup B == B \cup A$), which permits the application of reducer hyper objects (divide and conquer).

9.7 Pennants - Building Blocks for Bags

To implement a bag we need a **pennant**, which is a tree with a unary root having a child that is the root of a complete binary tree. As such a pennant has $1 + (2^k - 1) = 2^k$ nodes.
If two pennants are of exactly the same size $|X| == |Y|$, then they can be combined by choosing one of the roots as the root of the combined pennant and the other its unary child. The reverse—splitting a pennant into two pennants—can be done by reversing the above steps.

9.8 Pennant Combination Quiz

We cannot combine pennants of different sizes ($\Rightarrow$ mathematically we cannot find three distinct integers i, j, k to satisfy $2^i + 2^j = 2^k$).

9.9 Combining Pennants into Bags

We use the binary representation of n which is the number of elements in the bag. Each bit corresponds to a 2^k-sized pennant. To connect these pennants we use an array of pointers pointing to the root of each pennant. If the bit is 0 we use a null pointer. This pointer array is called **spine**.

9.10 Duality between Bags and Binary Math

The insertion begins from the least significant bit (LSB). A bit 1 at the kth significant bit represents a 2^k pennant. If the bit is occupied i.e. equals to 1, then combine the pennant it represents with the inserted pennant to form a larger pennant 2^{k+1}, which should then be carried over to next bit. Repeat the process until an unoccupied bit is reached and a pennant is moved into it.

9.11 What is the Cost of Insertion

By the fact that an integer n needs $\lceil \log n \rceil$ bits to store it (integer duality), the spine has a length of $\lceil \log n \rceil$. Because combining two pennants is just a matter of reshuffling two pointers, a single combine operation takes $O(1)$ time. Thus the cost of insertion is $O(\log n)$.

9.12 What is the Cost of Union

Again using binary representation, combining two bags each of size n is inserting elements in each of the bits. Because there are $\lceil \log n \rceil$ bits and each combination takes $O(1)$ time, the cost of union is still $O(\log n)$. This means that it takes the same amount of asymptotic time to insert n elements as it does to insert one, and it implies that the amortized time to insert an element is constant.

9.13 What is the Cost of Splitting

It is still $O(\log n)$ ($\Rightarrow$ otherwise time reversal symmetry breaking).

9.14 Bag Splitting

Recall that division by 2 can be represented by 1-bit right shift in binary form. In the worst case all $\lceil \log n \rceil$ bits of the spine are occupied thus $\lceil \log n \rceil$ bit moves is needed. Each bit move corresponds to a pennant split which takes constant time (again reshuffling two pointers as for union $\Rightarrow$ in fact opposite shuffle). Therefore the cost is $O(\log n) \times O(1) = O(\log n)$.

9.15 Finishing the Parallel BFS with Bags

If the bag is big enough we divide and conquer. Otherwise we use the sequential method.

```
parallelBFS(G = (V, E), s ∈ V)
    d[v] ← ∞
    d[s] ← 0
    l ← 0
    f_0 ← {s}
    while f_l ≠ ∅ do {
        f_{l+1} ← {}
        processLevel(G, f_l, f_{l+1}, d)
        l ← l + 1 }
    return d

processLevel(G, f_l, f_{l+1}, d)
    if |f_l| > φ_0 then
        (A, B) ← bagSplit(f_l)
        spawn processLevel(G, A, f_{l+1}, d)
        processLevel(G, B, f_{l+1}, d)
        sync
    else
        for v ∈ f_l do {
            par-for (v, w) ∈ E do {
                if d(w) = ∞ then
                    d(w) ← l + 1
                    bagInsert(f_{l+1}, w)
            }
        }
```

The par-for in the sequential method would lead to data race because each task tries to update the neighbors. But it is perfectly safe here because each task is writing exactly the same distance.
This algorithm is work-optimal (ref. Leiserson's paper) thus $W = O(|V| + |E|)$. The span is given by

$$D = \text{number of levels} \times \text{span of process level}$$

where span of process level consists of three parts

$$\begin{aligned}\text{span of process level} &= \text{recursion depth} \times \text{splitting} \times \text{sequential base case} \\ &= O\left(d \times \log^r(|V| + |E|)\right)\end{aligned}$$

10 Intro to Distributed Memory Models

10.1 A Basic Model of Distributed Memory

In a distributed memory machine model, a machine is a collection of nodes connected by some kind of network, each node consists of a processor connected to a private memory. The private nature of memory implies that to share data nodes need to send messages to one another. This parallel communication is called message-passing. The rules of this model are

1. the network is *fully connected*—every node in the network is reachable from every other node, but the shortest path between any two nodes could be longer than one edge (later on in the Topology lesson, we'll see the phrase "fully connected" refer to the situation where there is a direct link between every pair of edges, and the longest path between any two nodes is one edge)

2. the network links are bidirectional—the link can carry a message in both directions at the same time

3. a node can concurrently perform at most 1 send and 1 receive at the same time

4. the cost of sending a message is linear in the message size $T_{\text{msg}}(n) = \alpha + \beta n$ but independent of the path taken, where α is the latency and β is the inverse bandwidth

5. the linear cost rule applies only when there are no messages competing for links, when there are k messages simultaneously competing for a link (called a k-way congestion), the cost becomes $T_{k\text{-congestion}} = \alpha + \beta n k$

10.2 Pipelined Message Delivery Quiz

Consider a linear network having P nodes. Suppose node 0 has a message of size n that needs to be transmitted. Break it up into n words and transmit them one by one. Assume the message preparation time is a and a word going from one node to another takes t time. The minimum time to send this n-word message is

$$T = a + \underbrace{t(P-1)}_{\text{1st word}} + \underbrace{t(n-1)}_{\text{remaining words}} = \underbrace{a}_{\text{startup overhead}} + \underbrace{t(P-2)}_{\text{wire delay}} + tn = \alpha + \beta n$$

$$\implies \alpha = a + t(P-2) \qquad \beta = 1/t$$

Typically a is orders of magnitude larger than t thus the first two terms are comparable.

10.3 Getting a Feel for the Alpha-Beta Model

With the alphe-beta model an algorithm has three costs—the message cost defines the α and β and the computation cost defines τ (cost per compute operation). In practice $\tau \sim 10^{-12}$, $\beta \sim 10^{-9}$, $\alpha \sim 10^{-6}$ thus $\tau \ll \beta \ll \alpha$.

10.4 Applying the Rules - Scenario 2 Quiz

Two messages going in opposite directions will not cause congestion.

10.5 Applying the Rules - Scenario 3 Quiz

Since the paths intersect and go in the same direction, the two messages will need to serialize in their bandwidth, and thus $T = \alpha + 2\beta n$ ($\Rightarrow$ no timing consideration here).

10.6 Applying the Rules - Scenario 4 Quiz

In a mesh we can overlap two messages by choosing proper routes for both of them.

10.7 Collective Operations - Part 1

In an all-to-one reduce all nodes participate to produce a final result on one node, e.g. summing all elements in an array. In a binary tree recursion of summing all elements in an array, since at every level all odd numbered nodes send their data to the lower even number in parallel without congestion, the communication time is $\alpha + \beta$. Hence the total communication time is $\lceil \log n \rceil (\alpha + \beta)$.
Notice that at the kth level the odd number nodes have bit 1 at the kth place and after sending the data that bit is dropped. This bit manipulation is related to Hamming distance, hypercube, and gray code.

10.8 Point-to-Point Communication Primitives

- Assume a single-program multiple-data (SPMD) style. Call every running copy of pseudocode a process, each of which runs independently and asynchronously. To distinguish them assume each has access to two global variables—(1) rank (the ID of the process) and (2) P (the number of processes).
- Assume an asynchronous send primitive, which has two arguments—(1) buffer of size n and (2) destination rank. It returns a handle which does not indicate that the buffer is sent thus the buffer should not be modified. For this send to complete the destination rank has to post an asynchronous receive. The syntax of asynchronous receive is identical to that of asynchronous send—taking buffer of size n and source rank as arguments and returning a handle.
- Assume a wait primitive, which takes one or more handles as argument, which is a blocking operation. A special case is wait all, which waits for all outstanding sends and receives.

10.9 Point-to-Point Completion Semantics

When wait returns, the only thing we know is that buffer is safe to reuse. For a receive it means that the message was delivered. However for a send it doesn't tell whether the send operation is completed. The precise meaning of safe-to-reuse for a send is implementation dependent, and in fact the send could arrive way before the receive is ready. Thus to ensure that a sent message is received, we need to prove that every send must have a matching receive. This protocol that sends match receives is called two-sided messaging.

10.10 Send and Receive in Action Quiz

There is an ambiguity about what completion means in the case of a send. Thus if both processes execute asynchronous send and then wait for both sends to complete, it may result in a deadlock.

10.11 All-to-One Reduce Pseudocode

```
bitmask ← 1
while bitmask < P do
    partner ← rank ∧ bitmask
    if (rank & bitmask i.e. sender)
        sendAsync(s → partner)
        wait(*)
        break   // once sent, drop out
    else
        recvAsync(t ← partner)
        wait(*)
        s ← s + t
    bitmask ← (bitmask ≪ 1)
if rank = 0 (final result is stored here called root) then print(s)
```

where $\wedge$ is exclusive OR, because partners differ only in the bitmask position, with 1 for sender and 0 for receiver.

10.12 All-to-One Reduce Pseudocode Quiz

The above pseudocode works only for $P = 2^k$. For more general cases we can patch the number of processes to a power of 2. But we also need to make the following change to ensure that the patched processes do nothing

$$\text{else} \rightarrow \text{else if (partner} < P\text{)}$$

This else if statement follows from the fact that the bit pattern of senders and receivers look like

$$\begin{aligned} \text{bitmask:} &\quad 00\ldots010\ldots0 \\ \text{sender:} &\quad ??\ldots?10\ldots0 \\ \text{receiver:} &\quad ??\ldots?00\ldots0 \end{aligned}$$

Therefore we have rank(receiver) < rank(sender) $< P$. Hence a candidate receiver whose partner has a rank greater than P cannot be valid.

10.13 Vector Reductions

A vector reduction will be element wise reduction of vectors. The only change to pseudocode needed is

$$\text{scalar version: sendAsync}(s \rightarrow \text{partner}) \quad \Longrightarrow \quad \text{vector version: sendAsync}(s[1:n] \rightarrow \text{partner})$$

10.14 Vector Reductions Quiz

The time to do vector reduction is $(\alpha + \beta n) \log P$ because the cost of sending the full vector is $\alpha + \beta n$ and there are $\log P$ rounds of communication.

10.15 More Collectives

broadcast: opposite to all-to-one reduce is one-to-all broadcast, in which the process that holds all the data sends a copy to all other processes, to achieve it we just need to run the reduction in reverse, and because of this we call them dual of each other

scatter: different from broadcast, the process that holds all the data sends a piece of the data to all other processes, it also has a natural dual called gather

all-gather: initially each process has a piece of the data, after all-gather all processes have a copy of all the data, it also has a natural dual called reduce-scatter—initially all processes contain a vector of data, which then globally reduce the vectors using some combining operator (vector addition or element wise multiplication) and the result is scattered to all processes

10.16 A Pseudocode API for Collectives

To implement collective operation, the program must be structured in such a way that all processes execute it. For example

reduce: reduce($A_{\text{local}}[1:n]$, root)

broadcast: broadcast($A_{\text{local}}[1:n]$, root)

gather: gather(In$[1:m]$, Out$[1:m][1:P]$, root) where $n = m \cdot P$

scatter: scatter(In$[1:m][1:P]$, root, Out$[1:m]$)

all-gather: allGather(In$[1:m]$, Out$[1:m][1:P]$)

reduce-scatter: reduceScatter(In$[1:m][1:P]$, Out$[1:m]$)

For the latter two 2D arrays are involved, which motivates the following reshape primitive

$$\text{reshape}\,(A[1:m][1:n]) \to \widehat{A}[1:m \cdot n]$$
$$\text{reshape}\,(A[1:m \cdot n]) \to \widehat{A}[1:m][1:n]$$

It is a purely logical operation rather than a physical operation because it is conventional to linearize higher-dimensional arrays column wise. Hence we can assume constant cost $O(1)$ for reshape.

10.17 All Gather - From Building Blocks Quiz

allGather can be implemented by gather(In, Out, root) followed by broadcast(reshape(Out), root).

10.18 Collectives: Lower Bounds

With tree reduction the cost of reduce is $T(n) = (\alpha + \beta n) \log P$. Can we do better than that? In the very best case a process can pair sends and receives, and if pairing it needs at least $\log P$ rounds of communication. Since the alpha term essentially measures the number of rounds, it is optimal. As for the beta term in the very best case we need to send a total of $n(P-1)$ words since there are $(P-1)$ processes (except root). If all the $(P-1)$ processes could send their data simultaneously, the lower bound on time with respect to beta would be βn. Similar reasoning applies to all the other collectives. Therefore the lower bound on communication for all of them is $T(n) = \Omega(\alpha \log P + \beta n)$.

10.19 All Gather Quiz

A constant number of optimal operations is still optimal.

10.20 Implement Scatter Quiz

The pseudocode of scatter is

```
if rank = root
    for i ≠ root do
        sendAsync(In[:][i], i)
else
    recvAsync(Out[:], root)
wait(*)
```

Recall that a process can do at most one send and one receive simultaneously. Hence for scatter we have to pay for $(P-1)$ sends each of which costs $\alpha + \beta m$. Therefore the best cost is $(\alpha + \beta m)P$.

10.21 Implementing Scatter and Gather - Part 2

The idea of optimizing scatter stems from the observation that we let the root perform all the sends, which is not at all efficient because it can only do one send at a time. Thus the obvious way to optimize it is to let other processes send as well. This can be achieved by dividing the data into half and send one half of it to the $P/2$th process (suppose the root is the 0th). For next round we do this divide and conquer again within the 0th to $(P/2-1)$th processes and the $P/2$ to P processes. Hence the total cost is

$$\begin{aligned} T(n) = \sum_{i=1}^{\log P} T_i &= \sum_{i=1}^{\log P} (\alpha + \beta n_i) \\ &= \sum_{i=1}^{\log P} \left(\alpha + \beta \frac{n}{2^i}\right) = \alpha \log P + \beta n \end{aligned}$$

This way we attained the lower bound with respect to both latency and bandwidth.

10.22 When to Use Tree Based Reduce

For tree-based reduction the cost is $T(n) = \alpha \log P + \beta n \log P$. It is still a reasonable algorithm when the alpha term dominates the beta term $\beta n \log P \ll \alpha \log P$, which implies $n \ll \alpha/\beta$.

10.23 What's Wrong with Tree-Based Reduce Quiz

There is too much redundant communication—at every one of the $\log P$ rounds it sends full n words.

10.24 Bucketing Algorithms for Collectives

If we use gather + broadcast for all-gather, although gather can achieve the optimal $\alpha \log P + \beta n$, a tree-based broadcast would cost $(\alpha + \beta n) \log P$. Thus when combined the lower bound would not be achieved. To achieve optimal cost, note that intuitively the beta term is fundamentally about using as many of the links as possible. This can be achieved by having each process sending n/P words to its immediate right neighbor, and for next round each carries the words received from its immediate left neighbor over to its immediate right neighbor (⇒ from the perspective of words letting it traverse

through all the processes is equivalent to filling it into all the processes). This way there will be only $(P-1)$ rounds of communication and each costs $\alpha + \beta n/P$ ($\Rightarrow \approx n$ words sent total). Hence the total cost is

$$T(n) = \left(\alpha + \beta \frac{n}{P}\right)(P-1) \approx \alpha P + \beta n$$

Note that the beta term is optimal, but the alpha term is not since the factor is P not $\log P$. However it is a reasonable algorithm if the beta term dominates the alpha term $\beta n \gg \alpha P$, which occurs when $n \gg (\alpha/\beta)P$.

10.25 Bandwidth Optimal Broadcast

Use bandwidth optimal scatter + all-gather. The pseudocode is as follows

$B[1:m][1:P] \leftarrow \text{reshape}(A)$
$T[1:m] = \text{temporary array}$
$\text{scatter}(B, \text{root}, T)$
$\text{allGather}(T, B, \text{root})$
$A \leftarrow \text{reshape}(B)$

10.26 All-Reduce

Different from all-to-one reduce, all-reduce puts a copy of the reduction result on all the processes. To achieve a bandwidth optimal all-reduce, we can combine bandwidth optimal reduce-scatter and all-gather.

10.27 Conclusion

The message passing model has at least one major weakness—it forces programmers to keep track of many things like who and how many processes there are, when and how processes should communicate etc.

11 Topology

11.1 Intro to Network Models: Links and Diameter

link: number of connections ($P-1$ for linear network, $2(\sqrt{P}-1)\sqrt{P} \approx 2P$ for 2D mesh, $P(P-1)/2$ for fully-connected network), denoted by λ, significant because it is a proxy for cost

diameter: the longest of all the shortest paths between all pairs of nodes ($P-1$ for linear network, $2(\sqrt{P}-1)$ Manhattan distance for 2D mesh, 1 for fully-connected network), denoted by Δ, significant because it is a proxy for the maximum distance that any message must travel in the absence of network contention

11.2 Bisection (Band)Width

Bisection width is the minimum number of communication links that you have to take out in order to cut the network into two equal parts measured by the number of nodes. $B(P) = 1$ for linear network, $B(P) = 2$ for ring network, $B(P) = \sqrt{P}$ for 2D mesh, $B(P) \approx P^2/4$ for fully-connected network (since bisected networks have $P^2/4$ links each). It is significant because for all-to-all personalized exchange where every node sends private data to every other node, thus equivalently every node in the sending

half is communicating to every node in the receiving half, and all messages have to go through the bisection. Thus networks with larger bisection widths have a better capacity for carrying such traffic. **Bisection bandwidth** is the produce of bisection width and link speed $1/\beta$ (words/time) if all the links have the same speed, or the minimum of such products along bisection cuts.

11.3 Improve the Bisection of a 2D Mesh

Although all of them cut the diameter by half, only the 2D torus doubles the bisection width. Both the diagonal links and the peripheral links increase the bisection width by 2.

11.4 Some Other Network Topologies

ring: $\lambda(P) = P$, $\Delta(P) = P/2$, $B(P) = 2$

tree: leaves are compute nodes vs. internal nodes are routers, $\lambda(P) = P$, $\Delta(P) = \log P$, $B(P) = 1$, the scaling of links and diameters are good but the scaling of bisection width is terrible, thus in practice network designers typically fatten up bisection width as we move towards the top of the tree (e.g. more wires for higher levels of tree), this variant is called fat tree

d-dim torus: $\lambda(P; d) = dP$, $\Delta(P; d) \approx dP^{1/d}/2$, $B(P; d) = 2P^{(d-1)/d}$, d-dim tori and meshes are very important in practical high-end computing systems, in fact many of the world's top supercomputers use low-dimensional toroidal networks

hypercube: roughly speaking a $\log P$-dimensional torus, more specifically

	d-dim torus	d-dim hypercube
link	$\lambda_t(P; d) = dP$	$\lambda_h(P; d) = P \log P$
diameter	$\Delta_t(P; d) = dP^{1/d}/2$	$\Delta_h(P; d) = \log P$
bisection	$B_t(P; d) = 2P^{(d-1)/d}$	$P(P; d) = P/2$

Compared to torus, hypercube is much more expensive in terms of links (number of wires), but it has a lower diameter and a much larger bisection width.
A d-dim hypercube consists of $P = 2^d$ nodes. It is constructed by repeatedly mirroring the lower-dimensional hypercube and connecting every node with its mirror image. If follows directly from the construction that the bisection width is $P/2$.

11.5 Mappings and Congestion

When you design an algorithm for a distributed memory system, the communication pattern of your algorithm implies a network. If the edges of the physical network is a *superset* of the edges of the logical network, then no contention for links of the logical network implies no contention for links of the physical network. For the opposite situation, define **congestion** of a logical-to-physical network mapping as the maximum number of logical edges that map to a given physical edge.

11.6 2D to 1D Congestion Quiz

For a 2D torus to 1D ring logical-to-physical network mapping, the congestion is $\sqrt{P} + 2$.

11.7 A Lower Bound on Congestion

The idea stems from the fact that a bisection in the physical network corresponds to a bisection in the logical network. Denote by B_X the number of edges bisected in the physical network and L the number of edges bisected by the corresponding cut in the logical network. Then by the definition of congestion $C \geq L/B_X$. Furthermore denote by B_L the bisection width of the logical network thus $L \geq B_L$. Hence the lower bound of the congestion is the ratio of the logical bisection width to the physical bisection width $C \geq B_L/B_X$. Apply it to the 2D to 1D congestion quiz above we have $C \geq 2\sqrt{P}/2 = \sqrt{P}$. This lower bound gives an estimate of how much worst the algorithm cost might be if running in a physical network.

11.8 Congestion Lower Bounds

The congestion lower bound must be < 1 to make a congestion-free logical-to-physical network mapping possible.

11.9 Exploiting Higher Dimensions

Performing an all-gather on a linear network using the bucketing scheme has an execution time $T(n;P) = \alpha P + \beta n$, which is suboptimal in the alpha term because the lower bound is $T(n;P) = \Omega(\alpha \log P + \beta n)$. Promoting the network to the higher dimensional 2D mesh allows the reduction of execution time as follows

1. a 1D all-gather within each row, when it completes each processor has a complete row of data
2. a 1D all-gather within each column, when it completes each processor has the complete data

Again apply the bucketing scheme, the row all-gathers and column all-gathers combined take (where $m = n/P$)

$$T_{\text{row}} = T(m\sqrt{P};\sqrt{P}) = \alpha\sqrt{P} + \beta m\sqrt{P} \qquad T_{\text{column}} = T(n;\sqrt{P}) = \alpha\sqrt{P} + \beta n$$

$$\implies T(n;P) = T_{\text{row}} + T_{\text{column}} = 2\alpha\sqrt{P} + \beta m(P + \sqrt{P})$$

Hence taking advantage of the additional capacity offered by a higher dimensional network gets closer the lower bound.

11.10 2D Broadcast Quiz

The alpha term cost of a tree-based algorithm is proportional to $\log P$. By contrast the alpha term cost of a bucketing algorithm is proportional to P. Thus even if we operate on a 2D mesh thus the bucketing algorithm is proportional to $\sqrt{P}$. It is still worse than $\log P$. Nonetheless which one has a larger beta term? ⇒ ref. SUMMA communication time quiz below

$$T_{\text{tree}} = O\left(\alpha \log P + \beta n \log P\right)$$
$$T_{\text{bucket}} = O\left(\alpha P + \beta n\right)$$
$$T_{\text{mesh}} = O(\alpha\sqrt{P} + \beta n)$$

11.11 All-to-all Personalized Exchange

Consider performing matrix transpose on a ring network. Let the send and receive size be m and the row/column contains $n = mP$ words. To set up a lower bound, consider the ith word the 0th column needs to send to the ith column. The average distance a word needs to travel is (it is min because we operate on a ring network)

$$\text{avg distance} = \frac{1}{P-1}\sum_{i=1}^{P-1}\min(i, P-i) = \frac{2}{P-1}\sum_{i=1}^{(P-1)/2} i = \frac{P}{4}$$

Therefore the total volume of the traffic is $P \cdot m(P-1) \cdot (P/4)$. Since the total bandwidth is the number of links P multiplied by the link speed $1/\beta$, the lower bound of the communication time is thus

$$T(n; P) \geq \left[P \cdot m(P-1) \cdot \frac{P}{4}\right] \div \frac{P}{\beta} = \beta n \frac{P-1}{4}$$

A algorithm to achieve this lower bound is circular shift similar to all-gather bucketing scheme. However here the amount of data sent by each processor is decremented for each round, with the first round every column sending $m(P-1)$ words to the next column, the second round sending $m(P-2)$ words etc. because in each round every column sends all the data that needs to be transposed, and after each round every column has m more words in the right place. Therefore the total time it takes is

$$T(n; P) = \sum_{i=1}^{P-1}[\alpha + \beta m(P-i)] = \left(\alpha + \beta m\frac{P}{2}\right)(P-1) = \alpha(P-1) + \beta n\frac{P-1}{2}$$

The beta term is within a factor of 2 of the lower bound.

11.12 All-to-all Higher Dimensions Quiz

The network that has the best chance of reducing the asymptotic running time to $O(\alpha \log P + \beta n \log P)$ is hypercubes and fully-connected networks. Because an all-to-all personalized exchange is intrinsically limited by bisection width, and only hypercubes and fully-connected networks have linear or better bisection that is required to reduce time from $O(\alpha P + \beta n P)$ to $O(\alpha \log P + \beta n \log P)$.

11.13 Conclusion

The two big ideas of this lesson are: (1) congestion (2) exploit higher-dimensional networks.

12 Distributed Dense Matrix Multiply

12.1 Matrix Multiply: Basic Definitions

```
parfor i ← 1 to m do
    parfor j ← 1 to n do
        let T[1 : k] be temporary array
        parfor l ← 1 to k do
            T[l] ← A[i, l] · B[l, j]
        C[i, j] ← C[i, j] + reduce(T[:])
```

The work and span are $W(n) = O(n^3)$ and $D(n) = O(\log n)$ if $m = k = n$.

12.2 A Geometrical View

Imagine a cuboid where the faces are the matrix operands $C \leftarrow C + A \cdot B$ with A in front, B on top, and C on the right side. The row of A and the column of B that are involved in the computation of c_{ij} is the projection of the tube/tunnel through the (i, j)-th position on the right side onto the front and the top surfaces, with each of the $\sum_k a_{il} b_{lj}$ dot product multiplications corresponding to an internal point of the tube. This can be generalized to any cubic volume inside the cuboid, which corresponds to a submatrix multiplication. The number of multiplications is equal the cubic volume, and the size of the submatrix is $\sqrt{r \cdot s \cdot t}$. More generally by the Loomis-Whitney inquality in discrete geometry, the volume of an internal block I satisfies the following inequality

$$|I| \leq \sqrt{|S_A| \cdot |S_B| \cdot |S_C|}$$

where S_A, S_B, and S_C are the areas of the projection of I onto surface A, B, and C, and $|\cdot|$ represents the corresponding volume and area measures.

12.3 Applying Loomis-Whitney Quiz

The minimum interior volume occurs when there is no intersection among the three areas on the surfaces. The maximum interior volume occurs when there is optimal alignment among them.

12.4 1D Algorithms

Consider distributing the operands by block rows i.e. giving each of the P processors n/P consecutive rows of each matrix operand.

let $\widehat{A}[1 : n/P][1 : n]$ be a local part of A, same for $\widehat{B}$, $\widehat{C}$
let $\widetilde{B}[1 : n/P][1 : n]$ be a temporary storage of $\widehat{B}$
let $r_{\text{next}} \leftarrow (\text{rank} + 1) \bmod P$
let $r_{\text{prev}} \leftarrow (\text{rank} + P - 1) \bmod P$
for $L \leftarrow 0$ to $P - 1$
    $\widehat{C}[:][:] += \widehat{A}[:][\cdots L \cdots] \cdot \widehat{B}[\cdots L \cdots][:]$
    sendAsync$(\widehat{B} \to r_{\text{next}})$
    recvAsync$(\widetilde{B} \to r_{\text{prev}})$
    wait(*)
    swap$(\widehat{B}, \widetilde{B})$

12.5 1D Algorithm Cost Quiz

Let τ be the time per flop (i.e. one multiplication or one addition), the total computation time is ($\Rightarrow$ $2(n/p \times n \times n/p)$ per process $\times$ p processes)

$$T_{\text{comp}}(n; P) = \frac{2\tau n^3}{P}$$

Since each temporary storage $\widetilde{B}$ contains n^2/P words, and there are P rounds of circular shift (recall $L = 0, \ldots, P - 1$), the total communication time is

$$T_{\text{comm}}(n; P) = \alpha P + \beta n^2$$

12.6 1D Algorithm Cost Part 2 Quiz

We can rearrange the above pseudocode so that communication and computation overlap

$$
\begin{aligned}
&\text{for } L \leftarrow 0 \text{ to } P-1\\
&\quad \text{sendAsync}(\widehat{B} \to r_{\text{next}})\\
&\quad \text{recvAsync}(\widetilde{B} \to r_{\text{prev}})\\
&\quad \widehat{C}[:][:] += \widehat{A}[:][\cdots L \cdots] \cdot \widehat{B}[\cdots L \cdots][:]\\
&\quad \text{wait}(*)\\
&\quad \text{swap}(\widehat{B}, \widetilde{B})
\end{aligned}
$$

so that the total execution time is reduced to

$$T_{\text{1D}}(n;P) = \frac{2\tau n^3}{P} + \alpha P + \beta n^2 \xrightarrow{\text{overlap}} \max\left(\frac{2\tau n^3}{P}, \alpha P + \beta n^2\right)$$

12.7 Efficiency and the 1D Algorithm

Since the best sequential algorithm pays only for flops, $T_*(n) = 2\tau n^3$. **Parallel efficiency** is defined as the speedup divided by the number of processors and thus is given by

$$
\begin{aligned}
E(n;P) = \frac{S(n;P)}{P} = \frac{T_*(n)}{P \cdot T_{\text{1D}}(n;P)} &= \frac{2\tau n^3/P}{\max\left(2\tau n^3/P, \alpha P + \beta n^2\right)}\\
&= \frac{1}{\max\left(1, \dfrac{\alpha P^2}{2\tau n^3} + \dfrac{\beta P}{2\tau n}\right)}
\end{aligned}
$$

A parallel system is efficient if its parallel efficiency is a constant, and the higher constant the better which corresponds to higher fraction of linear speedup. From the above expression $E(n;P) = O(1) > 0$ if and only if $n = \Omega(P)$, which implies that doubling the number of processors requires doubling the problem dimension and increasing the number of flops by a factor of 8. This function $n = \Omega(P)$ has a special name called **isoefficiency function**, which is a function of P that n has to satisfy in order to have constant parallel efficiency.
Due to the need of temporary storage $\widetilde{B}$, the memory required for the 1D block row algorithm is

$$M(n;P) = (3+1)\frac{n}{P} \times n = \frac{4n^2}{P}$$

12.8 Isoefficiency Quiz

Let τ be the time per scalar addition. The execution time for a tree-based all-to-one reduction is

$$T_{\text{tree}}(n;P) = \tau n \log P + \alpha \log P + \beta n \log P$$

The parallel efficiency is given by

$$
\begin{aligned}
E(n;P) = \frac{T_*(n)}{P \cdot T_{\text{tree}}(n;P)} &= \frac{\tau n P}{P \cdot (\tau n \log P + \alpha \log P + \beta n \log P)}\\
&= \frac{1}{\left(1 + \dfrac{\beta}{\tau}\right)\log P + \dfrac{\alpha}{\tau}\log P}
\end{aligned}
$$

Note that there is no isoefficiency function $n = f(P)$ that can make the denominator constant, because $(1 + \beta/\tau)\log P \to \infty$ as $P \to \infty$.

12.9 A 2D Algorithm SUMMA

The SUMMA algorithm iterates over vertical strips of A and the corresponding horizontal strips of B i.e. for $l \leftarrow 1$ to n/s, and the processor that owns the current strip needs to broadcast to all relevant processors, i.e. it needs to propagate the strip along the row of A and the column of B. Since each strip has a dimension $n/\sqrt{P}$ by s, the total computation time is

$$T_{\text{SUMMA}}(n; P, s) = \frac{n}{s} \times \left(2\tau \frac{n^2 s}{P}\right) = \frac{2\tau n^3}{P}$$

12.10 SUMMA Communication Time Quiz

The communication all comes from broadcasting the strips along the rows of A and the columns of B. Recall that there are two schemes of broadcast, the communication time of which are

$$\begin{aligned} T_{\text{tree}} &= O\left(\alpha \log P + \beta m \log P\right) \\ T_{\text{bucket}} &= O(\alpha P + \beta m) \end{aligned}$$

Since $n/\sqrt{P} \cdot s$ is the number of words contained in one strip, the total number of words broadcast $m = (ns/\sqrt{P}) \cdot (n/s) = n^2/\sqrt{P}$ as there are $l \leftarrow 1$ to n/s broadcasts. Hence the SUMMA communication time is

$$\begin{aligned} T_{\text{tree}} &= O\left(\alpha \frac{n}{s} \log P + \beta \frac{n^2}{\sqrt{P}} \log P\right) \\ T_{\text{bucket}} &= O(\alpha \frac{n}{s} P + \beta \frac{n^2}{\sqrt{P}}) \end{aligned}$$

12.11 Efficiency of 2D SUMMA

The 2D SUMMA algorithm is intrinsically more scalable than the 1D block row algorithm, because the strip width $1 \leq s \leq n/\sqrt{P}$ is a tuning parameter of the algorithm. For the tree-based broadcast the parallel efficiency is (let $s = 1$)

$$E_{\text{tree}}(n; P) = \frac{1}{1 + \frac{1}{2}\frac{\alpha}{\tau}\frac{P \log P}{n^2} + \frac{1}{2}\frac{\beta}{\tau}\frac{\sqrt{P} \log P}{n}}$$

Hence the isoefficiency function is $n_{\text{tree}}(P) = \Omega(\sqrt{P} \log P)$, which is better than the 1D isoefficiency function $n_{\text{1D}}(P) = \Omega(P)$ as it scales more slowly with P.
The bucket-based broadcast has a slightly worse isoefficiency function $n_{\text{bucket}} = \Omega(P^{5/6})$, because it trades a higher latency cost for a lower communication cost.

12.12 SUMMA Memory Quiz

The required memory is

$$M_{\text{SUMMA}} = \underbrace{3\left(\frac{n}{\sqrt{P}}\right)^2}_{\text{3 subblocks of } A,\, B,\, C} + \underbrace{2\frac{n}{\sqrt{P}}s}_{\text{2 broadcast strips}} < 4\frac{n^2}{P} = M_{\text{1D}} \quad \text{for } s < \frac{1}{2}\frac{n}{\sqrt{P}}$$

Keep in mind however that s appears in the denominator of the latency term, thus there is a trade-off between memory and latency.

12.13 A Lower Bound on Communication

Suppose each processor has M words of memory, and it performs W multiplications. Divide the entire operation timeline into L phases and assume in each phase every processor sends and receives exactly M words except for the last phase due to indivisibility. Consider one of the phases. Let S_A be the number of unique elements of A seen during this phase and similarly for S_B and S_C. Since we assume during the phase the processor sends and receives exactly M words, we have $|S_A| \leq 2M$, $|S_B| \leq 2M$, $|S_C| \leq 2M$. By Loomis-Whitney inequality the maximum number of multiplication this phase can possibly do is

$$\#\text{ multiplies per phase} \leq \sqrt{|S_A| \cdot |S_B| \cdot |S_C|} \leq 2\sqrt{2}M^{3/2}$$

$$\implies \#\text{ phases } L \geq \left\lfloor \frac{W}{2\sqrt{2}M^{3/2}} \right\rfloor \geq \frac{W}{2\sqrt{2}M^{3/2}} - 1$$

$$\implies \#\text{ words communicated per processor} \geq M\left(\frac{W}{2\sqrt{2}M^{3/2}} - 1\right) = \frac{W}{2\sqrt{2}M^{1/2}} - M$$

Since $W \geq n^3/P$, and we distribute the words evenly over all processors thus $M = \Theta(n^2/P)$, we have

$$\implies \#\text{ words communicated per processor} = \Omega\left(\frac{n^2}{\sqrt{P}}\right)$$

Hence the lower bound of the beta term is $\beta n^2/\sqrt{P}$.

12.14 A Lower Bound on Communication Quiz

Since the largest message a processor can send is $M = \Theta(n^2/P)$, and the words sent by a processor has a lower bound $\Omega(n^2/\sqrt{P})$, the number of messages sent by a processor has the following lower bound

$$\#\text{ messages per processor} \geq \Theta\left(\frac{n^2/\sqrt{P}}{n^2/P}\right) = \Theta\left(\sqrt{P}\right)$$

Therefore the lower bound of the alpha term is $\alpha\sqrt{P}$, and the lower bound on the total communication time is

$$T_{\text{comm}}(n; P) = \Omega\left(\alpha\sqrt{P} + \beta\frac{n^2}{\sqrt{P}}\right)$$

12.15 Matching the Lower Bounds

Recall the communication time of the 1D block row algorithm is

$$T_{\text{1D, comm}}(n; P) = \alpha P + \beta n^2$$

and the communication time of the SUMMA algorithm is

$$T_{\text{SUMMA, comm}}(n; P) = \begin{cases} O\left(\alpha\frac{n}{s}\log P + \beta\frac{n^2}{\sqrt{P}}\log P\right) & \text{tree-based} \\ O\left(\alpha\frac{n}{s}P + \beta\frac{n^2}{\sqrt{P}}\right) & \text{bucket-based} \end{cases}$$

$$s \leq \frac{n}{\sqrt{P}} \implies T_{\text{SUMMA, comm}}(n; P) \geq \alpha\sqrt{P}\log P + \beta\frac{n^2}{\sqrt{P}}$$

Compared with the lower bound above the SUMMA algorithm is off only by the alpha term.
There is an algorithm called Cannon's algorithm that is even better than SUMMA and exactly matches

the lower bound of the communication time. In fact the lower bound can be beaten because in deriving it we made a critical assumption that each processor only has enough memory to store $M = \Theta(n^2/P)$ words. This assumption is akin to distributing surfaces of the cuboid across processors. This brings in the question of whether having more memory could reduce communication. One example of such scheme is a 3D algorithm, which distributes processors in a 3D mesh of size $P^{1/3}$. The memory required is increased by a factor of $P^{1/3}$ whilst the communication time is reduced by the same factor

$$M_{3D} = M_{2D} \cdot P^{1/3} \qquad T_{3D} = \frac{T_{2D}}{P^{1/3}}$$

13 Distributed Memory Sorting

13.1 Distributed Bitonic Merge via Binary Exchange

For block distributed bitonic merge communication happens whenever a dependence edge crosses a processor boundary. The communication pattern is called *binary exchange* because there are two processors swapping data with one another. Notice that edges cross processor boundaries only up to $\log P$ stages. In the remaining $\log(n/P)$ stages there is local computation but no communication ($\Rightarrow$ recall local sort takes $\log m$ steps where m is the number of elements).
For cyclically distributed bitonic merge the communication vs. computation phases are similar to those of block distribution, except that the $\log(n/P)$ stages of computation phase precedes the $\log P$ stages of communication phase. As to the communication volume, each processor sends n/P words per stage. Hence the total volume is $n \log P$, for both block distribution and cyclic distribution.

13.2 Pick a Network Quiz

Assuming $P = n$, only hypercube and fully-connected network allow for fully-connected exchanges without congestion, because binary exchange mandates a linear or better bisection width. Hypercube is a good fit due to the analogy of its construction with the communication pattern. Actually there is a network named butterfly which has exactly the same connection pattern as binary exchange.

13.3 Communication Cost of a Bitonic Merge Quiz

Assuming a block distributed binary-exchange scheme on a hypercube, the communication time of a bitonic merge is

$$T_b(n; P) = \alpha \log P + \beta \frac{n}{P} \log P = T_c(n; P)$$

which is the same for cyclic distribution scheme, because there are $\log P$ communication stages and each processor sends n/P words per stage.

13.4 Bitonic Merge via Transposes

Since for the block distribution scheme the latter $\log P$ stages involve only local computation but no communication, whereas for the cyclic distribution scheme the first $\log P$ stages involve only local communication but no communication, we can stack a cyclic distribution on top of a block distribution and the communication takes place only on the stage of connecting the two (reshuffling), which is called *transpose*. Transpose can be viewed as an all-to-all personalized exchange, in which each processor sends $(P - 1)$ messages to each of the other processors. Assuming a fully connected network the communication time of the transpose is

$$T_{\text{trans}}(n; P) = \alpha (P - 1) + \beta \frac{n}{P^2} (P - 1)$$

where the message size is n/P^2 ($\Rightarrow$ total n/P words per processor, which will be distributed evenly over all P processors). Compared to block and cyclic distribution it trades a higher latency cost for a lower communication cost.
In practice it is typically very hard for the block or cyclic distribution scheme to beat the transposed scheme for typical values of n/P.

13.5 Bitonic Sort Cost: Computation

There are $\log n$ merge stages. Assume P processors, then the kth stage performs $k(n/P)$ computations ($\Rightarrow$ n/P elements per processor and k rounds of comparison for n/P elements). Therefore the total computation cost (where τ = cost per comparison)

$$T_{\text{comp}}(n;P) = \sum_{k=1}^{\log n} \tau \frac{n}{P} k = O\left(\tau \frac{n \log^2 n}{P}\right)$$

This result makes sense, because it confirms the fact that bitonic sort is not work optimal by a factor of $\log n$ (ref. the section on comparison sort).

13.6 Bitonic Sort Cost: Communication

Recall that the bitonic sort network is a sequence of bitonic merging stages, the merge size being $n_k = 2^k$ for the kth stage. Assume a block distribution scheme with P processors. Communication is necessary only when the merge size 2^k exceeds the number of elements per processor n/P, thus there is no communication for $k \leq \log(n/P)$. For the stages involving communication, the number of processors participating in the exchange is $P_k = 2^{k-\log(n/P)}$, which equals P for the last stage $k = \log n$. Therefore the total communication time is

$$\begin{aligned} T_{\text{comm}}(n;P) &= \sum_{k=\log(n/P)+1}^{\log n} T_{\text{merge}}(n_k;P_k) = \sum_{k=\log(n/P)+1}^{\log n} \left(\alpha + \beta \frac{n_k}{P_k}\right) \log P \\ &= O\left(\alpha \log P + \beta \frac{n}{P} \log^2 P\right) \end{aligned}$$

13.7 Linear Time Distributed Sort: Part 1

Any comparison based algorithm for sorting scales at best $O(n \log n)$ asymptotically. Bucket sort can achieve linear time $O(n)$ with the strong assumption that the values to sort are evenly distributed over the range of values. Buckets are then set up to evenly divide the values $[0, n/k-1], [n/k, 2n/k-1], \ldots [(k-1)n/k, n-1]$. If the values are evenly distributed over the range, then the expected number of elements per bucket will be $\Theta(n/k)$. Apply comparison based sorting to each bucket $O(n/k \log(n/k))$. Then the total sorting time is $O(n \log(n/k))$, which is linear $O(n)$ if $k = \Theta(n)$ ($k > n$ is disallowed thus there is some caveat involved in choosing the value of k, ref. supplement notes on bucket sort).

13.8 Distributed Bucket Sort Quiz

The sort consists of three steps

1. each processor scans its local elements and decides which appropriate processor to send them, the cost is $\Theta(\tau n/P)$

2. each processor sends elements to appropriate processors, which is an all-to-all personalized exchange, the cost is $\Theta(\alpha P + \beta(n/P^2)P)$ assuming a fully connected network

3. local bucket sort, the cost is $O(n/P)$

The combined cost is thus

$$T(n;P) = O\left(\tau\frac{n}{P} + \alpha P + \beta\frac{n}{P}\right)$$

13.9 Linear Time Distributed Sort: Part 2

To relax the uniform distribution assumption of input values, sampling technique can be used—because the ultimate goal is to (almost) evenly distribute the values over the buckets, the additional degree of freedom bucket width can be adjusted for that purpose, and sampling helps determine the appropriate width. The detailed steps are

1. each processor performs local sort
2. each processor selects $P-1$ samples which are evenly spaced in the sorted local list
3. gather all the local samples to the root processor
4. the root processor sorts the local samples
5. select $P-1$ splitters from the root sorted samples which define the global bucket boundaries
6. broadcast the splitters to all processors
7. each processor partitions its local elements using the splitters
8. all processors send out-of-boundary elements to appropriate processors
9. each processor performs local sort

13.10 Cost of Distributed Sample Sort Quiz

For the sample sort outlined above, the largest asymptotic function of P in its asymptotic running time is P^2 or $P^2 \log P$. Because the root processor needs to sort $P(P-1)$ elements, which takes $O(P^2)$ or $O(P^2 \log P)$ depending on the sort algorithm used. This could be a delimiter to scalability if the system is truly massive.

14 Distributed BFS

14.1 Graphs and Adjacency Matrices

Start by giving each vertex an integer label, next create an **adjacency matrix** A to represent the graph with $a_{ij} = 1$ if there is an edge connecting node i and node j and 0 otherwise. For an undirected graph its adjacency matrix is symmetric and the number of nonzeros is twice the number of edges.

14.2 The Adjacency Matrix of a Directed Graph Quiz

To get the adjacency matrix of a different vertex numbering, some permutation of rows and columns is necessary.
Empty rows correspond to nodes with no outgoing edges, whereas empty columns correspond to nodes with no incoming edges.

14.3 Losing Your Direction Quiz

The adjacency matrix of the undirected version of a directed graph can be derived from the adjacency matrix of the directed one through $A_u = \mathrm{OR}(A_d, \mathrm{trans}(A_d))$, where OR is the logical OR operation and trans is matrix transpose operation.

14.4 Breadth First Search Review

A level-synchronous BFS with $G = (V, E)$ starts with source $s \in V$ and level $l = 0$ and then propagates the frontier consisting of all unvisited vertices, processing the level one at a time. Processing the level means visiting all the unvisited neighbors of the frontier, which then becomes the frontier at the next level. The sequential cost is $O(m + n)$ where m is the number of edges and n is the number of nodes.

14.5 Matrix-based BFS

The level-synchronous BFS can be translated into matrix form as follows. For an unvisited vertex i and a frontier at level l, if there is any edge from the frontier to i, the distance of i should be updated. In adjacency matrix this is equivalent to checking whether in column i there is any entry in the rows corresponding to the frontier has value 1. The update can be represented by an update vector and the check can be implemented algebraically as a boolean matrix-vector multiplication $u_i = \wedge_{j=1}^{n}(f_j \vee a_{ji})$ where $\wedge$ and $\vee$ are logical OR and AND operation respectively or concisely $u = A^T \cdot f$. Note that the cost of this update or matrix-vector multiplication is not $O(n^2)$ for a sparse graph, because it is then a sparse matrix multiplied by a sparse vector and can be implemented efficiently to achieve work-optimal matrix-based BFS.
The distance can be updated as follows

$$\begin{aligned} &\text{for all } u_i = 1 \text{ and } d_i = \infty \text{ do} \\ &\quad d_i \leftarrow l + 1 \\ &\quad f_i^{\text{next}} \leftarrow 1 \end{aligned}$$

14.6 1D Distributed BFS

The steps are

1. partition columns of A and entries of u among the processors
2. compute $u = A^T f$ locally
3. update distances locally
4. identify local vertices of the next frontier
5. all-to-all exchange to form the full frontier visible to all processors

The last step is the only communication step, the cost of which depends on the structure of the graph and is linear in the number of processors i.e. $O(\alpha \cdot P + \beta \cdots)$.

14.7 2D Distributed BFS

Let P be the number of processors. 1D distributed BFS has a communication cost that is linear in P i.e. $O(\alpha \cdot P + \beta \cdots)$. In contrast 2D distributed BFS has a communication cost that is linear in $\sqrt{P}$ i.e. $O(\alpha \cdot \sqrt{P} + \beta \cdots)$. Because A can be partitioned into a $\sqrt{P} \times \sqrt{P}$ grid and we only need to merge local frontiers within the respective columns and rows.

14.8 Conclusion

The basic idea of realizing distributed operations is to cast BFS into a matrix form so that distributed matrix operations can be applied.

15 Graph Partitioning

15.1 The Graph Partitioning Problem

Given graph $G = (V, E)$ and the number of partitions P, compute a (vertex) partition $V = V_0 \cup V_1 \cup V_2 \cup \cdots \cup V_{P-1}$ such that

disjoint: $V_i \cap V_j = \emptyset$

balanced vertex partition: $|V_i| \sim |V_j|$

minimize edge cuts: minimize $|E_{\text{cut}}|$ where $E_{\text{cut}} = \{(u, v) | u \in V_i, v \in V_j, i \neq j\}$ are edge cuts

15.2 Do You Really Want a Graph Partition Quiz

The above requirements on partition are not exactly equivalent to the goals of load balancing and minimizing communication. Because it is possible that the vertex counts per partition are the same, but the number of nonzero entries in the rows of the corresponding sparse matrix i.e. work are not.

15.3 Graph Bisection and Planar Separators

Graph partitioning is an NP-complete problem, thus we need good heuristics and exploit special structure for intuition. One such good heuristic is graph bisection and one special structure is planar graph. A planar graph is one that can be drawn in the plane with no edge crossings. For planar graphs there is a separator theorem (Lipton and Tarjan) which states that, any planar graph $G = (V, E)$ with $|V| = n$ vertices has a disjoint partition $V = A \cup S \cup B$ such that

- S separates A and B, i.e. taking S out of a connected graph results in two or more connected graphs
- $|A|, |B| \leq 2n/3$, which implies that $|A|/|B| \leq 2$
- $|S| \leq 2\sqrt{2} \cdot \sqrt{n} = O(\sqrt{n})$

and there is a polynomial time algorithm to find such a separator.

15.4 Partitioning via Breadth First Search Quiz

The idea comes from the fact that levels separate subgraphs. Assume the graph is connected. Run BFS from any vertex. Stop when about 1/2 vertices are visited. Assign visited vertices to one partition and unvisited vertices to the other.

15.5 Kernighan-Lin Part 1: No Gain is Pain

Given a graph, choose any vertex partition to divide the vertices into two subsets of equal or nearly equal size $V = V_1 \cup V_2$ where $|V_1| \approx |V_2|$. Define the cost of partition $\text{cost}(V_1, V_2)$ to be the number of edges between V_1 and V_2. Consider two subsets $X_1 \subseteq V_1$ and $X_2 \subseteq V_2$ with $|X_1| = |X_2|$. The cost to swap these two subsets can be computed as follows: define

- the external cost of a vertex to be the number of edges that land in the other partition

$$E_1(a \in V_1) = \#\text{edges}(a, v \in V_2)$$
$$E_2(b \in V_2) = \#\text{edges}(b, v \in V_1)$$

- the internal cost of a vertex $I_1(a \in V_1)$ to be the number of edges that land in the same partition

$$I_1(a \in V_1) = \#\text{edges}(a, v \in V_1)$$
$$I_2(b \in V_2) = \#\text{edges}(b, v \in V_2)$$

Using external and internal cost the cost of partition can be expressed as

$$\text{cost}(V_1, V_2) = \text{cost}(V_1 - \{a\}, V_2 - \{b\}) + E_1(a) + E_2(b) - c_{a,b}$$

where the constant $c_{ab} = 1$ if there is an edge connecting a and b and 0 otherwise ($\Rightarrow$ subtraction due to double counting by E_1 and E_2).
Now consider swapping a and b to form two new partitions $\widehat{V}_1 = (V_1 - \{a\}) \cup \{b\}$ and $\widehat{V}_2 = (V_2 - \{b\}) \cup \{a\}$, the cost of which is ($\Rightarrow$ addition due to double omission by I_1 and I_2)

$$\text{cost}(\widehat{V}_1, \widehat{V}_2) = \text{cost}(V_1 - \{a\}, V_2 - \{b\}) + I_1(a) + I_2(b) + c_{a,b}$$

The change in cost named gain is

$$\text{gain}(a \in V_1, b \in V_2) = \text{cost}(V_1, V_2) - \text{cost}(\widehat{V}_1, \widehat{V}_2) = E_1(a) + E_2(b) - I_1(a) - I_2(b) - 2c_{a,b}$$

15.6 Kernighan-Lin Algorithm Quiz

Assume that (1) every vertex has a partition label that can be accessed by constant time (2) the maximum degree of any vertex is d. The sequential computation of $\text{gain}(a, b)$ takes $O(d)$ time, because (i) to compute the gain we need to sweep over the neighbors of a and b and they have at most d neighbors as the maximum degree is d (ii) accessing the neighbors' partition labels takes constant time.

15.7 Kernighan-Lin Algorithm

Given a graph $G = (V, E)$ and a partition of its vertices V_1 and V_2. Improve the partition by swapping two subsets X_1 and X_2 as follows

1. start by computing the external and internal costs of all vertices

$$\text{for all } a \in V_1 \text{ do compute } E_1(a),\ I_1(a)$$
$$\text{for all } b \in V_2 \text{ do compute } E_2(b),\ I_2(b)$$
$$\text{for all } v \in V \text{ do visited}[v] \leftarrow \text{false}$$

2. iteratively select the pair of unvisited vertices with the largest gain, update all external cost and internal cost as if they have been swapped ($\Rightarrow N/2$ iterations and $O(N^2)$ time per iteration, thus is expensive and only applicable to small graphs, which motivates coarsening, ref. office hour)

   while $\exists$unvisited vertices do {
   choose unvisited $a \in V_1$ and $b \in V_2$ with largest gain(a, b)
   visited$[a]$, visited$[b] \leftarrow$ true
   update all $E_1(\cdot)$, $E_2(\cdot)$, $I_1(\cdot)$, $I_2(\cdot)$ as if a and b are swapped }

3. define cumulative gain $\text{Gain}(j) = \sum_{k=1}^{j} \text{gain}(a_k, b_k)$, choose $j_{\max} = \text{argmax}_j \text{Gain}(j)$, if the maximum cumulative gain is positive then swap these two subsets

   if $\text{Gain}(j_{\max}) > 0$ then {
   $X_1 = \{a_1, a_2, \ldots, a_{j_{\max}}\}$
   $X_2 = \{b_1, b_2, \ldots, b_{j_{\max}}\}$
   $V_1 \leftarrow (V_1 - X_1) \cup X_2$
   $V_2 \leftarrow (V_2 - X_2) \cup X_1$
   $\text{cost}(V_1, V_2) \leftarrow \text{cost}(V_1, V_2) - \text{Gain}(j_{\max})$ }

   the algorithm terminates when $\text{Gain}(j) \leq 0$ for all j's

The main problem with this algorithm is that its sequential running time is $O(|V|^2 d)$ where d is the maximum degree of vertices, albeit there are some complex variations of this algorithm that can reduce the per iteration running time to $O(|E|)$.

15.8 Graph Coarsening

A different partitioning strategy is multi-level graph coarsening. It is a form of divide and conquer. The idea is to repeat the graph coarsening a few times until we obtain a version of the graph that is small enough to partition quickly. The partition of the coarsest graph would then suggest a partition of its parent graph if the coarsening preserves the relationship between the coarsened graph and its parent graph. Repeat the step until we have a partition of the original graph.
Most coarsening scheme is carried out by identifying subsets of the vertices to collapse. To track the collapse we weight both vertices and edges as follows: let $G_i = (V_i, w_i : V_i \to \mathbb{N}, E_i, x_i : E_i \to \mathbb{N})$ where w is the vertex weight function, x is the edge weight function, and i is the level of the graph. Collapsing vertex u and v to s amounts to the following update of vertex weights and edge weights

$$w_{i+1}(s) = w_i(u) + w_i(v)$$
$$x_{i+1}(s, t) = x_i(u, t) + x_i(v, t)$$

15.9 Maximal and Maximum Matchings

One scheme to decide which vertices to collapse is matching, a **matching** of a graph $G = (V, E)$ is a subset of edges $\widehat{E} \subseteq E$ with no common end points. It is basically an independent set of edges. Note the difference between maximal matching and maximum matching

maximal matching: a matching to which you can't add any more edges without sharing common end points

maximum matching: the largest matching, largest in the sense of having most edges or total edge weights

Maximal matchings are easier to compute so we'll focus on these.

15.10 A Fact about Maximal Matchings

Consider an initial graph G_0, repeatedly finding a maximal matching and using it to collapse vertices produces a sequence of graphs $G_1, G_2, \ldots, G_k$. Suppose $|V_0| = n$ and $|V_k| = s$. k has the lower bound $k = \Omega(\log(n/s))$. Because in the best scenario a maximal matching will touch every vertex (what is the worse scenario?), thus upon collapse/coarsening the number of vertices is halved. We therefore have the following sequence of inequalities

$$s = |V_k| \geq \frac{|V_{k-1}|}{2} \geq \frac{|V_{k-2}|}{2^2} \geq \cdots \geq \frac{|V_0|}{2^k} = \frac{n}{2^k} \implies k \geq \log_2(n/s)$$

15.11 Computing a Maximal Matching

A simple scheme of computing maximal matching based on randomization consists of (1) picking an unmatched vertex at random (2) matching this vertex to one of its unmatched neighbors. There are many ways to choose unmatched neighbors, one of them called *heavy edge matching* chooses the unmatched edge with the highest weight. The informal intuition behind this is the relationship in the sum of edge weights between the initial graph and the graph after collapse

$$X(G_{i+1}) = X(G_i) - X(M_i)$$

where M_i is the matching and $X(M_i)$ is the edge weight sum of the matched edges. The strategy of selecting heavy edges is maximizing $X(M_i)$ so as to minimize $X(G_{i+1})$, because the smaller $X(G_{i+1})$ the more likely the edge cuts in G_{i+1} will be small.

15.12 Fine-to-Coarse and Back Again Quiz

The set of edges and vertices in the original graph that correspond to the separator in the coarsened graph is called the *projected separator*, which may include unnecessary edges. This implies that the coarsening technique will probably need partition refinement.

15.13 Partition Refinement Quiz

A minimum balanced edge cut in a coarsened graph does not necessarily minimize the balanced edge cut in the next finer graph, because coarsening is only a heuristic (⇒ and a maximal matching is not necessarily the maximum matching?).

15.14 Spectral Partitioning, Part 1: The Graph Laplacian

Represent a graph G by its incidence matrix $C = C(G)$ in which each row corresponds to an edge of G and each column is a vertex. The directions of the edges are marked by putting $+1$ at the source vertex and -1 at the sink for each edge/row. The graph Laplacian $L(G)$ is defined as $L(G) = C^T C$. Symbolically

$$C = \begin{pmatrix} e_0^T \\ e_1^T \\ \vdots \\ e_{m-1}^T \end{pmatrix} \implies C^T C = \sum_{k=0}^{m-1} e_k \cdot e_k^T = D - A$$

where D is the diagonal matrix containing the degree of each vertex and A is the adjacency matrix of the undirected version of G. Because suppose e_k represents edge$(i \to j)$, then

$$e_k \cdot e_k^T = \begin{pmatrix} \vdots \\ +1 \\ \vdots \\ -1 \\ \vdots \end{pmatrix} \cdot \begin{pmatrix} \cdots & +1 & \cdots & -1 & \cdots \end{pmatrix} = \begin{pmatrix} & \vdots & & \vdots & \\ \cdots & +1 & \cdots & -1 & \cdots \\ & \vdots & & \vdots & \\ \cdots & -1 & \cdots & +1 & \cdots \\ & \vdots & & \vdots & \end{pmatrix}$$

where the diagonals are the number of incident edges at i and j, whereas the off-diagonals indicate the presence of an edge connecting i and j albeit the directional information is lost.

15.15 Graph Laplacian Quiz

As a sanity check, every row and every column of graph Laplacian should sum to 0.

15.16 Spectral Partitioning, Part 2: Springs Fling

Applying Newton's second law and Hooke's law to a 1D spring system we have (where x_i denotes the vertical displacement of spring i)

$$m\frac{d^2}{dt^2}\begin{pmatrix} x_0(t) \\ x_1(t) \\ \vdots \\ x_{n-1}(t) \end{pmatrix} = -k \begin{pmatrix} 1 & -1 & & & \\ -1 & 2 & -1 & & \\ & -1 & 2 & -1 & \\ & & \ddots & & \\ & & -1 & 2 & -1 \\ & & & -1 & 1 \end{pmatrix} \cdot \begin{pmatrix} x_0(t) \\ x_1(t) \\ \vdots \\ x_{n-1}(t) \end{pmatrix}$$

where the coefficient matrix is the graph Laplacian of a line graph. The solution to this differential equation, the motion of the spring system, can be described as a sum of fundamental displacements (harmonics in music).

15.17 Spectral Partitioning, Part 3: Algebraic Connectivity

Since graph Laplacian is symmetric, it has real-valued non-negative eigenvalues λ_i and real-valued orthogonal eigenvectors q_i. We can sort the eigenvalues as $\lambda_{n-1} \geq \lambda_{n-2} \geq \cdots \geq \lambda_1 \geq \lambda_0 \geq 0$. Then Fiedler's theorem states that the graph has k connected components if and only if $\lambda_{k-1} = \lambda_{k-2} = \cdots = \lambda_1 = \lambda_0 = 0$. This is significant because it means that the spectrum of graph Laplacian determines the connectivity of the graph.
Consider a partition $V = V_+ \cup V_-$. Define a vertex function x to be $x_i = +1$ if $i \in V_+$ and $x_i = -1$ if $i \in V_-$. Then the number of edges cut by this partition is given by $x^T L(G)x/4$. Hence minimizing partition edge cut is equivalent to minimizing this matrix-vector product.

15.18 Counting Edge Cuts Quiz

Decompose the above matrix-vector product as

$$x^T L(G)x = \sum_{i=j} l_{ij}x_i x_j + \sum_{i,j\in V_+, i\neq j} l_{ij}x_i x_j + \sum_{i,j\in V_-, i\neq j} l_{ij}x_i x_j + \sum_{i\in V_+, j\in V_-} l_{ij}x_i x_j + \sum_{i\in V_-, j\in V_+} l_{ij}x_i x_j$$

where

- $\sum_{i=j} l_{ij}x_ix_j$ sums the degrees of all vertices as $l_{ii} = d_i$ and $x_i^2 = 1$, which is equivalent to counting the number of edges twice.
- $\sum_{i,j\in V_+, i\neq j} l_{ij}x_ix_j$ sums the negative number of edges contained in V_+ twice as $l_{ij} = -1$ and $x_i = x_j = 1$
- $\sum_{i,j\in V_-, i\neq j} l_{ij}x_ix_j$ sums the negative number of edges contained in V_- twice as $l_{ij} = -1$ and $x_i = x_j = 1$
- $\sum_{i\in V_+, j\in V_-} l_{ij}x_ix_j$ sums the number of edges crossing the partition as $l_{ij} = -1$ and $x_i = 1$ and $x_j = -1$
- $\sum_{i\in V_+, j\in V_-} l_{ij}x_ix_j$ sums the number of edges crossing the partition as $l_{ij} = -1$ and $x_i = -1$ and $x_j = 1$

Summing them together gives four times the number of edges cut by the partition.

15.19 Spectral Partitioning, Part 4: Putting It All Together

As mentioned above to minimize the number of edge cuts we only need to minimize $x^TL(G)x/4$ subject to two conditions (1) $x_i = \pm 1$ (2) $\sum_i x_i = 0$ i.e. V_+ and V_- have the same number of vertices. This problem however is NP-complete. Nevertheless relaxing $x_i = \pm 1$ we can apply Courant-Fisher minimax theorem to get a lower bound. The theorem states that let y be any real vector, then

$$\min_{y^Ty=n, \sum_i y_i=0} \frac{1}{4}y^TL(G)y = \frac{1}{4}nq_1^TL(G)q_1 = \frac{1}{4}n\lambda_1$$

where q_1 and λ_1 are the second smallest eigenvector and eigenvalue of $L(G)$. Hence the lower bound of the number of edge cuts is

$$\min_{x_i=\pm 1, \sum_i x_i=0} \frac{1}{4}x^TL(G)x \geq \frac{1}{4}n\lambda_1$$

and choosing x to be q_1 gives the lower bound. This motivates the following spectral graph partitioning algorithm

1. compute $L(G)$
2. compute (λ_1, q_1) eigenvalue and eigenvector of $L(G)$
3. choose $x_i \leftarrow \text{sign}(q_1(i))$

which works really well for planar graphs.

15.20 Conclusion

Since graph partitioning is NP-complete, we need algorithms with good heuristics. The heuristics covered are (1) multi-level partitioning (2) exploiting special structure like planarity (3) no gain is pain improvement technique (4) spectral partitioning.

16 Algorithm Time: Energy and Power

16.1 Speed Limits Quiz

The speed of light posts a physical limit on communication speed.

16.2 Space Limits Quiz

At some point the physical limit of the size of one bit storage will be reached.

16.3 Balance in Time

The density of transistors doubles every 1.9 years, whereas the slow-fast memory communication bandwidth doubles every 2.9 years. Hence the balance doubles roughly every 5.5 years, because

$$B = \frac{R}{\beta} = \frac{2^{t/1.9}}{2^{t/2.9}} = 2^{t(1/1.9 - t/2.9)} \approx 2^{t/5.5}$$

It suggests that it might be more beneficial to trade off more computation for less communication. Note that W, D, and Q count the number of operations ignoring R and β, in other words we usually compute W, D and Q assuming unit cost operations. Translating to real cost $1/R$ and $1/\beta$ the work and span law still applies, but with an additional term accounting for the slow-fast memory communication and scaled by R and β

$$T_p \geq \max\left(\frac{D}{R}, \frac{W}{PR}, \frac{QL}{\beta}\right)$$

Assuming $W/P \gg D$ (a short critical path), to benefit from transistor density scaling and to minimize the right-hand side we want the compute time to dominate over the communication time, i.e.

$$\frac{W}{PR} \gg \frac{QL}{\beta} \iff \frac{W}{Q} \gg \frac{R}{\beta} PL$$

Thus on the algorithm side the goal is to make W/Q as large as possible knowing that R/β is subject to inevitable scaling trend that causes it to grow over time. Hence achieving balance is the best shot at scaling in the future—the balance principle.

16.4 Double, Double Toil and Trouble Quiz

For sorting $W/D \sim L \log(Z/L)$. Plug it into the balance principle formula

$$L \log \frac{Z}{L} \sim \frac{W}{Q} \gg \frac{R}{\beta} PL \implies \log \frac{Z}{L} \sim \frac{R}{\beta} P$$

Thus either (1) squaring both Z and L or (2) doubling β would maintain the balance if P is doubled. However (1) requires upgrading cache to a unprecedented generation and for (2) historically bandwidth doesn't grow as fast as transistor density. So neither one of them is practical. Hence it seems that at least for a computation like sorting there appears to be some fundamental limits that make it hard to build a balanced system.

16.5 Power Limits

Consider a power consumption model $P = P_0 + \Delta P$ where P_0 stands for constant or static power that is independent of the running computation, whereas ΔP is the dynamic power that depends on the computation.

16.6 The Dynamic Power Equation

The dynamic power equation is given by

$$\Delta P = CV^2 \times f \times a$$

where

- CV^2 is the energy per gate, C being the gate's capacity and V the supply voltage
- f is the clock rate, number of cycles per unit time, which determines the maximum number of times the gate can switch states per unit time
- a is the activity factor, number of switches per cycle, with a maximum value of 1

Note that the clock rate f is proportional to the supply voltage V. Hence ΔP is proportional to V^3.

16.7 Power Motivates Parallelism Quiz

Given the dynamic power equation, a k-fold reduction in clock rate would lead to a k^3-fold reduction in dynamic power due to the clock rate's proportionality with supply voltage albeit the time to run the same program is increased by k fold. This is the usual argument people make for going parallel through multi-core design rather than increasing clock rate, because you can run the program over k such low-clock-frequency cores to achieve the same execution time along with the k^3-fold power reduction.

16.8 Power Knobs Quiz

Which of the factors in the dynamic power equation can be controlled in software? The following answer is subject to debate

- V and f, through the dynamic voltage and frequency scaling (DVFS) of modern processors and memory systems, e.g. *cpufreq* in Linux
- a, if you know at the algorithm level the big chunks of the processing system that you don't need, you might turn them off, e.g. turning the cache off while performing reduction

16.9 Algorithmic Energy Quiz

Among the five metrics of the work-span model: (1) work W (2) span D (3) average available parallelism W/D (4) time $\max(D, W/P) \leq T_P \leq D + (W - D)/P$ and (5) speedup $S_P = T_1/T_P$, the metric that best quantifies energy is work D. Because while we can reduce time by hiding it through overlap (e.g. parallelism), we must pay energy cost for every operation. This implies that at the algorithmic level, if we only care about energy then work-optimality is of paramount importance.

16.10 Algorithmic Dynamic Power Quiz

Among the five metrics of the work-span model: (1) work W (2) span D (3) average available parallelism W/D (4) time $\max(D, W/P) \leq T_P \leq D + (W - D)/P$ and (5) speedup $S_P = T_1/T_P$, the metric that best expresses dynamic power is sppedup $S_P = T_1/T_P$. Because power is energy divided by time and energy is proportional to work W, which is basically the time to execute the algorithm on a single processor T_1. This implies that there is a fundamental tension between speed and power—high speed is less time but more power.

16.11 Parallelism and DVFS Quiz

Suppose we slow down the clock by a factor of σ, that is $f \to f/\sigma$, and we consume the same power by using $\sigma^3 P$ instead of P cores. To find the best value of σ from the execution time perspective is to apply Brent's theorem

$$T_P \leq D + \frac{W - D}{P} \implies T_P \leq \sigma \left(D + \frac{W - D}{\sigma^3 P} \right) = g(\sigma)$$

To minimize the upper bound, take the first derivative of $g(\sigma)$ and equate it to zero we obtain

$$\frac{dg}{d\sigma} = D - 2\frac{W - D}{\sigma^3 P} = 0 \implies \sigma = \left(2\frac{W - D}{PD} \right)^{1/3}$$

Nonetheless we may not always get a speed up with this optimal σ, which depends on the value of P.