1 Prologue

When we analyze the running time of algorithms, we tend to treat each computational step as having unit cost. But when we run these algorithms on actual computers, their running time is strongly influenced by features of the architecture, such as parallelism and multi-level memory hierarchies, that are ignored in the model that treats every step of an algorithm as having unit cost. The theme of these lecture notes is that quite often, divide and conquer algorithms are able to automatically take advantage of these features without the programmer having to explicitly think about them.

To illustrate this point, we start with a simple example. Consider the following two programs for computing the maximum of a set of non-negative integers.

Algorithm 1 LoopMax$(a_1, \ldots, a_n)$

```
1: m = 0
2: for i = 1, ..., n do
3:     m ← max(m, a_i)
4: end for
5: return m
```

Algorithm 2 RMax$(a_1, \ldots, a_n)$

```
1: if n = 1 then
2:     return a_1
3: else
4:     k ← ⌊n/2⌋
5:     m_0 ← RMax(a_1, ..., a_k)
6:     m_1 ← RMax(a_{k+1}, ..., a_n)
7:     return max(m_0, m_1)
8: end if
```

Figure 1: Two algorithms for computing the maximum of n numbers.

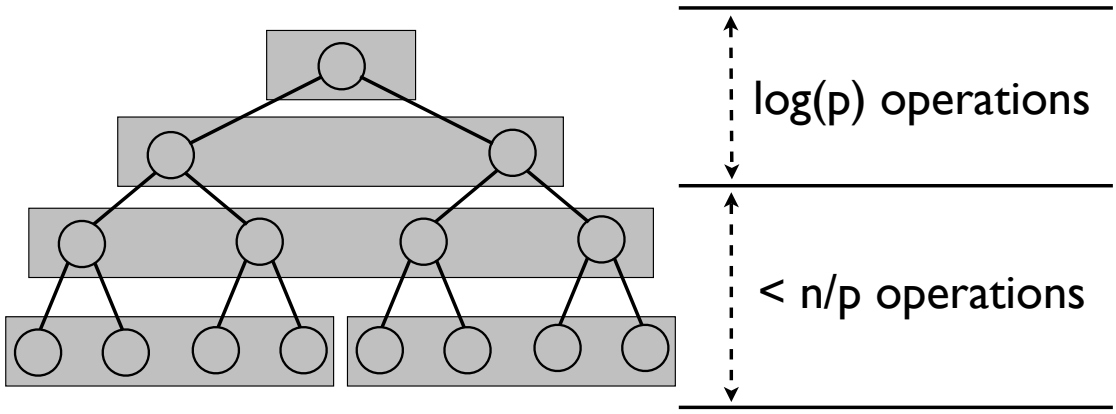

Figure 2: Parallel implementation of algorithm RMax

Both of these algorithms run in time $O(n)$. However, on a machine with p parallel processors, the running time of LoopMax is still $O(n)$ but the running time of RMax is $O(n/p + \log(p))$

assuming that the compiler and operating system take maximum advantage of the resources available; see Figure 2 for a schematic diagram illustrating the "comparison tree" produced by the RMax algorithm and the partition of that tree into $O(n/p + \log(p))$ groups of comparisons that can be done in parallel.

2 Cache-Oblivious Algorithms

Now we'll see how divide and conquer algorithms naturally take advantage of another architectural feature: multi-level memory hierarchies.

Data is stored in a computer's memory in the form of *lines* consisting of L consecutive *words*. (A word is an indivisible chunk of data in memory, i.e. the contents of a single memory address. On typical present-day systems, a word consists of 4 or 8 bytes. When we discuss algorithms for processing data stored in the form of vectors, matrices, linked lists, we implicitly assume that each component of the vector, entry of the matrix, or element of the linked list occupies $O(1)$ words in memory.) We assume that the memory addresses are numbered starting from 0, and that every line of memory begins with an address that is an integer multiple of L and finishes with the address preceding the next integer multiple of L.

The processor has a cache of size Z that can hold up to Z/L lines of data. When the processor reads any item stored in memory, it pulls the entire line containing that item into the cache. Similarly, when it writes any item to memory, it writes the entire cache line containing that item. Performing operations on the data stored in the cache is much faster than reading or writing memory.

The preceding paragraphs describe a two-level memory hierarchy. In reality, the hierarchy extends over multiple levels. In fact, one can consider the disk to be a further extension of the memory hierarchy, and to some extent, the network (local-area and/or wide-area) extends the hierarchy one or two levels further still. In these notes we will limit ourselves to analyzing a two-level memory hierarchy. However, the type of algorithms we will be analyzing have the very attractive property that their analysis treats different levels of the hierarchy independently, so that if we can prove that they perform optimally in a two-level memory hierarchy then it automatically follows that they perform optimally in a multi-level hierarchy!

2.1 Cache complexity

As stated earlier, we assume that our algorithms run on a system with a two-level memory hierarchy whose cache has size Z and is divided into Z/L lines of size L. We will assume that the cache is *tall*, meaning that $Z/L \geq c \cdot L$ for some positive constant c. This is a common assumption in the design of cache-oblivious algorithms, and the caches on actual computers typically satisfy the tall cache assumption, with $c \geq 1$. For the rest of these lecture notes, we'll make the simplifying assumption that $c \geq 1$, in other words, that $Z/L \geq L$.

We will be designing algorithms that are *cache-oblivious* in the sense that the parameters Z and L are not encoded in the algorithm. This has the obvious advantage that the algorithm is portable across multiple cache configurations. The question is, do we suffer a significant performance penalty for our decision not to hard-code parameters such as the size of the cache into the algorithm? We will see that often, the answer is that there is little or no penalty associated with cache-obliviousness, provided that the algorithm is properly designed.

We will analyze algorithms according to two measures of performance: their running time (which is the standard notion of running time for a sequential algorithm, i.e. charging one unit of

cost for each operation the algorithm performs) and their cache complexity, which is the number of read and write operations that the algorithm performs when moving lines of memory into and out of the cache. Such operations occur whenever the algorithm attempts to read or write a memory location that is not already stored in the cache. When this happens, it must read a new line of memory (i.e., a block of L consecutive words starting from an address that is an integer multiple of L) into the cache. If the cache was already full, it must evict one of the lines that was stored in the cache before the new read/write operation took place. When the algorithm writes data to a memory location belonging to a line that is already in the cache, the data stored in that cache location is updated and there is no cost in terms of cache complexity. The actual memory is not updated until that cache line is evicted or the algorithm terminates, whichever happens first.

We will assume that the cache is *fully associative*, meaning that any line of memory can be stored in any line of the cache. We will assume that the algorithm uses an *optimal replacement* policy, meaning that when it needs to evict a line of memory from the cache, it evicts the one which will be accessed farthest in the future. Both of these assumptions are unrealistic — the latter one obviously so — but they will make the presentation of the ideas much simpler, and it turns out that both assumptions can be removed (under mild hypotheses on the algorithm's behavior) without increasing cache complexity by more than a constant factor. See "Cache-Oblivious Algorithms" by Frigo, Leiserson, Prokop, and Ramachandran (Proceedings of FOCS 1999) for details.

2.2 First example: Scanning consecutive locations

As it happens, many simple algorithms that scan data by making a single pass through it (or any fixed number of passes) are already cache-oblivious without having to resort to any special divide-and-conquer tricks. For example, let us analyze the cache complexity of the algorithm LoopMax given above in Section 1.

Theorem 1. *Assuming that the numbers $a_1, \ldots, a_n$ are stored in consecutive locations in memory, the cache complexity of* LoopMax *is at most $1 + \lceil n/L \rceil$.*

Proof. Since LoopMax never writes to any location in memory, we only have to bound the amount of work done reading the data $a_1, \ldots, a_n$ from memory. The algorithms reads these words in consecutive order, so it pulls a line into the cache when reading a_1 and every subsequent time that it reads a number whose address is an integer multiple of L. There are at most $\lceil n/L \rceil$ of the latter type of cache misses, so the total number of cache misses is bounded by $1 + \lceil n/L \rceil$. □

2.3 Matrix transposition

Let us consider a more sophisticated example: matrix transposition. We will assume that matrices are stored in *row-major order*. This means that a matrix A whose entries are denoted by a_{ij} $(i = 1, \ldots, m; j = 1, \ldots, n)$ is stored in the order $a_{11}, a_{12}, \ldots, a_{1n}, \cdots, a_{m1}, a_{m2}, \ldots, a_{mn}$. The matrix transposition problem requires us to write out the entries of the matrix $B = A^{\mathsf{T}}$ in row-major order.

The difficulty of matrix transposition arises from the fact that writing $B = A^{\mathsf{T}}$ in row-major order requires reading A in *column-major order*, and when we access the entries of A in this order they are very far from being consecutive in the memory. (See Figure 3.) To formalize this

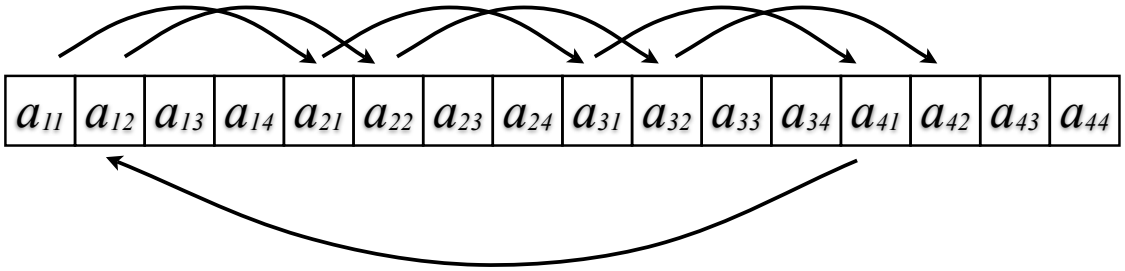

Figure 3: The matrix A is written in row-major order. When reading its entries in column-major order, the ordering is very far from consecutive, resulting in many cache misses.

insight, consider the cache complexity of the following simple matrix transposition algorithm, LoopTrans.

Algorithm 3 LoopTrans(A, B)

```
1: /* Write the transpose of A in the memory locations addressed by the entries of B */
2: for i = 1, ..., n do
3:    for j = 1, ..., m do
4:       b_ij ← a_ji
5:    end for
6: end for
```

Assuming $m, n > Z/L \geq L$, the LoopTrans algorithm suffers a cache miss on every read operation. To see this, notice that each time LoopTrans reads an entry a_{ji} of the matrix A, that entry belongs to a different row of A from the other $m - 1$ most recently accessed entries of A. In fact, if we consider the m most recently accessed entries of A, no contiguous block of $n - 1$ memory addresses contains more than one of them. By our assumption that $L < n$, this means that all of these m elements are stored in different lines of memory. By our assumption that $Z/L < m$, the cache is too small to hold all these different lines of memory simultaneously, so the act of reading a_{ji} must result in a cache miss.

Reversing the order of nesting of the loops makes no difference in the cache complexity: now we read the entries of A in row-major order — which uses the cache efficiently — but we write the entries of B in column-major order, which forces us to suffer a cache miss on every write operation for the same reason as before.

To get around this problem, we use a divide-and-conquer algorithm that recursively splits matrix A, along its larger dimension, into submatrices of nearly equal size. It transposes each of these submatrices and then stacks the results alongside each other in the appropriate configuration.

Algorithm 4 RTRANS(A, B)

1: /* Write the transpose of A in the memory locations addressed by the entries of B */
2: **if** $m = n = 1$ **then**
3: $\quad b_{11} \leftarrow a_{11}$
4: **else if** $m > n$ **then**
5: $\quad k \leftarrow \lceil m/2 \rceil$
6: $\quad$ Let $A = \begin{pmatrix} A_1 \\ A_2 \end{pmatrix}$, where A_1 has k rows and A_2 has $n - k$ rows.
7: $\quad$ Let $B = \begin{pmatrix} B_1 & B_2 \end{pmatrix}$, where B_1 has k columns and B_2 has $n - k$ columns.
8: $\quad$ RTRANS(A_1, B_1)
9: $\quad$ RTRANS(A_2, B_2)
10: **else**
11: $\quad k \leftarrow \lceil n/2 \rceil$
12: $\quad$ Let $A = \begin{pmatrix} A_1 & A_2 \end{pmatrix}$, where A_1 has k columns and A_2 has $n - k$ columns.
13: $\quad$ Let $B = \begin{pmatrix} B_1 \\ B_2 \end{pmatrix}$, where B_1 has k rows and B_2 has $n - k$ rows.
14: $\quad$ RTRANS(A_1, B_1)
15: $\quad$ RTRANS(A_2, B_2)
16: **end if**

There's a subtlety concerning the way the divide-and-conquer algorithm constructs submatrices and passes them as arguments in recursive calls to the same procedure. It's hard to express this subtlety in pseudocode, so let us discuss it here instead. The issue is best understood by considering the submatrices A_1, A_2 defined in line 12 of the pseudocode. Each of these submatrices is stored in a non-contiguous set of locations in memory, because the rows of A itself are contiguous, but each row of A contains one row of A_1 followed by one row of A_2. An obvious way to deal with this is to interpret a statement such as, "Let $A = \begin{pmatrix} A_1 & A_2 \end{pmatrix}$" to mean that the algorithm, upon reaching that line of the pseudocode, actually reformats the entries of A so that each of A_1, A_2 is stored in a contiguous block of memory in row-major order. But this be too costly: it requires $O(mn)$ work to move all the entries of A around in memory, and summing up over the $\log(mn)$ levels of recursion in the algorithm we would find that its running time is $O(mn \log(mn))$, whereas we want to do matrix transposition in $O(mn)$ time. (Permuting the entries of A to make A_1, A_2 contiguous would cost a similar $\log(mn)$ factor overhead in the cache complexity.)

Instead, we define an $m \times n$ matrix A to be stored in *generalized row-major order with offset* p if:

- Each row of A is stored in a sequence of n consecutive memory locations.
- For any two consecutive rows of A, their starting addresses differ by exactly p.

The location of a matrix in generalized row-major order is completely determined by four parameters: the numbers m, n, p together with the starting address of the first row. We refer to these four parameters as the *address specification* of the matrix A.

Note that if we implement RTRANS in a way that never permutes the contents of A, each of the submatrices A_1, A_2 defined by the algorithm is stored in generalized row-major order. The same applies to the submatrices B_1, B_2 of B, and the same property is preserved upon recursive calls to the same procedure. In other words, we should interpret RTRANS(A, B) as a procedure

whose inputs are the address specifications of two matrices A, B in generalized row-major order — not the contents of the matrices themselves. (Those contents are stored in memory, not passed as arguments to the procedure.) An instruction such as line 12 is not an instruction to write the contents of A_1, A_2 anywhere in memory, but only to compute the address specifications of A_1 and A_2, which takes constant time and does not require reading or writing any lines of memory. The only read/write operations in the entire algorithm occur when it reaches the base case of the recursion, line 3.

The running time $T(m, n)$ of RTRANS satisfies the recurrence

$$T(m,n) \leq \begin{cases} 1 & \text{if } m = n = 1 \\ 2T\left(\lceil m/2 \rceil, n\right) & \text{if } m > n \\ 2T\left(m, \lceil n/2 \rceil\right) & \text{otherwise.} \end{cases}$$

We can make a two-dimensional table containing upper bounds for $T(m, n)$ based on this recurrence, by filling in entries from left to right and top to bottom.

	$n=1$	2	3	4	5	6	7	8
$m=1$	1	2	4	4	8	8	8	8
2	2	4	8	8	16	16	16	16
3	4	8	16	16	32	32	32	32
$\vdots$	$\vdots$							$\vdots$

From this table, we can guess the bound $T(m, n) < 4mn$ and then verify it by induction.

What about the cache complexity, which we will denote by $Q(m, n)$? The recurrence for cache complexity is almost the same, except that the base case is different because when m, n are small enough that the entire matrices A, B fit in cache simultaneously, then the cache complexity of the entire matrix transposition algorithm is equivalent to the cache complexity of merely reading the entries of A and B. Specifically, suppose that m and n are both less than or equal to $L/4$. Then each row of A overlaps at most two lines of memory (if we are unlucky and a line boundary occurs in the middle of the row) and similarly, each row of B overlaps at most two lines of memory. Therefore, storing all the rows or A and all the rows of B in cache requires at most $2(m + n)$ cache lines. By our assumption that $m, n \leq L/4$, along with the tall cache assumption that $Z/L \geq L$, this implies

$$2(m + n) \leq L \leq Z/L$$

which shows that the entire matrices A, B fit into the cache simultaneously and the cache complexity of RTRANS in this case is $m + n$. Thus we have the recurrence

$$Q(m,n) \leq \begin{cases} m+n & \text{if } m, n \leq L/4 \\ 2T\left(\lceil m/2 \rceil, n\right) & \text{if } m > \max n, L/4 \\ 2T\left(m, \lceil n/2 \rceil\right) & \text{otherwise.} \end{cases} \tag{1}$$

As before, we can guess the solution of this recurrence by making a table containing upper bounds for $Q(m, n)$. This time, it will be easier to think of the rows and columns of the table as being

grouped into ranges of consecutive values, as shown here.

	$n = 1, \ldots, L/4$	$L/4+1, \ldots, L/2$	$L/2+1, \ldots, L$	$L+1, \ldots, 2L$
$m = 1, \ldots, L/4$	$L/2$	L	$2L$	$4L$
$L/4+1, \ldots, L/2$	L	$2L$	$4L$	$8L$
$L/2+1, \ldots, L$	$2L$	$4L$	$8L$	$16L$
$L+1, \ldots, 2L$	$4L$	$8L$	$16L$	$32L$
$\vdots$	$\vdots$			$\vdots$

Based on these values, we can guess the bound

$$Q(m, n) \leq 32mn/L$$

and it is easy to check, by induction, that this holds for all m, n. The important thing to notice about this bound on cache complexity is that merely copying the entries of A into the memory locations indexed by B — without performing any transposition — would have cache complexity $2mn/L$. So up to a constant factor, the cache complexity of RTrans is optimal.