Distributed and Parallel Systems	Due on Sunday, March 3, 2019

Assignment 1

CS4402B / CS9635B — *University of Western Ontario*

Submission instructions.

Format: The answers to the problem questions should be typed:

- source programs must be accompanied with input test files and,
- in the case of `CilkPlus` code, a `Makefile` (for compiling and running) is required, and
- for algorithms or complexity analyzes, LaTeX is highly recommended.

A PDF file (no other format allowed) should gather all the answers to non-programming questions. All the files (the PDF, the source programs, the input test files and Makefiles) should be archived using the UNIX command `tar`.

Submission: The assignment should submitted through the OWL website of the class.

Collaboration. You are expected to do this assignment *on your own* without assistance from anyone else in the class. However, you can use literature and if you do so, briefly list your references in the assignment. Be careful! You might find on the web solutions to our problems which are not appropriate. For instance, because the parallelism model is different. So please, avoid those traps and work out the solutions by yourself. You should not hesitate to contact me if you have any questions regarding this assignment. I will be more than happy to help.

Marking. This assignment will be marked out of 100. A 10 % bonus will be given if your paper is clearly organized, the answers are precise and concise, the typography and the language are in good order. Messy assignments (unclear statements, lack of correctness in the reasoning, many typographical and language mistakes) may yield a 10 % malus.

PROBBLEM 1. [20 points] Consider the following multithreaded algorithm for performing pairwise addition on n-element arrays $A[1..n]$ and $B[1..n]$, storing the sums in $D[1..n]$, shown in Algorithm 5.

Algorithm 1: Pairwise addition

```
SUM-ARRAY(A, B, D, n)
    int grain_size = ?; /* To be determined                                    */
    int r = ⌈n/grain_size⌉;
    for k = 0; k < r; ++k do
        spawn ADD-SUBARRAY (A, B, D, k · grain_size,
                                                    min((k + 1) · grain_size, n));
    sync;

ADD-SUBARRAY(A, B, D, i, j)
    for k = i, k < j; ++k do
        D[k] = A[k] + B[k];
```

1.1 Suppose that we set $grain_size = 1$. What is the *work*, *span* and *parallelism* of this implementation?

Solution.

- With $grain_size = 1$, the for-loop of the procedure SUM-ARRAY performs n iterations. Moreover, at each iteration, the function call ADD-SUBARRAY performs constant work. Therefore, the work is in the order of $\Theta(n)$.
- As for the span, it is also $\Theta(n)$: indeed, spawning the function calls does not reduce the critical path.
- Therefore, the parallelism is in $\Theta(1)$.

1.2 For an arbitrary $grain_size$, what is the *work*, *span* and *parallelism* of this implementation?

Solution.

- Let us denote the grain size by g, each function call has a cost in $\Theta(g)$.
- With $grain_size = g$, the for-loop of the procedure SUM-ARRAY performs n/g iterations. Moreover, at each iteration, the function call ADD-SUBARRAY performs $\Theta(g)$. Therefore, the work remains in the order of $\Theta(n)$.
- Here again, spawning the function calls does not reduce the critical path. So each of the n/g iterations has a span of $\Theta(g)$ and in the possible worst case, these n/g function calls are executed one after another. Hence, the span is in $O(n)$.

- Therefore, the parallelism is in $\Omega(1)$, which is not an attractive result. In practice, some benefits can come from a spawning a function call at each iteration of a for-loop, but this is hard to capture theoretically. Moreover, using cilk_for is generally the better way to go.

1.3 Determine the best value for *grain_size* that maximizes parallelism. Explain the reasons.

Solution.

- To give a precise answer, we would need to know whether some of the function calls to ADD-SUBARRAY are performed concurrently. Let us consider the best and the worst cases.
- In the worst case, these function calls execute serially, one after another, whatever is g. In which case, the parallelism is in $\Theta(1)$ and the value of g has no effect.
- In the best case, all the function calls execute in parallel. In which case, the span drops to $\Theta(n/g + g)$. The function $g \longmapsto n/g + g$ reaches a minimum (for $g > 0$) at $g = \sqrt{n}$, which suggests to use this value for maximizing parallelism.

1.4 Implement in C/C++ this algorithm with the best value of *grain_size* (which can be determined from either theory or practice), and then use Cilkview to collect the following information of the whole program with $n = 4096$ or larger:

Work (instructions) Span (instructions) Burdened span (instructions)
Parallelism Burdened parallelism

as well as the speedup estimated on 2, 4, 8, 16, 32, 64 and 128 processors, respectively.

This question receives 10 points distributed as follows:

- the code compiles: 3 points,
- the Code runs: 4 points,
- the code runs correctly against verification: 3 points.

PROBBLEM 2. [20 points] The objective of this problem is to prove that, with respect to the Theorem of Graham & Brent, a greedy scheduler achieves the stronger bound:

$$T_P \leq (T_1 - T_\infty)/p + T_\infty.$$

Let $G = (V, E)$ be the DAG representing the instruction stream for a multithreaded program in the fork-join parallelism model. The sets V and E denote the vertices and edges of G respectively. Let T_1 and T_∞ be the work and span of the corresponding multithreaded program. We assume that G is connected. We also assume that G admits a single source (vertex with no predecessors) denoted by s and a single target (vertex with no successors) denoted by t. Recall that T_1 is the total number of elements of V and T_∞ is the maximum number of nodes on a path from s to t (counting s and t).

Let $S_0 = \{s\}$. For $i \geq 0$, we denote by S_{i+1} the set of the vertices w satisfying the following two properties:

(*i*) all immediate predecessors of w belong to $S_i \cup S_{i-1} \cup \cdots \cup S_o$,

(*ii*) at least one immediate predecessor of w belongs to S_i.

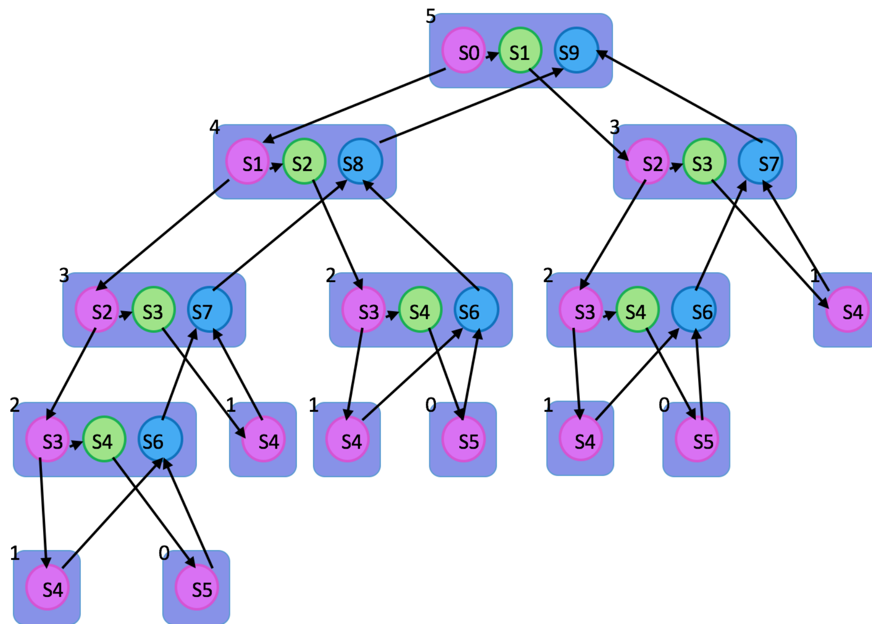
Therefore, the set S_i represents all the units of work which can be done during the i—-th parallel step (and not before that point) on infinitely many processors.

Let $p > 1$ be an integer. For all $i \geq 0$, we denote by w_i the number of elements in S_i. Let ℓ be the largest integer i such that $w_i \neq 0$. Observe that $S_0, S_1, \ldots, S_\ell$ form a partition of V. Finally, we define the following sequence of integers:

$$c_i = \begin{cases} 0 & \text{if } w_i \leq p \\ \lceil w_i/p \rceil - 1 & \text{if } w_i > p \end{cases}$$

2.1 For the computation of the 5-th Fibonacci number (as studied in class) what are $S_0, S_1, S_2, \ldots$?

Solution.



2.2 Show that $\ell + 1 = T_\infty$ and $w_0 + \cdots + w_\ell = T_1$ both hold.

Solution.

For each $i = 0 \cdots \ell - 1$, the set S_{i+1} consists of strands which cannot be executed before those in $S_i \cup S_{i-1} \cup \cdots \cup S_o$ are executed. Therefore the span T_∞ is at least $\ell + 1$. On the other hand, all strands in S_{i+1} can be executed (concurrently) after those

in $S_i \cup S_{i-1} \cup \cdots \cup S_o$ *are executed. Therefore the* T_∞ *is at most* $\ell + 1$*. These two observations imply* $\ell + 1 = T_\infty$*.*

Since $S_0, S_1, \ldots, S_\ell$ *form a partition of* V*, we clearly have* $w_0 + \cdots + w_\ell = T_1$*.*

2.3 Show that we have:

$$c_0 + \cdots + c_\ell \leq (T_1 - T_\infty)/p.$$

Solution. *We have*

$$\begin{array}{rcl} c_0 + \cdots + c_\ell & \leq & \sum_{i=0}^{i=\ell} \left(\lceil w_i/p \rceil - 1 \right) \\ & \leq & \sum_{i=0}^{i=\ell} \left(w_i/p - 1/p \right) \\ & \leq & \frac{1}{p} \sum_{i=0}^{i=\ell} \left(w_i - 1 \right) \\ & \leq & \frac{1}{p} \left(T_1 - T_\infty \right). \end{array} \tag{1}$$

Indeed, for every positive integer a, b*, one can easily verify the following inequality*

$$\lceil \frac{a}{b} \rceil - 1 \ \leq \ \frac{a-1}{b}. \tag{2}$$

2.4 Prove the desired inequality:

$$T_P \leq (T_1 - T_\infty)/p + T_\infty.$$

Solution. *We start by an interpretation of the quantity* c_i*:*

- *if* $w_i \geq p$*, that is, if one could perform at least one complete step with the strands in* S_i*, then* c_i *counts the number of other steps (incomplete or complete) that can be done after that first complete step,*
- *if* $w_i < p$*, that is, if one can only perform one step (in fact, an incomplete one) with the strands in* S_i*, then* $c_i = 0$

Therefore, in all cases, c_i *counts the number steps the number of other steps that can be done in* S_i *after that first one whether it is complete or incomplete. Hence* $c_0 + \cdots + c_\ell = T_P - (\ell + 1)$*. Recall that we have* $\ell + 1 = T_\infty$*. With the result of the previous question, we deduce the desired inequality*

$$T_P - T_\infty \ \leq \ \frac{1}{p} \left(T_1 - T_\infty \right). \tag{3}$$

2.5 Application: Professor Brown takes some measurements of his (deterministic) multi-threaded program, which is scheduled using a greedy scheduler, and finds that $T_8 = 80$ seconds and $T_{64} = 20$ seconds. Give lower bound and an upper bound for Professor Brown's computation running time on p processors, for $1 \leq p \leq 100$? Using a plot is recommended.

Solution.

According span law, work law and conclusion we get from Question 4. When p processes are used, the running time should satisfy:

$$\frac{T_1 - T_\infty}{p} + T_\infty = \frac{T_1}{p} + \frac{(p-1)T_\infty}{p} \geq T_p \geq \mathrm{Max}(T_1/p, T_\infty)$$

As $T_4 = 80$, we have $\frac{T_1}{4} \leq T_4 = 80$ and $T_\infty \leq T_4 = 80$. So we have $T_1 \leq 320$ and $T_\infty \leq 80$

As $T_{64} = 10$, we have $\frac{T_1}{64} \leq T_{64} = 10$ and $T_\infty \leq T_{64} = 10$. So we have $T_1 \leq 640$ and $T_\infty \leq 10$

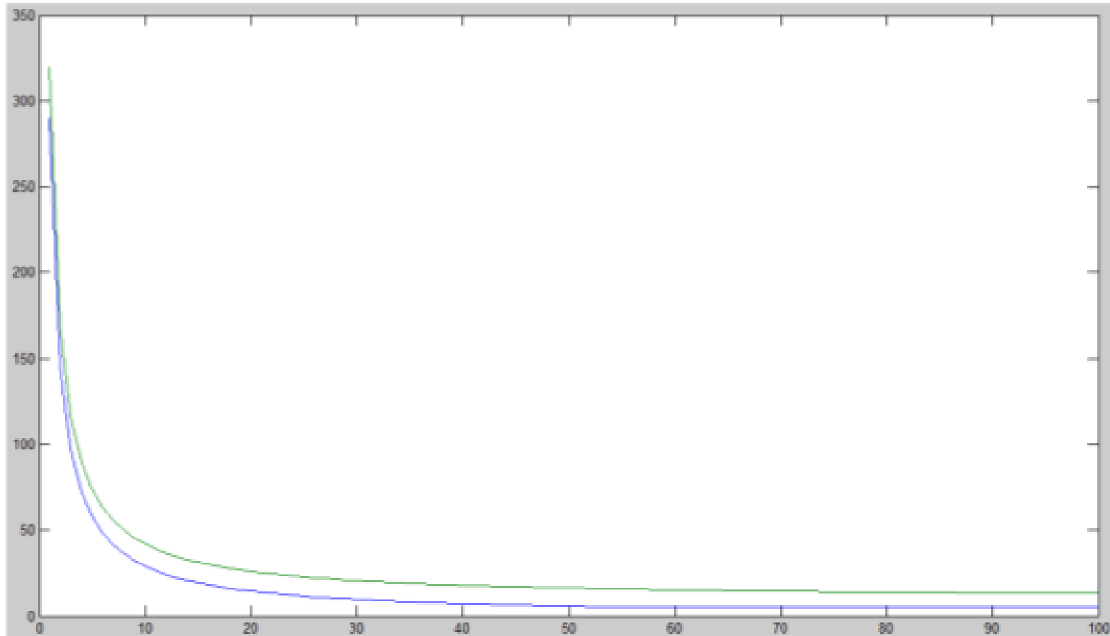
Thus $T_1 \leq 320$ and $T_\infty \leq 10$

Besides, $\frac{T_1 - T_\infty}{4} + T_\infty \geq 80$ and $\frac{T_1 - T_\infty}{64} + T_\infty \geq 10$. So we have $T_1 + 3T_\infty \geq 320$ (1) and $T_1 + 63T_\infty \geq 640$ (2). In (1), $T_\infty \leq 10$, so $T_1 \geq 290$. In (2), $T_1 \leq 320$, so $T_\infty \geq 320/63$.

$$10 + 319/p = \frac{320}{p} + \frac{10(p-1)}{p} \geq \frac{T_1}{p} + \frac{(p-1)T_\infty}{p} \geq T_p \geq \mathrm{Max}(T_1/p, T_\infty)$$

$$\geq \mathrm{Max}(290/p, 320/63)$$

So we have $\mathrm{Max}\left(290/p, \frac{320}{63}\right) \leq T_p \leq \mathrm{Min}(10 + \frac{319}{p}, 320)$ $1 \leq p \leq 100$. A plot is given below.



x-axis is number of processors (p), y-axis is time (seconds). Green line is upper bound and blue is lower bound.

The above solution is elegant and addresses the question in the best possibl way. Neverthless we accept grosser solutions where Equation (3) is used an equality in order to numerically determine T_1 and T_∞. After that, one observes

$$\frac{T_1 + (p-1)T_\infty}{p} \geq T_P \geq \max(T_1/p, T_\infty)$$

and plots the above upper and lower bounds of T_P.

PROBBLEM 3. [20 points] Given a weighted directed graph $G = (V, E)$, where each edge $(v, w) \in E$ (vertices $v, w \in V$) has a non-negative weight, the Floyd-Warshall algorithm, shown in Algorithm 2, can find the shortest paths between all pairs of vertices in G. Let $|V|$ be the number of vertices in G.

3.1 Determine which loops among the k-loop, i-loop and j-loop can be parallelized and explain the reasons.

Solution. From the proposed pseudo-code, it is unclear that any of the 3 for loops could become of a parallel loop. Thus, it is an acceptable solution to say: none! The challenge is the *dynamic programming formulation.* In fact, one needs to rework the algorithm a bit so as to obtain a *bloking strategy formulation.* See for instance:

Algorithm 2: The Floyd-Warshall algorithm

```
/* Let D be a |V| × |V| array of minimum distances initialized by the
   weighted directed graph G.                                          */
for k = 0; k < |V|; ++k do
    for i = 0; i < |V|; ++i do
        for j = 0; j < |V|; ++j do
            if D[i][j] > D[i][k] + D[k][j] then
                D[i][j] = D[i][k] + D[k][j];
```

https://gkaracha.github.io/papers/floyd-warshall.pdf

and

https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=1333649

From there, one deduces that the two inner for loops can bemcome parallel_for loops. Indeed, the the "i" and "j" iterations are independent of each other. This yields a fork-join algorithm with a work in $\Theta(n^3)$ and a span in $\Theta(n)$.

3.2 The wikipedia page `https://en.wikipedia.org/wiki/Parallel_all-pairs_shortest_path_algorithm#Parallelization` explains a parallelization of Floyd-Warshall algorithm. Give a multithreaded pseudo-code using the cilk language and expressing the algorithm presented in that wikipedia page

Solution. The section of the parallelization of the Floyd algorithm, in the wikipedia page, provides us with an interesting point of view. We can see Floyd-Warshall algorithm as a stencil computation, see Algorithm 3. Note that the parallel for-loops in Algorithm 3 can be expressed in the `cilk` language using `cilk_for` with the appropriate grain size.

3.3 Analyze its *work*, *span* and *parallelism* of your multithreaded pseudo-code.

Solution.

- Removing the two **in parallel** clauses yields a serial algorithm of work in $\Theta(N^3)$.
- The outermost loop and the two innermost loops are serial. This yields a span of $\Theta(N(2\log((N-1)/b))b^2)$. If ve wiew b as a small constant, we can simply answer $\Theta(N\log(N))$.

Algorithm 3: Parallel Floyd-Warshall algorithm using blocking

```
/* Let D be a |V| × |V| array of minimum distances initialized by the
   weighted directed graph G.                                            */
Define D^(0) = D and let N = |V| ;
Let b be an integer dividing N − 1 ;
for k = 0; k < N; ++k do
    Initialize an N × N matrix D^(k+1) to zero ;
    for i = 0; i ≤ (N − 1)/b; ++i ; in parallel do
        for j = 0; j ≤ (N − 1)/b; ++j ; in parallel do
            for h = 0; h < b; ++h do
                for ℓ = 0; ℓ < b; ++ℓ do
                    D^(k+1)_{i b+h, j b+ℓ} = min(D^(k)_{i b+h, j b+ℓ}, D^(k)_{i b+h, k} + D^(k)_{k, j b+ℓ})
    D^(k) = D^(k+1);
```

- Therefore, the parallelism in in $\Theta(N^2/\log(N))$.

PROBBLEM 4. [40 points]

Computational science is replete with algorithms that require the entries of an array to be filled in with values that depend on the values of certain already computed neighboring entries, along with other information that does not change over the course of the computation. The pattern of neighboring entries that does not change during the computation and is called a ***stencil***.

Consider a simple stencil calculation on an $n \times n$ array A in which, the value placed in to entry $A[i, j]$ depends on the average value of its neighbors: $A[i-1, j]$, $A[i+1, j]$, $A[i, j-1]$ and $A[i, j+1]$. The serial pseudo-code is shown in Algorithm 4.

Algorithm 4: A simple stencil calculation

```
/* An auxiliary array D is used.                                         */
for i = 1; i < n − 1; ++i do
    for j = 1; j < n − 1; ++j do
        D[i, j] = 0.25 * (A[i − 1, j] + A[i + 1, j] + A[i, j − 1] + A[i, j + 1]);
for i = 0; i < n; ++i do
    for j = 0; j < n; ++j do
        A[i, j] = D[i, j];
```

We can divide the $n \times n$ array A into four $n/2 \times n/2$ subarrays as,

$$A = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix},$$

and then recursively to update each subarray in parallel.

4.1 Based on this decomposition, give a multithreaded pseudo-code using a divide-and-conquer algorithm.

Solution.

4.2 Draw the computation dag of your pseudo-code, and show how to schedule the dag on 4 processors using greedy scheduling.

4.3 Give and solve recurrences for the *work* and *span* for this algorithm in terms of n. What is the *parallelism*?

Solution.

Copy part:

Work: $O(n^2)$

Span: $C_\infty(n) = C_\infty(n/2) + O(1) \in O(\log n)$

The whole algorithm:

Work: $O(n^2)$

Span: $S_\infty(n) = S_\infty(n/2) + O(\log n) = \Theta(\log^2 n)$

Parallelism: $O(n^2/\log^2 n)$

Choose an integer $b \geq 2$. Divide the $n \times n$ array into b^2 subarrays, each of size $n/b \times n/b$, recursing with as much parallelism as possible.

4.4 In terms of n and b, what are the *work*, *span* and *parallelism* of your algorithm?

Copy part:

Work: $O(n^2)$

Span: $C_\infty(n) = C_\infty(n/b) + O(1) \in O(\log_b n))$

The whole algorithm:

Work: $O(n^2)$

Span: $S_\infty(n) = S_\infty(n/b) + O(\log_b n) = \Theta(\log_b^2 n)$

Parallelism: $O(n^2/\log_b^2 n)$

Algorithm 5: Parallel Stencil

```
UPDATE(A, D, b, N)
    UPDATE-BLOCKS (A, D, b, 0, 0, N − 1, N − 1);
    COPY-BLOCKS (A, D, b, 0, 0, N − 1, N − 1);

UPDATE-BLOCKS(A, D, b, i_0, j_0, d_i, d_j)
    if d_i > b then
        d = d_i/2;
        spawn UPDATE-BLOCKS (A, D, b, i_0, j_0, d, d_j) ;
        UPDATE-BLOCKS (A, D, b, i_0 + d, j_0, d, d_j) ;
        return ;
    if d_j > b then
        d = d_j/2;
        spawn UPDATE-BLOCKS (A, D, b, i_0, j_0, d_i, d) ;
        UPDATE-BLOCKS (A, D, b, i_0, j_0 + d, d_i, d) ;
        return ;
    UPDATE-BLOCK(A, D, i_0, j_0, d_i, d_j)

COPY-BLOCKS(A, D, b, i_0, j_0, d_i, d_j)
    if d_i > b then
        d = d_i/2;
        spawn COPY-BLOCKS (A, D, b, i_0, j_0, d, d_j) ;
        COPY-BLOCKS (A, D, b, i_0 + d, j_0, d, d_j) ;
        return ;
    if d_j > b then
        d = d_j/2;
        spawn COPY-BLOCKS (A, D, b, i_0, j_0, d_i, d) ;
        COPY-BLOCKS (A, D, b, i_0, j_0 + d, d_i, d) ;
        return ;
    COPY-BLOCK(A, D, i_0, j_0, d_i, d_j)

UPDATE-BLOCK(A, D, i_0, j_0, d_i, d_j)
    for i = i_0, k < i_0 + d_i; ++i do
        for j = j_0, k < j_0 + d_j; ++j do
            D[i, j] = 0.25 * (A[i − 1, j] + A[i + 1, j] + A[i, j − 1] + A[i, j + 1]);

COPY-BLOCK(A, D, b, i_0, j_0)
    for i = i_0, k < i_0 + b; ++i do
        for j = j_0, k < j_0 + b; ++j do
            A[i, j] = D[i, j] ;
```

4.5 For any choice of $b \geq 2$, analyze the trends of *parallelism* and *burden parallelism.*

Parallelism grows but grows slower and slower.

Burdened parallelism grows slower than parallelism

4.6 Implement in C/C++ your multithreaded pseudocode from 4.1.

The code can be found in problem4/stencilDnC.cpp, For simplicity, the order of the matrix is set to $n + 2$ and we ignore the edge cells.

PROBBLEM 5. [20 points]

In the chapter *Analysis of Multithreaded Algorithms*, we studied the 2-way and 3-way construction of a tableau.

5.1 Describe, in plain words, how to construct a tableau in a k-way fashion, for an arbitrary integer $k \geq 2$, using the same stencil (the one of the Pascal triangle construction) as in the lectures.

One can use either a divide-and-conquer or a blocking strategy, as seen in class for Pascal's triangle.

5.2 Determine the work and the span for an input square array of order n.

For an input $n \times n$ array, the work is clearly in $\Theta(n^2)$ Let $S_k(n)$ be the non-burdened span for the k-way divide and conquer approach. We have:

$$S_k(n) = (2k-1)S_k(n/k) + \Theta(1) \in \Theta(n \log^{2k-1} k)$$

5.3 Determine the burdened span, similarly to what we did for the Pascal triangle construction at then of the chapter *Multithreaded Parallelism and Performance Measures*

$S_k(n) \in \Theta(\frac{n}{k} \log \frac{n}{k})$