1. Realiza el código utilizando sólo tareas implícitas, y creando una estructura de datos de sincronización, similar a lo realizado en el Lab5 para tener en cuenta la dependencia de datos. (2,5p)

#define N 8

void process(int A[N][N]) {

// Crear un array de 'omp\_lock\_t' para manejar la sincronización

omp\_lock\_t locks[N][N];

int i, k;

// Inicializar los locks

for (i = 0; i < N; i++) {

for (k = 0; k < N; k++) {

omp\_init\_lock(&locks[i][k]);

if (k > 0) {

omp\_set\_lock(&locks[i][k]); // Bloquear todos excepto la primera columna

}

}

}

#pragma omp parallel num\_threads(8)

{

#pragma omp for collapse(2) // Colapsamos los dos bucles for en una única partición de trabajo

for (i = 0; i < N; i++) {

for (k = 1; k < N; k++) {

omp\_set\_lock(&locks[i][k-1]); // Esperar a que el valor de A[i][k-1] esté listo

int tmp = A[i][k] - A[i][k-1];

A[i][k] = tmp / 2;

omp\_unset\_lock(&locks[i][k]); // Desbloquear para el siguiente procesador

// En la primera iteración de k, no hay dependencia de la fila anterior

if (k == 1 && i > 0) {

// Esperar a que el último elemento de la fila anterior esté listo

omp\_set\_lock(&locks[i-1][N-1]);

}

// Operación que depende de A[i][k-1]

int tmp = A[i][k] - A[i][k-1];

A[i][k] = tmp / 2;

// Desbloquear para la próxima iteración o para el próximo procesador

omp\_unset\_lock(&locks[i][k]);

if (k == N-1) {

// Si estamos en la última columna, desbloqueamos la primera columna de la siguiente fila

if (i < N-1) {

omp\_unset\_lock(&locks[i+1][0]);

}

}

}

}

}

// Destruir los locks al finalizar

for (i = 0; i < N; i++) {

for (k = 0; k < N; k++) {

omp\_destroy\_lock(&locks[i][k]);

}

}

1. Realiza una paralelización utilizando tareas explícitas de manera que cada bloque de la figura sea una tarea (#pragma omp task), y resolviendo la dependencia de datos con depend. (1p)

#define N

int A[N][N];

void process\_block(int start\_row, int end\_row, int start\_col, int end\_col) {

for (int i = start\_row; i < end\_row; i++) {

for (int k = start\_col; k < end\_col; k++) {

if (k > 0) {

int tmp = A[i][k] - A[i][k-1];

A[i][k] = tmp / 2;

}

}

}

}

int main() {

// Inicializamos A

#pragma omp parallel

#pragma omp single

{

// Tarea para P0

#pragma omp task depend(out: A[0:N/4][1:N/4])

process\_block(0, N/4, 1, N/4);

// Tarea para P1

#pragma omp task depend(in: A[0:N/4][0:N/4-1]) depend(out: A[0:N/4][N/4:2\*N/4])

process\_block(0, N/4, N/4, 2\*N/4);

// Tarea para P2

#pragma omp task depend(in: A[0:N/4][N/4-1:2\*N/4-1]) depend(out: A[0:N/4][2\*N/4:3\*N/4])

process\_block(0, N/4, 2\*N/4, 3\*N/4);

// Tarea para P3

#pragma omp task depend(in: A[0:N/4][2\*N/4-1:3\*N/4-1]) depend(out: A[0:N/4][3\*N/4:N])

process\_block(0, N/4, 3\*N/4, N);

// Tarea para P4

#pragma omp task depend(out: A[N/4:2\*N/4][1:N/4])

process\_block(N/4, 2\*N/4, 1, N/4);

// Tarea para P5

#pragma omp task depend(in: A[N/4:2\*N/4][0:N/4-1]) depend(out: A[N/4:2\*N/4][N/4:2\*N/4])

process\_block(N/4, 2\*N/4, N/4, 2\*N/4);

// Tarea para P6

#pragma omp task depend(in: A[N/4:2\*N/4][N/4-1:2\*N/4-1]) depend(out: A[N/4:2\*N/4][2\*N/4:3\*N/4])

process\_block(N/4, 2\*N/4, 2\*N/4, 3\*N/4);

// Tarea para P7

#pragma omp task depend(in: A[N/4:2\*N/4][2\*N/4-1:3\*N/4-1]) depend(out: A[N/4:2\*N/4][3\*N/4:N])

process\_block(N/4, 2\*N/4, 3\*N/4, N);

}

#pragma omp taskwait // Esperar a que todas las tareas terminen

}