Choose either Cilk or Cilk++, and download the corresponding zip file from Blackboard. Solve each problem below using Cilk/Cilk++ on the cs-parallel server. Use Cilk/Cilk++ parallelism constructs (cilk_spawn, cilk_sync, cilk_for) to develop efficient solutions. Do not use other Cilk/Cilk++ features (such as mutex, reducer, cilk_api). Do not modify the provided function signatures or file names. Test your functions using the example test cases in the provided zip file, and compare your outputs to the provided output files. You are encouraged to also create some additional examples to test more thoroughly. Compress your solutions into a zip file, and upload to Blackboard.

Your score on each problem will depend on both:

- **Correctness of the underlying sequential C or C++ program.**
- **Efficient use of parallelism without creating race conditions. (Ideally a parallel algorithm will take polylogarithmic or O(lg^k n) time for some constant k, on a machine with unlimited cores.)**

1. Write an efficient Cilk/Cilk++ function count(A, n, k) that counts the number of occurrences of the key value k within the 2-dimensional matrix A which has size n-by-n.

```
int count (int **A, int n, int k) {

}
```

2. Write an efficient Cilk/Cilk++ function multpoly(p, m, q, n) that multiplies two polynomials $P(x) = \Sigma_{0\le k\le m}\, p_k\, x^k$ and $Q(x) = \Sigma_{0\le k\le n}\, q_k\, x^k$ as follows. Compute the product $R(x) = P(x) * Q(x) = \Sigma_{0\le k\le m+n}\, r_k\, x^k$. The input parameters include the coefficient arrays p[0…m] and q[0…n], and the return value is the array r[0…m+n].

```
int *multpoly (int *p, int m, int *q, int n) {

}
```

3. Write an efficient Cilk/Cilk++ function fun(f, g, p, id, A, low, high) that applies unary function g to each element of array A[low…high], then determines which of those results satisfy predicate p, and then combines all such results using binary function f. However, if none of those results satisfy p, then return id. You may assume all primitive types are ints. Example shown below.

int f (int x, int y) { return x+y; }
int g (int x) { return x*x; }
int p (int x) { return x>10; }
int A[ ] = {2,6,3,5,-3,-4,1,-2};
fun (f, g, p, 0, A, 0, 7) returns $6^2 + 5^2 + (-4)^2 = 77$.

```
int fun (int (*f)(int,int), int (*g)(int), int (*p)(int), int id, int *A, int low, int high) {

}
```

4. Write an efficient Cilk/Cilk++ function enumsort(A, n) that sorts array A[0…n–1] using the enumeration sort algorithm as follows: First compare each pair of elements in A, then use the results of those comparisons to determine the rank of each element, and finally use the rank information to move each element to its correct position. Example:

    int A[ ] = {20,40,30,20,10,30};
    enumsort (A, 6); // now A contains {10,20,20,30,30,40}

A		comparison results							ranks		A
20		0	0	0	0	1	0		1		10
40		1	0	1	1	1	1		5		20
30	→	1	0	0	1	1	0	→	3	→	20
20		1	0	0	0	1	0		2		30
10		0	0	0	0	0	0		0		30
30		1	0	1	1	1	0		4		40

```
void enumsort (int *A, int n) {

}
```

5. Write an efficient Cilk/Cilk++ function Floyd(d, n) that implements Floyd's all-pairs shortest paths algorithm as follows. Let G = (V,E) denote a weighted directed graph with V = {1,2,…,n} and weights w(u,v) such that no negative cycles exist. Define $d_k[i][j]$ = minimum total weight along any path from i to j that does not pass through any intermediate vertex > k. Hence $d_0[i][j] = w(i,j)$ and $d_n[i][j]$ = length of shortest path from i to j. The function Floyd(d, n) computes all the $d_k[i][j]$ values for $1 \leq k \leq n$.

```
void Floyd (int ***d, int n) {

}
```

6. Write an efficient Cilk/Cilk++ function APSP(D, n) that implements all-pairs shortest paths without using Floyd's algorithm as follows. Let G = (V,E) denote a weighted directed graph with V = {1,2,…,n} and weights w(u,v) such that no negative cycles exist. Define $D_m[i][j]$ = minimum total weight along any path from i to j that contains ≤ m edges. Hence $D_1[i][j]$ = w(i,j) and $D_m[i][j]$ = length of shortest path from i to j whenever $m \geq n-1$. The function APSP(D, n) uses this idea to compute all the $D_m[i][j]$ values when m = n. Hints: You may assume that n is a power of 2. Given an array A[1..n][1..n] and specified row i and column j, write an efficient recursive divide-and-conquer helper function that computes min $\{A[i][k] + A[k][j] : 1 \leq k \leq n\}$.

```
void APSP (int ***D, int n) {

}
```