Homework 29

Brian Knotten, Brett Schreiber, Brian Falkenstein

November 5, 2018

9

The general outline of the algorithm is as follows:

Due to the EREW restriction, the algorithm must first make copies of the input string C. The algorithm uses n^2 processors to make n^2 copies of the string in $log(n)$ time. It does this by using $x * n$ processors to make x copies of the input string, where each processor reads a character from the string and writes it to an array. The array is of size $n \times n^2$, each row being one copy of C. The array will also be referred to as C, as it can be thought of as adding another dimension to the input string C, where each added row is a copy of C. Note that when $x > n$, we cannot make xn copies in one step, as $x * n > n^2$. However, the additional $((x * n) - n^2)$ copies can be done in constant time. This makes the whole initial copying take $log(n)$.

Next, k processors are used to write an answer array, we'll call it M, where the $i'th$ index of M is i if there exists a prefix and suffix of length i, or 0 otherwise.

Then, n^2 processors are used to check all possible prefix/suffixes of lengths 1 to k. Each processor is identified by 2 numbers i, j. If its found that there is no prefix suffix of length $k = i$, write a zero to $M[i]$.

Next, the max of the M array must be found. Because $M[i] = i$ iff a prefix and suffix exist of length i, and 0 otherwise, if we find the max of M, we will find the max length of prefix/suffix, and thus k.

Below is the algorithm for a single processor i, j:

This algorithm takes $\log n$ time. The first phase of making n^2 copies of the input string takes $\log n$ time, since n^2 processors can double the number of copies at each step, and so can reach n^2 copies in $\log n$ steps. The second phase of using n^2 processors to check the equality of the prefixes and suffixes takes constant time. The third phase checks to see if any cell in M is zero, in which case the whole row (represented by $M[i][1]$ becomes zero. This takes $\log n$ time since z is iteratively halved. The fourth phase takes the maximum over $M[i][1]$ so that $M[1][1]$ contains the maximum k and outputs. This takes $\log n$ time since y is iteratively halved.

10

The general outline for this algorithm is as follows:

At the first time step, use $n - 1$ processors to write the numbers $1, 2, 3, 4, ...n - 1$ in memory location M. These represent values for k such that there exists a matching prefix and suffix of size k.

At the second time step, use n^2 processors to compare the prefixes and suffixes of all lengths for k from 1 to $n - 1$. If the prefix and suffix aren't equal to each other, then "zero out" the location in memory. For example, if the first 3 characters do not match the last 3 characters, then the numbers in M become $1, 2, 0, 4, ...n - 1$.

Finally, at the third time step, use n^2 processors to find the maximum number left in M. This will return the maximum valid k, and it can be done in constant time as discussed in class.

Below is an algorithm for one of the $(n - 1)^2$ processors, i, j. (Without loss of generality, the processors are labelled with two numbers, $i \in [1...n - 1]$ and $j \in [1...n - 1]$ for easier usage).

Algorithm 1 EREW $O(log(n))$ algorithm

Require: A string C of size n to be expanded to an $n^2 \times n$ array of copies, a processor $p_{i,j}$, a memory location M of size n^2

$M[i][j] \leftarrow i$ ▷ Use n^2 processors to write 1...k into $M[i]$ for all i.
$number\_of\_copies \leftarrow 1$ ▷ This is a variable to store the current number of copies made.
while $number\_of\_copies < n^2$ **do**
  if $j < number\_of\_copies$ **then** ▷ Only use the processors needed to make c copies.
    $C[i][j + number\_of\_copies] \leftarrow C[i][j]$ ▷ Copy current character to new copy location
  end if
  if $number\_of\_copies > n$ **then**
    $number\_of\_copies \leftarrow number\_of\_copies + n$ ▷ All n^2 processors can copy at most n strings eath of length n.
  else
    $number\_of\_copies \leftarrow number\_of\_copies * 2$ ▷ Otherwise, with less than n^2 processors, the number of copies can be doubled.
  end if
end while
if $C[j][2i + j] \neq C[n - i + j][2i + j]$ **then** ▷ Each processor compares two individual characters to see if the prefix is of size k. The extra dimension on C is to ensure that all processors are reading from different places in memory.
  $M[i][j] \leftarrow 0$ ▷ If any of the pairs of characters don't match, then that k isn't viable.
end if
$z \leftarrow \lfloor n/2 \rfloor$ ▷ Flatten the M array so that any zero entry makes $M[i][1] \leftarrow 0$.
while $j < z$ **do**
  $M[i][j] \leftarrow MIN(M[i][j], M[i][j + z])$ ▷ If any entry in the j column is zero, then $M[i][1]$ is 0.
  $z \leftarrow \lfloor z/2 \rfloor$
end while
$y \leftarrow \lfloor n/2 \rfloor$ ▷ Now get the max of the $M[1]$ array
while $i < y$ **do**
  $M[i][1] \leftarrow MAX(M[i][1], M[i + y][1])$ ▷ A processor can take the max of 2 values in constant time. Overwrite the greater number into $M[i]$. After $\log n$ iterations, $M[1][1]$ will contain the maximum k.
  $y \leftarrow \lfloor y/2 \rfloor$
end while
if $i == 1$ and $j == 1$ **then** ▷ Designate the first processor to exclusively read and output the solution.
  Output $M[1][1]$
end if

Algorithm 2 CRCW Common $O(1)$ algorithm

Require: A string C of size n, a processor $p_{i,j}$, a memory location M of size $n - 1$ and a memory location And of size $n - 1$.

$M[i] \leftarrow i$ ▷ Use n processors to copy the numbers $1, 2, 3...n - 1$ into M.
if $C[j] \neq C[n - i + j]$ **then**
  $M[i] \leftarrow 0$ ▷ If any of the pairs of characters don't match, then that k isn't viable.
end if
$And[i] \leftarrow 1$ ▷ Perform an EREW AND operation to find a row in T of all 1s.
if $M[i] < M[j]$ **then** ▷ Perform all possible pairwise comparisons of M using n^2 processors.
  $And[i] \leftarrow 0$ ▷ If $M[i]$ is less than any $M[j]$, then $M[i]$ cannot be the maximum k.
end if
if $And[i] = 1$ **then**
  Output i ▷ That row is the maximum k. A maximum always exists, so this will always output.
end if

This algorithm runs in $O(1)$ time, since each processor only performs a constant number of operations, as described above.

11

a

Let the input to the algorithm be two n-bit integers $X = x_0x_1...x_{n-1}$ and $Y = y_0y_1...y_{n-1}$. Let p denote the number of processors and $p = n = |X| = |Y|$. For ease of outlining the algorithm, without loss of generality let $n = 2^k$ for some $k \geq 1$. Additionally, note that any n-bit integer Z can be easily split into the sum of two integers Z_1 and Z_2 where Z_1 is an n-bit integer and Z_2 is a k-bit integer if we let Z_2 be the first (counting from right to left) k bits of Z and let Z_1 be the remaining $n - k$ bits multiplied by 2^k. For example, let $n = 8$, $Z = 11101011$ and $k = 3$. Then $Z_2 = 011$, $Z_1 = 11101 * 2^3 = 11101000$, and $Z = Z_1 + Z_2 = 11101000 + 011 = 11101011$. Additionally, note that the construction of Z_2 is merely the concatenation of k zeros onto the right side of the remaining $n - k$ bits.

The general outline for this algorithm is as follows:

Using the Divide and Conquer methodology, the inputs X and Y are split into two chunks, X_1, X_2 and Y_1, Y_2 constructed as described above, where X_1 and Y_1 are the last (from right to left) $n/2$ digits of X and Y (respectively) multiplied by $2^{n/2}$ and X_2 and Y_2 are the first (from right to left) $n/2$ digits of X and Y respectively. The pairs X_1 and Y_1 are then summed together recursively, as are X_2 and Y_2. This is done by assigning the xth of the n processors the sum of the xth bit of X and Y and then passing up any resulting carry. It takes $log(n)$ time to cascade up the addition tree, as every time we go up a level the number of sums to be performed concurrently halves.

b

The EREW PRAM algorithm works similarly to the CREW PRAM algorithm in part a, except at each of the $log(n)$ steps copies of the n bits must be made in $log(n)$ time, so the algorithm runs in $O(log^2 n)$ time.