```
//#define _CRT_SECURE_NO_WARNINGS
//#define WIN32_LEAN_AND_MEAN
//#include "stdafx.h"
// #pragma omp parallel for reduction(+:sum) private(x)

#include <omp.h>
#include <stdio.h>
#include <stdlib.h>
#include <windows.h>

#define SLOW_DOWN_EXECUTION

// A*X*X*X*X*X*X*X + B*X*X*X*X*X*X + C*X*X*X*X*X + D*X*X*X*X + E*X*X*X + F*X*X + G*X + H
#define A 4
#define X 5
#define B 7
#define C 10
#define D 13
#define E 16
#define F 19
#define G 22
#define H 25

#define RESULT ( A * X * X * X * X * X * X * X + B * X * X * X * X * X * X + C * X * X * X
* X * X + D * X * X * X * X * X + E * X * X * X + F * X * X + G * X + H )

// (f0:A*X**7) (f1:B*X**6) (f2:C*X**5) (f3:D*X**4) (f4:E*X**3) (f5:F*X**2) (f6:G*X) (f7:H)
//            ||                      ||                      ||                    ||
//            \/                      \/                      \/                    \/
//          (f8:+)                  (f9:+)                  (f10:+)               (f11:+)
//                      ||                                              ||
//                      \/                                              \/
//                    (f12:+)                                         (f13:+)
//                                            ||
//                                            \/
//                                          (f14:+)

// stage with index 0

void f0(int * argArr, int * resArr) {
    if(/*argArr && */resArr){
        resArr[0] = A * X * X * X * X * X * X * X;
       }
#ifdef SLOW_DOWN_EXECUTION
    Sleep(1000);
#endif
}

void f1(int * argArr, int * resArr) {
    if(/*argArr && */resArr){
        resArr[1] = B * X * X * X * X * X * X;
       }
#ifdef SLOW_DOWN_EXECUTION
    Sleep(1000);
#endif
}

void f2(int * argArr, int * resArr) {
    if(/*argArr && */resArr){
        resArr[2] = C * X * X * X * X * X;
       }
#ifdef SLOW_DOWN_EXECUTION
    Sleep(1000);
#endif
}
```

```
void f3(int * argArr, int * resArr) {
    if(/*argArr && */resArr){
        resArr[3] = D * X * X * X * X;
      }
#ifdef SLOW_DOWN_EXECUTION
    Sleep(1000);
#endif
}

void f4(int * argArr, int * resArr) {
    if(/*argArr && */resArr){
        resArr[4] = E * X * X * X;
      }
#ifdef SLOW_DOWN_EXECUTION
    Sleep(1000);
#endif
}

void f5(int * argArr, int * resArr) {
    if(/*argArr && */resArr){
        resArr[5] = F * X * X;
      }
#ifdef SLOW_DOWN_EXECUTION
    Sleep(1000);
#endif
}

void f6(int * argArr, int * resArr) {
    if(/*argArr && */resArr){
        resArr[6] = G * X;
      }
#ifdef SLOW_DOWN_EXECUTION
    Sleep(1000);
#endif
}

void f7(int * argArr, int * resArr) {
    if(/*argArr && */resArr){
        resArr[7] = H;
      }
#ifdef SLOW_DOWN_EXECUTION
    Sleep(1000);
#endif
}

// stage with index 1

void f8(int * argArr, int * resArr) {
    if(argArr && resArr){
        resArr[0] = argArr[0] + argArr[1];
      }
#ifdef SLOW_DOWN_EXECUTION
    Sleep(1000);
#endif
}

void f9(int * argArr, int * resArr) {
    if(argArr && resArr){
        resArr[2] = argArr[2] + argArr[3];
      }
#ifdef SLOW_DOWN_EXECUTION
    Sleep(1000);
#endif
}
```

```
void f10(int * argArr, int * resArr) {
    if(argArr && resArr){
        resArr[4] = argArr[4] + argArr[5];
       }
#ifdef SLOW_DOWN_EXECUTION
    Sleep(1000);
#endif
}

void f11(int * argArr, int * resArr) {
    if(argArr && resArr){
        resArr[6] = argArr[6] + argArr[7];
       }
#ifdef SLOW_DOWN_EXECUTION
    Sleep(1000);
#endif
}

// stage with index 2

void f12(int * argArr, int * resArr) {
    if(argArr && resArr){
        resArr[0] = argArr[0] + argArr[2];
       }
#ifdef SLOW_DOWN_EXECUTION
    Sleep(1000);
#endif
}

void f13(int * argArr, int * resArr) {
    if(argArr && resArr){
           resArr[4] = argArr[4] + argArr[6];
       }
#ifdef SLOW_DOWN_EXECUTION
    Sleep(1000);
#endif
}

// stage with index 3

void f14(int * argArr, int * resArr) {
    if(argArr && resArr){
           resArr[0] = argArr[0] + argArr[4];
       }
#ifdef SLOW_DOWN_EXECUTION
    Sleep(1000);
#endif
}

#define MAX_STAGE_COUNT 10
#define MAX_PE_COUNT 10

void(*fArr[MAX_STAGE_COUNT][MAX_PE_COUNT])(int * argArr, int * resArr) = {
       { f0,  f1,   f2,   f3,   f4,   f5,   f6,   f7   },
       { f8,  NULL, f9,   NULL, f10,  NULL, f11,  NULL },
       { f12, NULL, NULL, NULL, f13,  NULL, NULL, NULL },
       { f14, NULL, NULL, NULL, NULL, NULL, NULL, NULL },
};
#define LAST_STAGE_INDEX 3

void compute(){
       int argArr[MAX_PE_COUNT * 2] = { 0 }, resArr[MAX_PE_COUNT * 2] = { 0 }, iIndex,
jIndex;
       for (iIndex = 0; iIndex < MAX_STAGE_COUNT; ++iIndex) {
```

```
        #pragma omp parallel for shared(argArr, resArr)
              for (jIndex = 0; jIndex < MAX_PE_COUNT; ++jIndex) {
                    if (!fArr[iIndex][jIndex]) continue;
                    if (!(iIndex % 2)) fArr[iIndex][jIndex](argArr, resArr);
                    else fArr[iIndex][jIndex](resArr, argArr);
              }
       }

       int * res = (LAST_STAGE_INDEX + 1) % 2 ? resArr : argArr;
       printf("result of execution = %d \r\n", *res);
       printf("expected result     = %d \r\n", RESULT);
       printf("--------------------------------\r\n");
       if (*res == RESULT){
              printf("verify status: succes\r\n");
       }
       else{
              printf("verify status: not success\r\n");
       }
}

int main(int argc, char* argv[]) {
       compute();

       return 0;
}
```