TPP – Tecnología de la Programación Paralela

Master Universitario en Computación en la Nube y de Altas Prestaciones

Diseño de Algoritmos Paralelos en Memoria Compartida

Pedro Alonso

Departamento de Sistemas Informáticos y Computación
Universitat Politècnica de València

UNIVERSITAT POLITÈCNICA DE VALÈNCIA

DSIC DEPARTAMENTO DE SISTEMAS INFORMÁTICOS Y COMPUTACIÓN

Contenido

Apartado 1

Implementación de algoritmos en memoria compartida

- Caso de Estudio: Sudoku
- Caso de Estudio: Descomposición de Cholesky
- Caso de Estudio: MIMO
- Caso de Estudio: Sudoku
- Caso de Estudio: Descomposición de Cholesky
- Caso de Estudio: MIMO

Apartado 2

Introducción

- Caso de Estudio: Sudoku
- Caso de Estudio: Descomposición de Cholesky
- Caso de Estudio: MIMO
- Caso de Estudio: Sudoku
- Caso de Estudio: Descomposición de Cholesky
- Caso de Estudio: MIMO

Introducción al curso

1. Algoritmos de búsqueda de una solución.
   Casos particulares:
   - El juego del Sudoku.
   - El problema de optimización MIMO.

   Herramientas hardware: Multicore.

   Herramientas software: OpenMP.
2. Algoritmos de cálculo sobre muchos datos.
   Casos particulares:
   - La factorización de Cholesky.

   Herramientas hardware: Multicore y aceleradores.

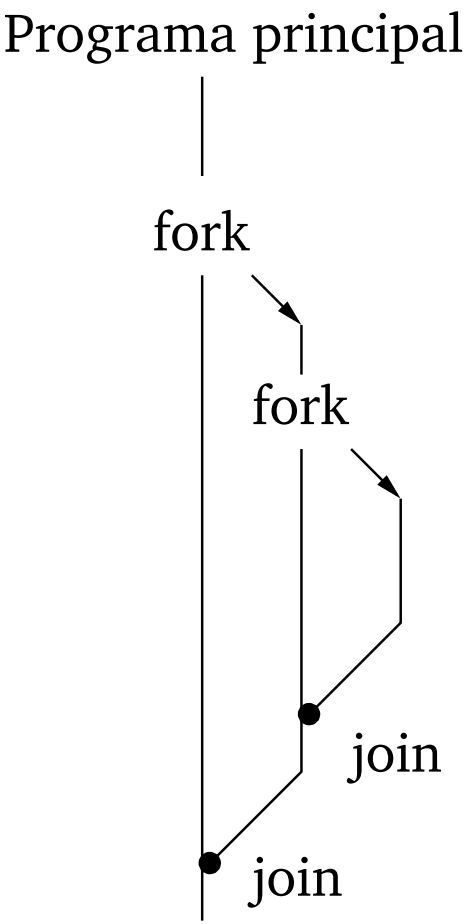
   Herramientas software: OpenMP.
3. Otras herramientas software: Threading Building Blocks (TBB).

Apartado 3

Modelo de memoria compartida

- Caso de Estudio: Sudoku
- Caso de Estudio: Descomposición de Cholesky
- Caso de Estudio: MIMO
- Caso de Estudio: Sudoku
- Caso de Estudio: Descomposición de Cholesky
- Caso de Estudio: MIMO

Procesos concurrentes



Para especificar procesos concurrentes es habitual utilizar construcciones de tipo fork-join.

- *Fork* crea una nueva tarea concurrente que empieza a ejecutar en el mismo punto en que estaba la tarea padre
- *Join* espera a que termine la tarea.
- Ejemplo: llamada al sistema `fork()` en Unix.

Este esquema se puede implementar a nivel de:

- Procesos del sistema operativo (procesos pesados).
- Hilos (procesos ligeros).

Modelo de Memoria Compartida

Características:

- Espacio de direcciones de memoria único para todos
- Programación bastante similar al caso secuencial
  - Cualquier dato es accesible por cualquiera
  - No hay que intercambiar datos explícitamente
- Inconvenientes
  - El acceso concurrente a memoria puede dar problemas
    - Se ha de coordinar: semáforos, monitores, ...
    - Resultado impredecible si no se protegen bien los accesos a memoria
  - Difícil controlar la localidad de datos (memorias cache)

Modelo de Hilos

Este modelo está muy ligado al de memoria compartida
Hilo (thread): flujo de instrucciones independiente que puede ser planificado para ejecución por el sistema operativo.

- Un proceso puede tener múltiples hilos de ejecución
- Cada hilo tiene datos "privados"
- Comparten recursos/memoria del proceso
- Se requiere sincronización

Hilos POSIX (`pthreads`)

Estandarización de hilos en sistemas Unix (estándar IEEE POSIX 1003.1c, 1995).

- Basado en librería (API de llamadas al S.O.).
- Sólo lenguaje C/C++.
- Paralelismo explícito: bastante esfuerzo de programación.
- Difícil de programar: propenso a interbloqueos.

Algunas operaciones

- Creación: `pthread_create`, `pthread_join`.
- Semáforos: `sem_wait`, `sem_post`.
- Exclusión mutua: `mutex_lock`, `mutex_unlock`.
- Variables condición: `pthread_cond_wait`, `pthread_cond_signal`, `pthread_cond_broadcast`.

OpenMP

Estandarización de hilos portable.

- Basado en directivas de compilador.
- Disponible en C/C++ y Fortran.
- Portable/multi-plataforma (Unix, Windows).
- Fácil de usar: paralelización incremental.

Algunas directivas y funciones

- `#pragma omp parallel for`
- `omp_get_thread_num()`

La creación y finalización de hilos está implícita en algunas directivas

- El programador no se ha de preocupar de hacer fork/join.

Procesos UNIX

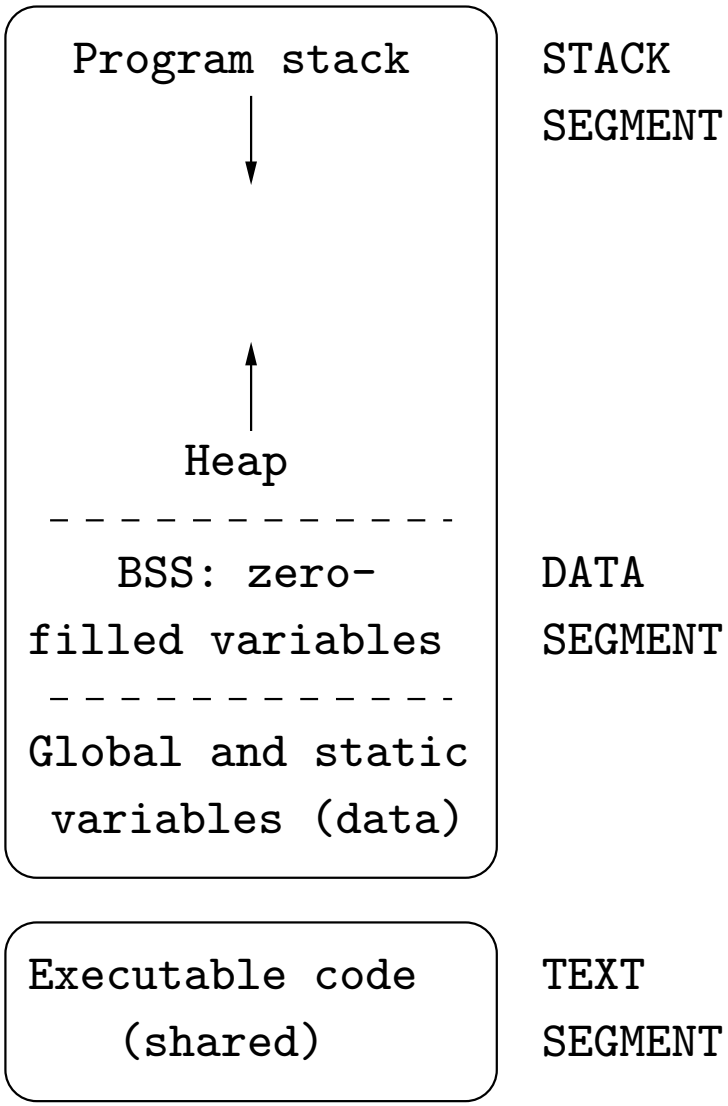
Cada proceso contiene información sobre recursos y estado de ejecución:

- Código del programa (solo lectura, puede ser compartido).
- Variables (globales, heap y stack).
- Contexto de ejecución: registros, puntero de pila, etc.
- Recursos del sistema (solo accesible a través del S.O.).
  - Identificadores (proceso, usuario, grupo).
  - Entorno, directorio de trabajo, señales.
  - Descriptores de ficheros.

En procesos multi-hilo:

- Cada hilo tiene su propio contexto de ejecución.
- Cada hilo tiene una pila de llamadas independiente.
- Se comparten los recursos del sistema.

Procesos UNIX

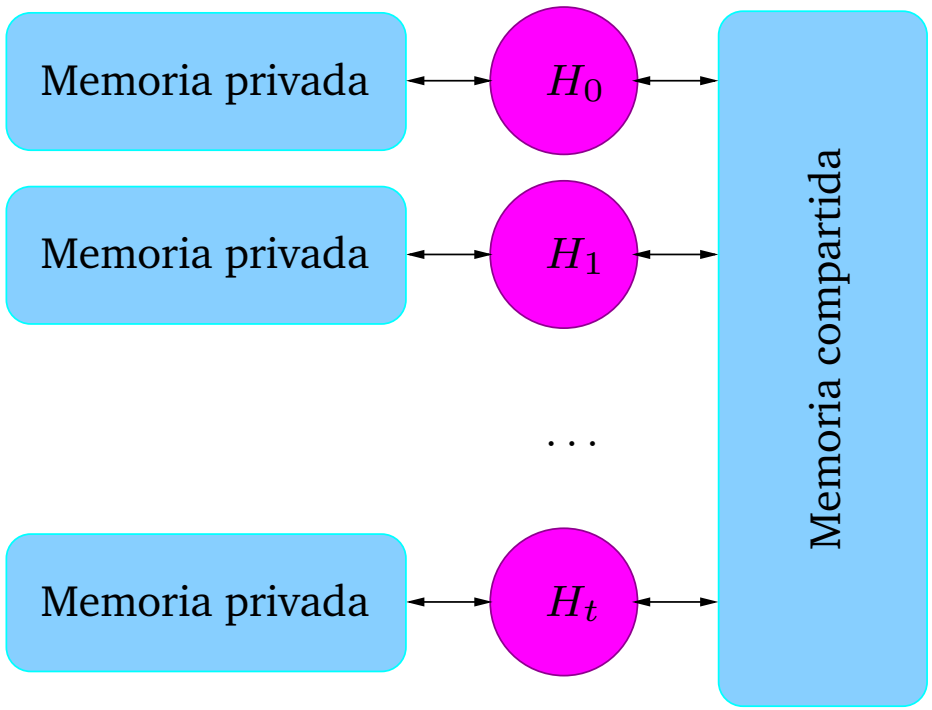


Información en el núcleo del sistema operativo (PCB: process control block)

- Program counter
- Stack pointer
- Registers
- Process state
- Process ID
- User ID
- Group ID
- Memory limits
- Open files, sockets
- ...

Modelo de Memoria con Hilos

Espacio único de direcciones con variables privadas por cada hilo.



Una pila de llamadas por cada hilo

- Algunas variables se crean en la pila (locales)
- Un hilo no puede saber si la pila de otro hilo está activa

Coordinación de Accesos a Memoria

El intercambio de información entre hilos se hace mediante lectura/escritura de variables en memoria compartida.

- El acceso simultáneo puede producir una condición de carrera.
- El resultado final puede ser incorrecto y es de naturaleza no determinista.

Solución:

1. Operaciones atómicas
   - Operaciones sencillas que se realizan sin interrupción (++, =+, ...).
   - Instrucciones especiales del procesador: `compare_and_swap`.
2. Sección crítica
   - Fragmentos de código con más de una instrucción
   - No permitir que haya más de un hilo ejecutándola
   - Requiere mecanismos de sincronización: semáforos, etc.
   - Puede aparecer riesgo de interbloqueo

Apartado 4

Diseño de algoritmos paralelos basado en hilos

- Algoritmos de búsqueda de una solución en un espacio
  - Caso de Estudio: Sudoku
- Algoritmos basados en particionado de datos
  - Caso de Estudio: Descomposición de Cholesky
- Algoritmos de búsqueda de la solución óptima en un espacio
  - Caso de Estudio: MIMO
  - Caso de Estudio: Sudoku
  - Caso de Estudio: Descomposición de Cholesky
  - Caso de Estudio: MIMO

Diseño de algoritmos paralelos basado en hilos

- Algoritmos de búsqueda de una solución en un espacio
  - Caso de Estudio: Sudoku
- Algoritmos basados en particionado de datos
  - Caso de Estudio: Descomposición de Cholesky
- Algoritmos de búsqueda de la solución óptima en un espacio
  - Caso de Estudio: MIMO

Grafos de dependencias de tareas

Abstracción utilizada para expresar las dependencias entre las tareas y su relativo orden de ejecución.

- Se trata de un grafo acíclico dirigido (DAG o Directed Acyclic Graph).
- Los nodos representan tareas (pueden tener asociado un coste).
- Las aristas representan las dependencias entre tareas.

Definiciones:

- Longitud de un camino: suma de los costes c_i de los nodos que lo componen
- Camino crítico: el más largo entre un nodo inicial y uno final
- Máximo grado de concurrencia: mayor número de tareas que pueden ejecutarse al mismo tiempo
- Grado medio de concurrencia: $M = \sum_{i=1}^{N} \frac{c_i}{L}$.
  (N = nodos totales, L = longitud del camino crítico)

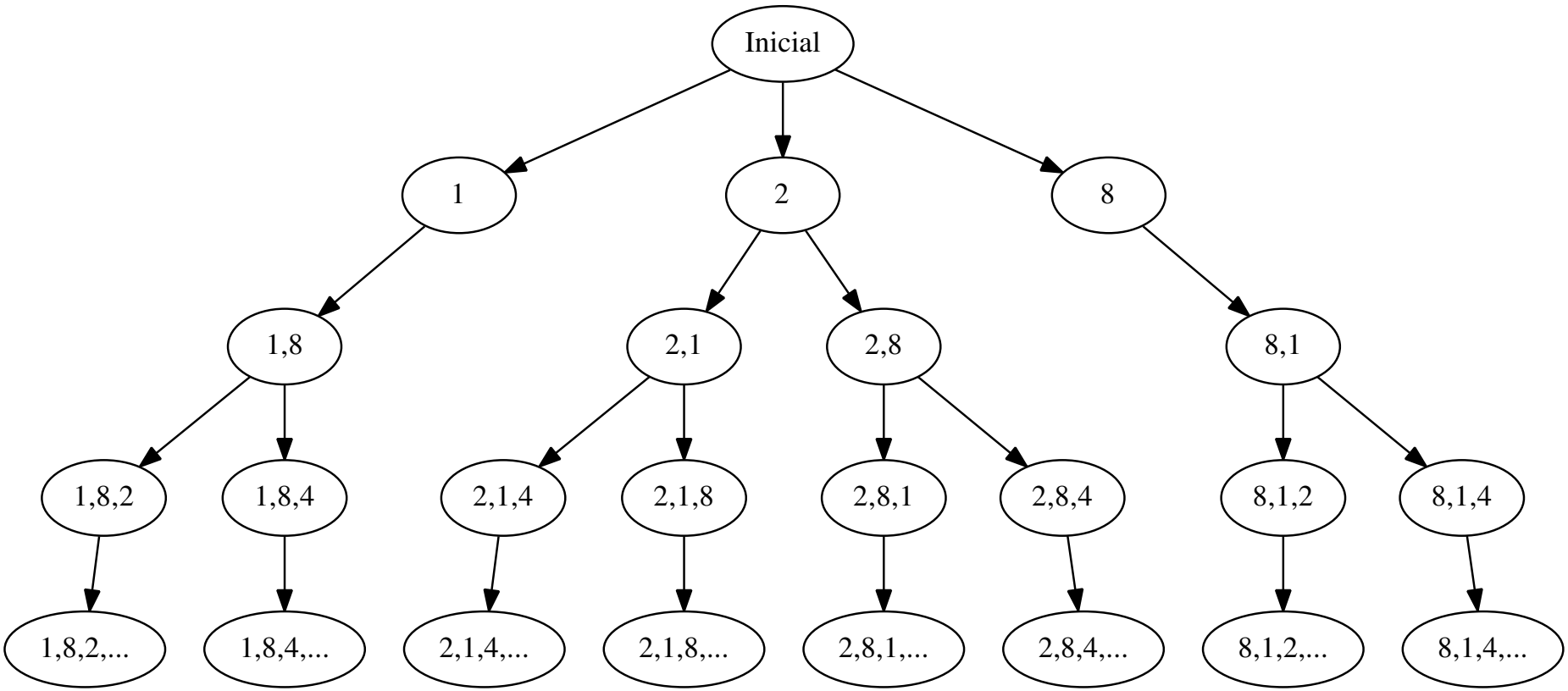
Ejemplo: Sudoku

			7				5	3
	9		3					4
		6		4		8		
					5		9	
4	7			9	6			
9		5		3				
	2	7	5				3	
	4	9	2				8	7
5		3				2	4	

Sudoku: Algoritmo *backtracking*

```
#define sol(a,b) sol[(a-1)*9+(b-1)]
#define mascara(a,b) mascara[(a-1)*9+(b-1)]
void sudoku_sol( int i, int j, int sol[81], int mascara[81] ) {
   int k;
   if( mascara(i, j) == 0 ) {
      for( k = 1; k <= 9; k++ ) {
         sol( i, j ) = k;
         if( es_factible( i, j, sol ) ) {
            if( j < 9 ) {
               sudoku_sol( i, j+1, sol, mascara );
            } else if( i < 9 ) {
               sudoku_sol ( i+1, 1, sol, mascara );
            } else {
               printf("Solucion: \n");
               prin_sudoku(sol);
            }
         }
      }
      sol(i, j) = 0;
   } else {
      if( j < 9 ) {
         sudoku_sol( i , j+1, sol, mascara );
      } else if( i < 9 ) {
         sudoku_sol ( i+1, 1, sol, mascara );
      } else {
         printf("Solucion: \n");
         prin_sudoku(sol);
      }
   }
}
```

Sudoku: DAG


Inicial
1
2
8
1,8
2,1
2,8
8,1
1,8,2
1,8,4
2,1,4
2,1,8
2,8,1
2,8,4
8,1,2
8,1,4
1,8,2,...
1,8,4,...
2,1,4,...
2,1,8,...
2,8,1,...
2,8,4,...
8,1,2,...
8,1,4,...

Sudoku: DAG

Características del DAG del Sudoku

- ¿Es necesario recorrer todos los nodos para encontrar la solución? No necesariamente.
- ¿Cuántos hijos tiene cada nodo? Entre 0 y 9.
- ¿Es regular el grafo? No. Cada nodo tiene diferente número hijos y cada rama puede tener una profundidad diferente.
- ¿Qué profundidad máxima tiene el grafo? Igual al número de casillas vacías del tablero inicial.
- ¿Se puede paralelizar? Sí. Se pueden evaluar los hijos de cada nodo independientemente.
- ¿Cuál es el camino crítico? El compuesto por los nodos que conducen a la solución.

Sudoku: DAG

Características del DAG del Sudoku

- ¿Es necesario recorrer todos los nodos para encontrar la solución? No necesariamente.
- ¿Cuántos hijos tiene cada nodo? Entre 0 y 9.
- ¿Es regular el grafo? No. Cada nodo tiene diferente número hijos y cada rama puede tener una profundidad diferente.
- ¿Qué profundidad máxima tiene el grafo? Igual al número de casillas vacías del tablero inicial.
- ¿Se puede paralelizar? Sí. Se pueden evaluar los hijos de cada nodo independientemente.
- ¿Cuál es el camino crítico? El compuesto por los nodos que conducen a la solución.

Sudoku: DAG

Características del DAG del Sudoku

- ¿Es necesario recorrer todos los nodos para encontrar la solución? No necesariamente.
- ¿Cuántos hijos tiene cada nodo? Entre 0 y 9.
- ¿Es regular el grafo? No. Cada nodo tiene diferente número hijos y cada rama puede tener una profundidad diferente.
- ¿Qué profundidad máxima tiene el grafo? Igual al número de casillas vacías del tablero inicial.
- ¿Se puede paralelizar? Sí. Se pueden evaluar los hijos de cada nodo independientemente.
- ¿Cuál es el camino crítico? El compuesto por los nodos que conducen a la solución.

Sudoku: DAG

Características del DAG del Sudoku

- ¿Es necesario recorrer todos los nodos para encontrar la solución? No necesariamente.
- ¿Cuántos hijos tiene cada nodo? Entre 0 y 9.
- ¿Es regular el grafo? No. Cada nodo tiene diferente número hijos y cada rama puede tener una profundidad diferente.
- ¿Qué profundidad máxima tiene el grafo? Igual al número de casillas vacías del tablero inicial.
- ¿Se puede paralelizar? Sí. Se pueden evaluar los hijos de cada nodo independientemente.
- ¿Cuál es el camino crítico? El compuesto por los nodos que conducen a la solución.

Sudoku: DAG

Características del DAG del Sudoku

- ¿Es necesario recorrer todos los nodos para encontrar la solución? No necesariamente.
- ¿Cuántos hijos tiene cada nodo? Entre 0 y 9.
- ¿Es regular el grafo? No. Cada nodo tiene diferente número hijos y cada rama puede tener una profundidad diferente.
- ¿Qué profundidad máxima tiene el grafo? Igual al número de casillas vacías del tablero inicial.
- ¿Se puede paralelizar? Sí. Se pueden evaluar los hijos de cada nodo independientemente.
- ¿Cuál es el camino crítico? El compuesto por los nodos que conducen a la solución.

Sudoku: DAG

Características del DAG del Sudoku

- ¿Es necesario recorrer todos los nodos para encontrar la solución? No necesariamente.
- ¿Cuántos hijos tiene cada nodo? Entre 0 y 9.
- ¿Es regular el grafo? No. Cada nodo tiene diferente número hijos y cada rama puede tener una profundidad diferente.
- ¿Qué profundidad máxima tiene el grafo? Igual al número de casillas vacías del tablero inicial.
- ¿Se puede paralelizar? Sí. Se pueden evaluar los hijos de cada nodo independientemente.
- ¿Cuál es el camino crítico? El compuesto por los nodos que conducen a la solución.

Sudoku: DAG

Características del DAG del Sudoku

- ¿Es necesario recorrer todos los nodos para encontrar la solución? No necesariamente.
- ¿Cuántos hijos tiene cada nodo? Entre 0 y 9.
- ¿Es regular el grafo? No. Cada nodo tiene diferente número hijos y cada rama puede tener una profundidad diferente.
- ¿Qué profundidad máxima tiene el grafo? Igual al número de casillas vacías del tablero inicial.
- ¿Se puede paralelizar? Sí. Se pueden evaluar los hijos de cada nodo independientemente.
- ¿Cuál es el camino crítico? El compuesto por los nodos que conducen a la solución.

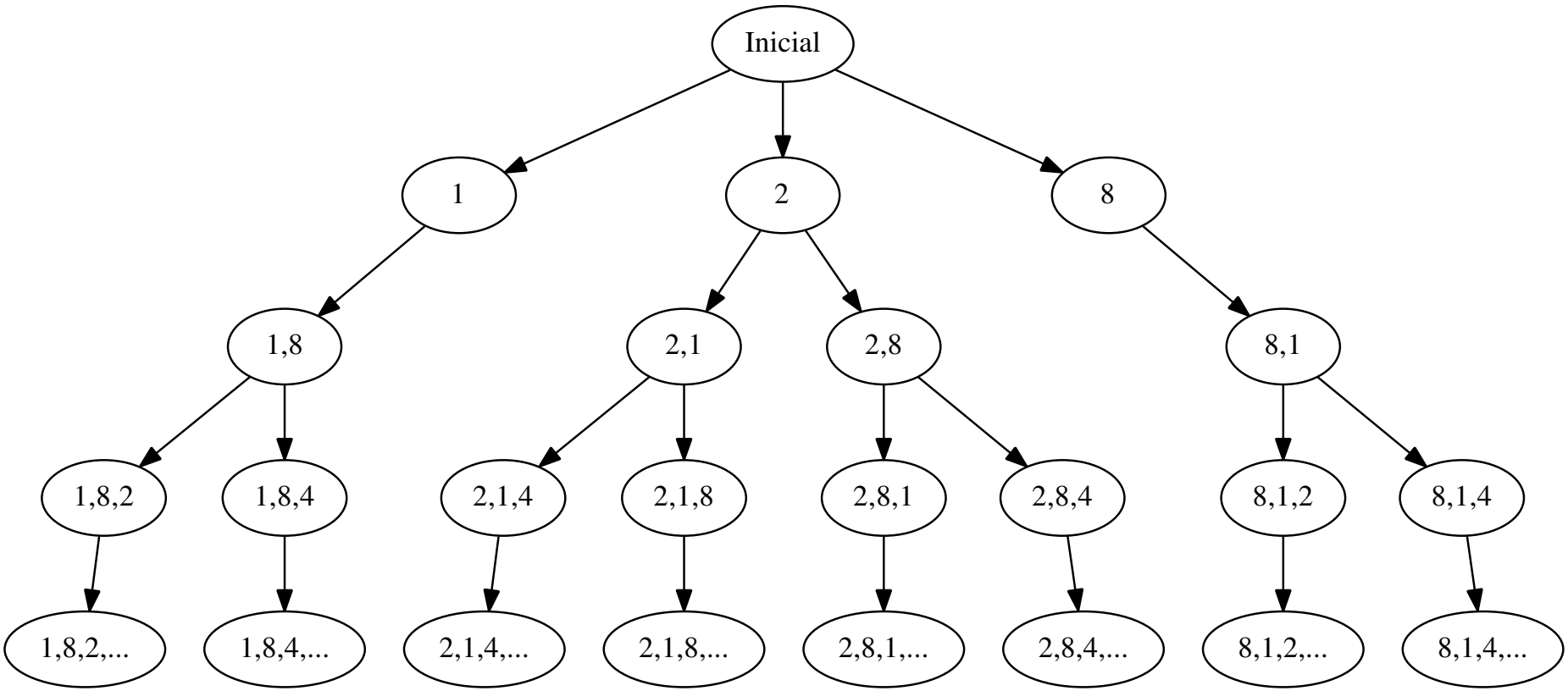
Sudoku: Conversión recursivo iterativa

Recursivo–iterativo

```
int A[3000][81];
int B[3000][81];
int nivel, nodo, k, l;
for(int l = 0; l < 81; l++) A[0][l] = sol[l];
int tableros = 1;
for( int nivel = 0; nivel < profundidad; nivel ++){
  int j = 0;
  for( int nodo = 0; nodo < tableros; nodo++ ) {
    int k = 0; while( k < 81 && A[nodo][k] != 0 ) k++;
    if( k<81 ) {
      for( int i=1; i<=9; i++ ) {
        A[nodo][k] = i;
        if( es_factible( k/9+1, k%9+1, A[nodo] ) ) {
          for( int l = 0; l<81; l++ ) { B[j][l] = A[nodo][l]; }
          j++;
        }
        A[nodo][k] = 0;
      }
    }
  }
  tableros = j;
  for( int i = 0; i<tableros; i++ )
    for( int k = 0; k<81; k++ )
      A[i][k] = B[i][k];
```

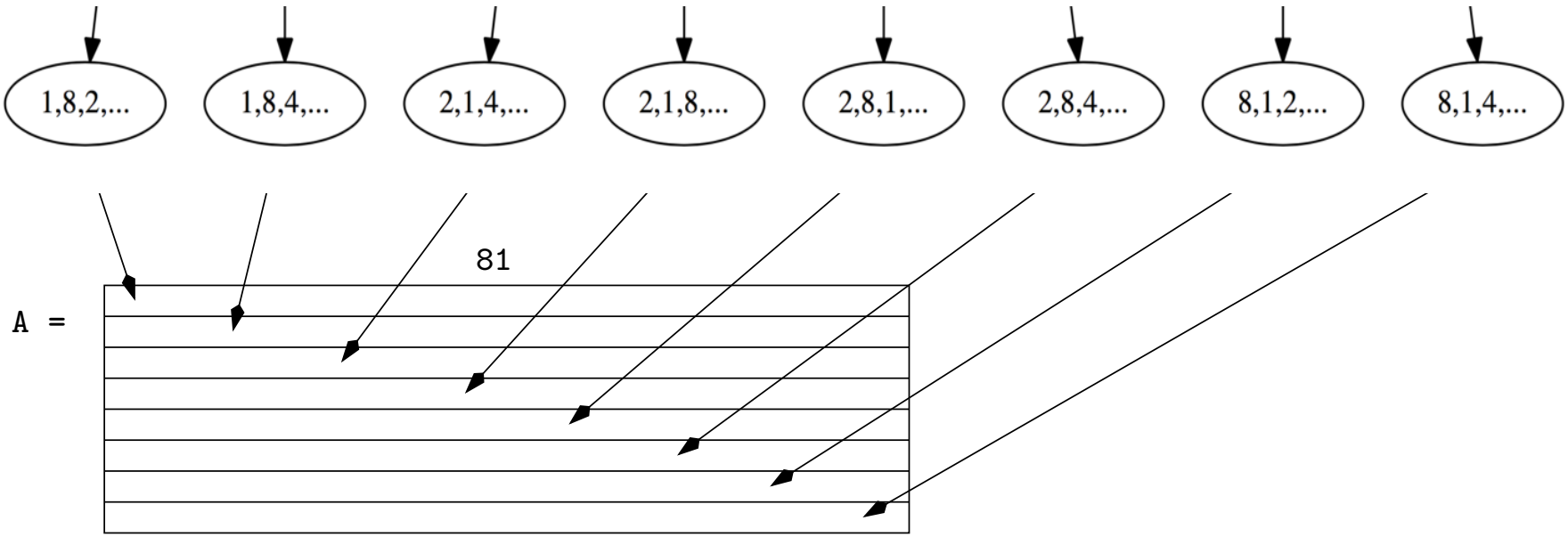
La matriz `A` contiene en cada fila un tablero del nivel `profundidad`.

Sudoku: DAG


Inicial
1
2
8
1,8
2,1
2,8
8,1
1,8,2
1,8,4
2,1,4
2,1,8
2,8,1
2,8,4
8,1,2
8,1,4
1,8,2,...
1,8,4,...
2,1,4,...
2,1,8,...
2,8,1,...
2,8,4,...
8,1,2,...
8,1,4,...

Sudoku: Solución paralela basada en paralelismo de hilos

Resolución de los tableros de la matriz `A` en el nivel `profundidad`:



Bucle a paralelizar

```
for(int tablero = 0; tablero < tableros; tablero++) {
  int mascara[81];
  for ( int i = 0; i < 81; i++ ) mascara[i] = A[tablero][i] != 0;
  sudoku_sol(1,1,A[tablero],mascara);
}
```

Sudoku: Análisis de costes

- Tiempo secuencial: T_s
- Tiempo paralelo: T_p

$$T_p = t_s + \frac{t_p}{p}$$

donde:

- t_s es tiempo secuencial de generar la matriz de tableros,
- t_p tiempo paralelizable.
- p es el número de hilos.

Caso de Estudio: Cholesky

- Definición del problema: Dada una matriz $A \in \mathcal{R}^{n \times n}$ *simétrica* y *definida positiva*, el problema consiste en encontrar la descomposición triangular siguiente:

$$A = C \cdot C^T \ ,$$

  siendo $C \in \mathcal{R}^{n \times n}$ una matriz triangular inferior.
- Esta descomposición sirve para resolver sistemas de ecuaciones con A como matriz del sistema.

Cholesky: algoritmo escalar

function CHOL_ESCALAR(A) **return** C
  $C \leftarrow A$
  for $k = 1 \rightarrow n$ **do**
    $c = \sqrt{C(k,k)}$
    $C(k,k) = c$
    for $i = k+1 \rightarrow n$ **do**
      $C(i,k) = C(i,k)/c$
    end for
    for $i = k+1 \rightarrow n$ **do**
      for $j = k+1 \rightarrow i-1$ **do**
        $C(i,j) = C(i,j) - C(i,k) \cdot C(j,k)$
      end for
      $C(i,i) = C(i,i) - C(i,k) \cdot C(i,k)$
    end for
  end for
end function

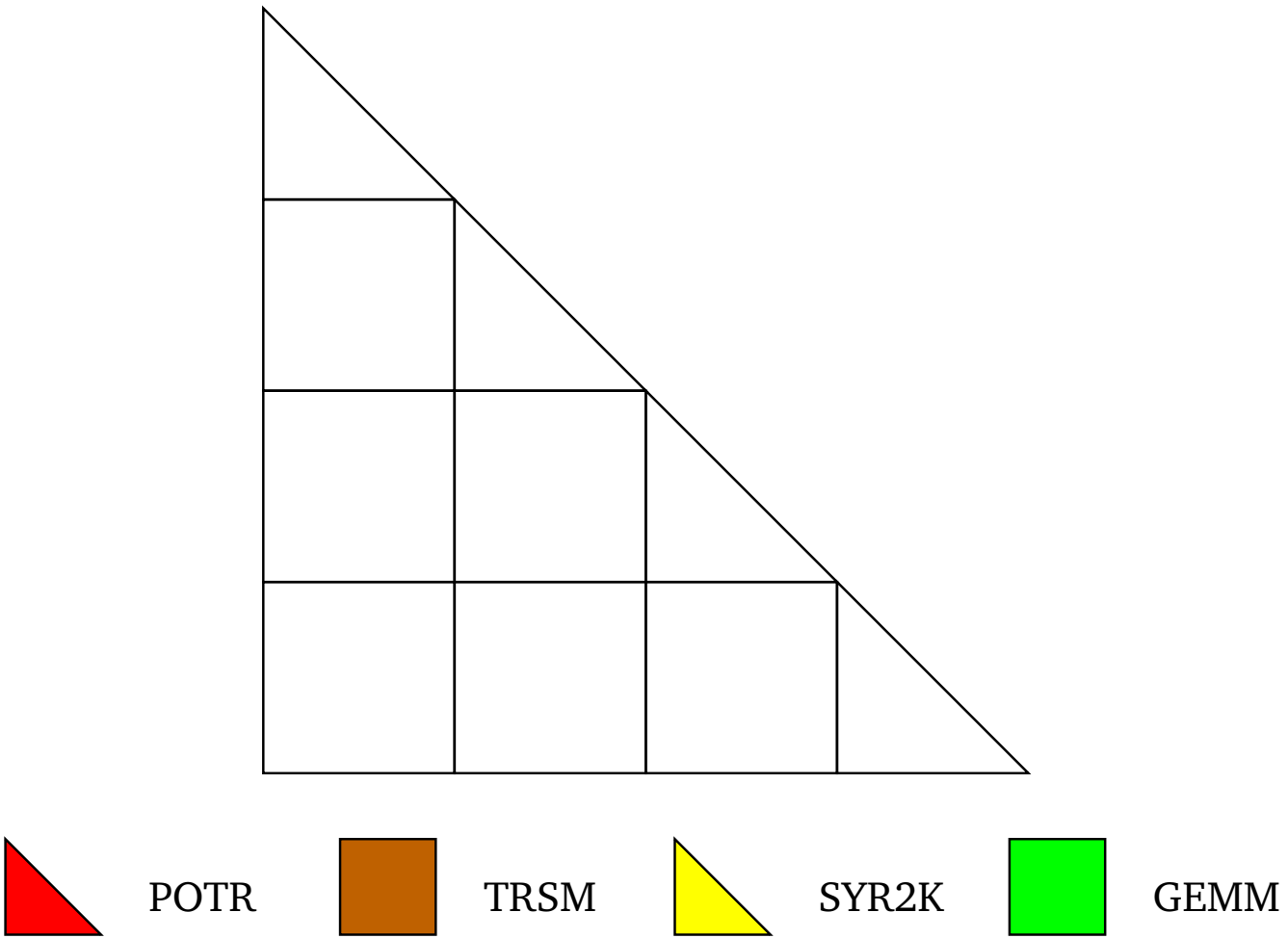
Notas sobre el algoritmo escalar de Cholesky

- Coste del algoritmo: $O(n^3/3)$.
- Los blucles internos son paralelizables.
- La versión escalar es ineficiente, en secuencial y en paralelo. Es necesario disponer de una versión eficiente por bloques.

Cholesky: algoritmo por bloques

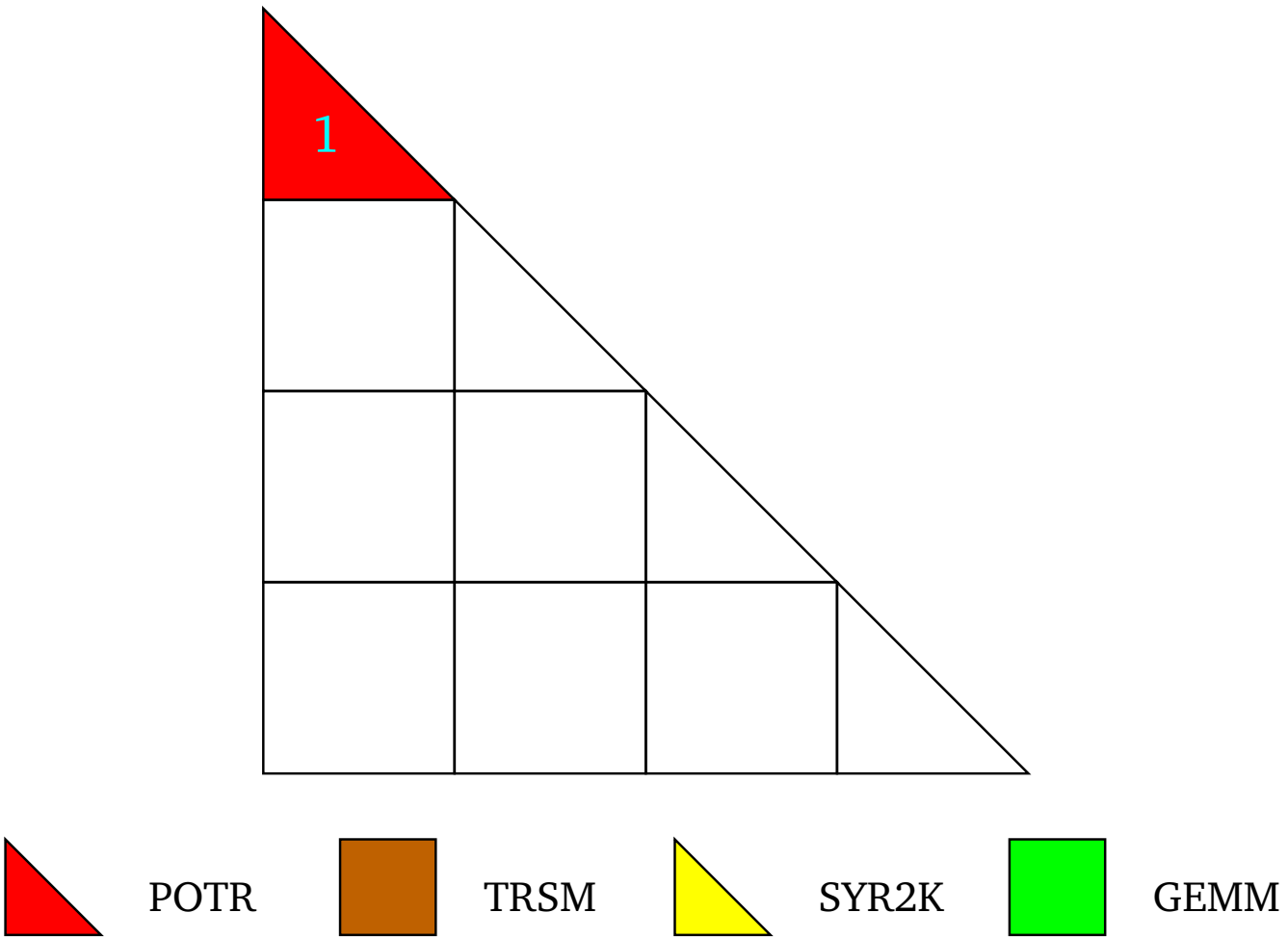
function CHOL_BLOQUES(A, b) **return** C
  $C \leftarrow A$
  for $k = 1 : b : n$ **do**
    $m_k = \min(k + b - 1, n)$
    $D = \text{CHOL\_ESCALAR}(C(k : m_k, k : m_k))$ ▷ POTRF
    $C(k : m_k, k : m_k) = D$
    FOR $i = k + b : b : n$ **DO**
      $m_i = \min(i + b - 1, n)$
      $C(i : m_i, k : m_k) \leftarrow C(i : m_i, k : m_k)/D^T$ ▷ TRSM
    END FOR
    FOR $i = k + b : b : n$ **DO**
      $m_i = \min(i + b - 1, n)$
      FOR $j = k + b : b : i - 1$ **DO**
        $m_j = \min(j + b - 1, n)$
        $C(i : m_i, j : m_j) \leftarrow C(i : m_i, j : m_j) - C(i : m_i, k : m_k) \cdot C(j : m_j, k : m_k)^T$ ▷ GEMM
      END FOR
      $C(i : m_i, i : m_i) \leftarrow C(i : m_i, i : m_i) - C(i : m_i, k : m_k) \cdot C(i : m_i, k : m_k)^T$ ▷ SYRK
    END FOR
  END FOR
END FUNCTION

Cholesky por bloques secuencial



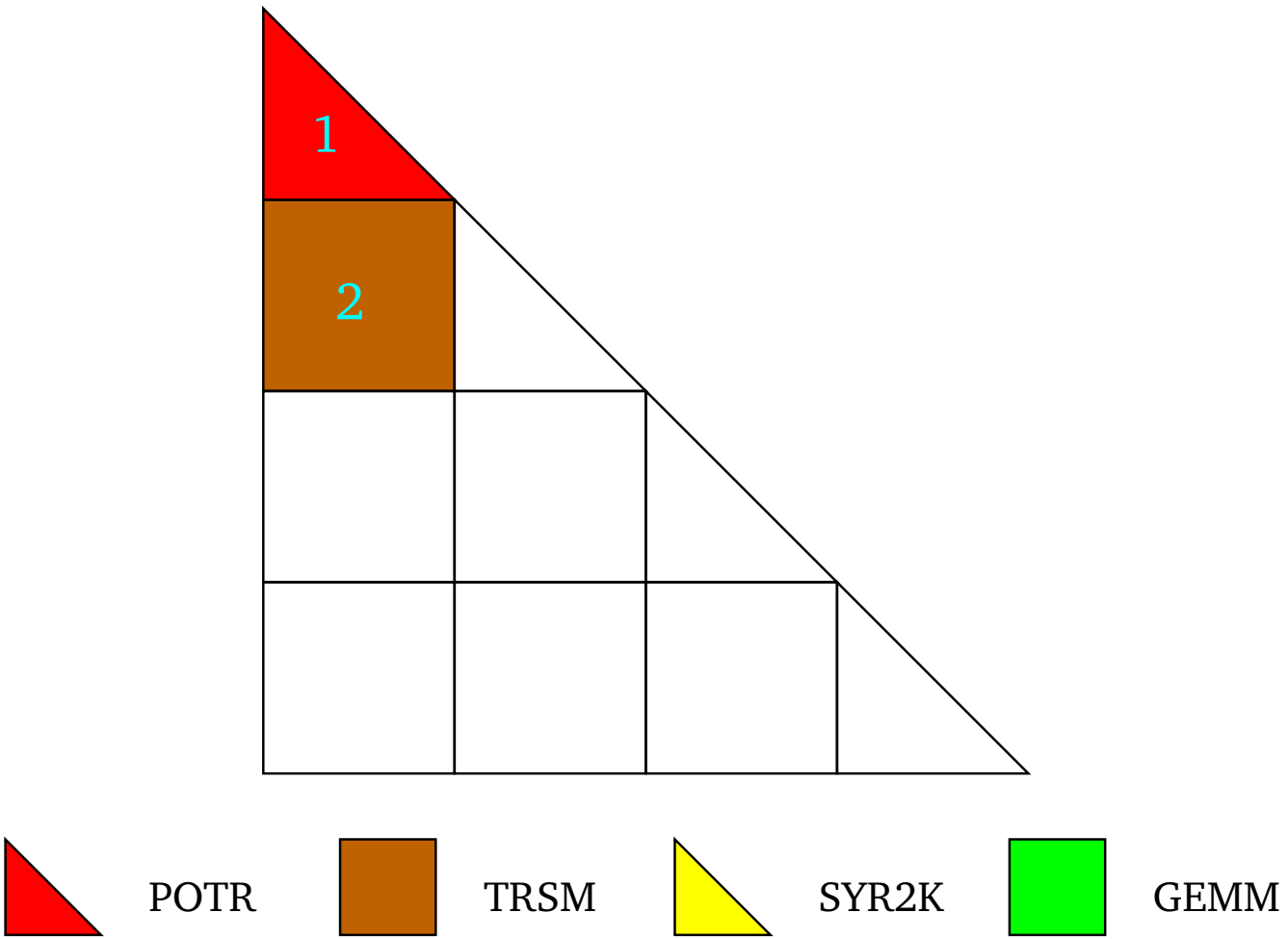
Cholesky por bloques secuencial

iteración `k=0`



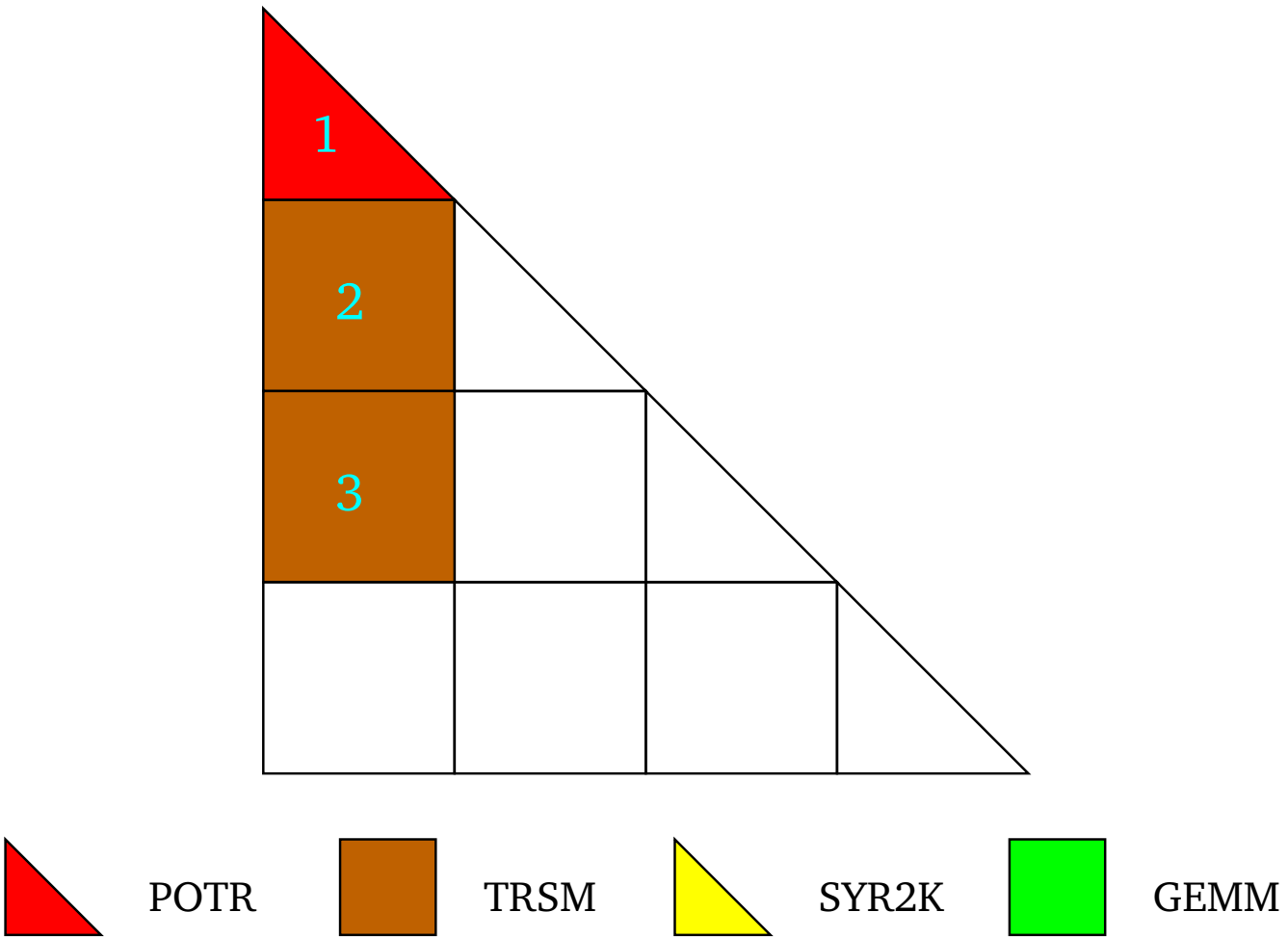
Cholesky por bloques secuencial

iteración `k=0`



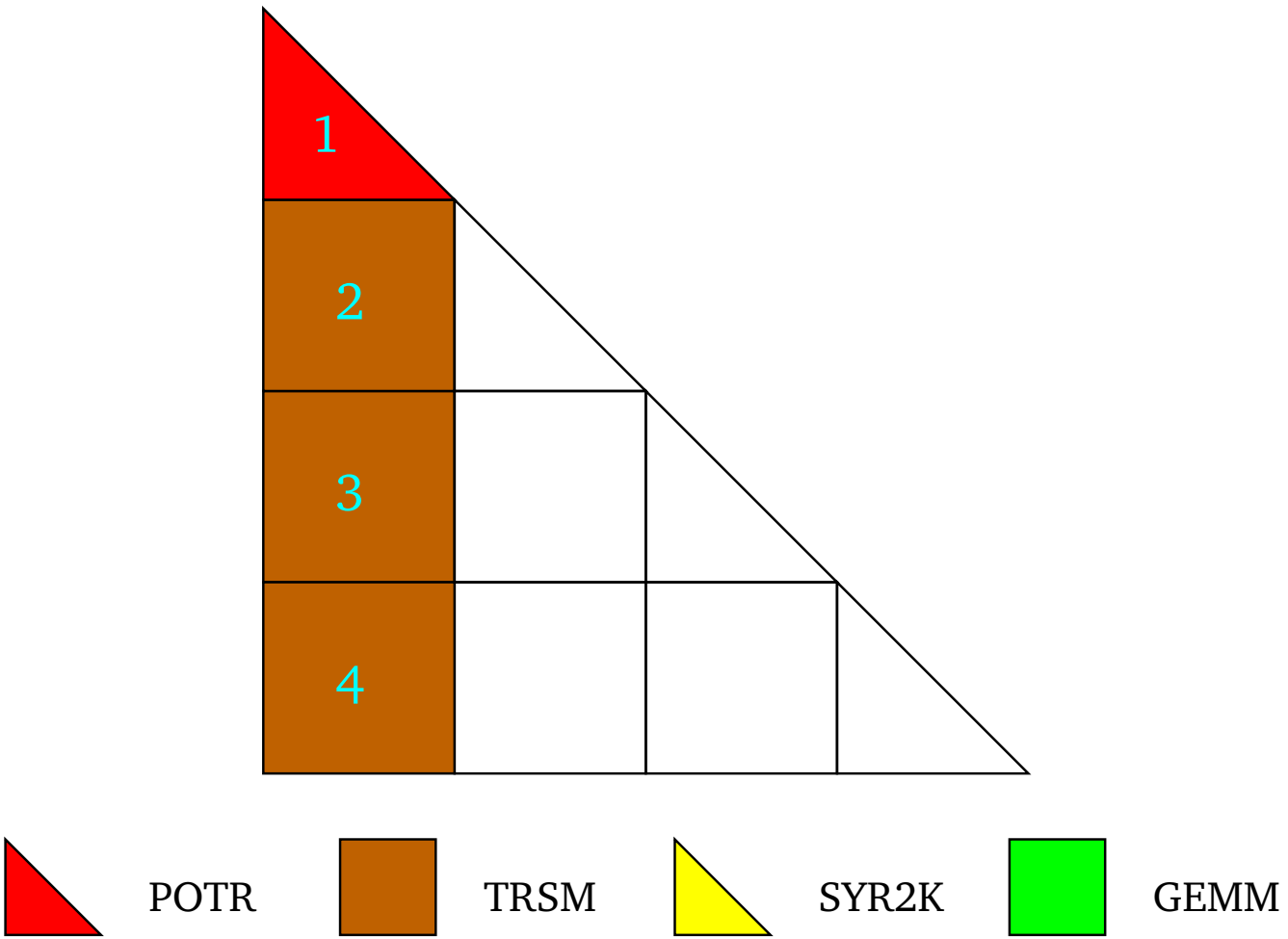
Cholesky por bloques secuencial

iteración `k=0`



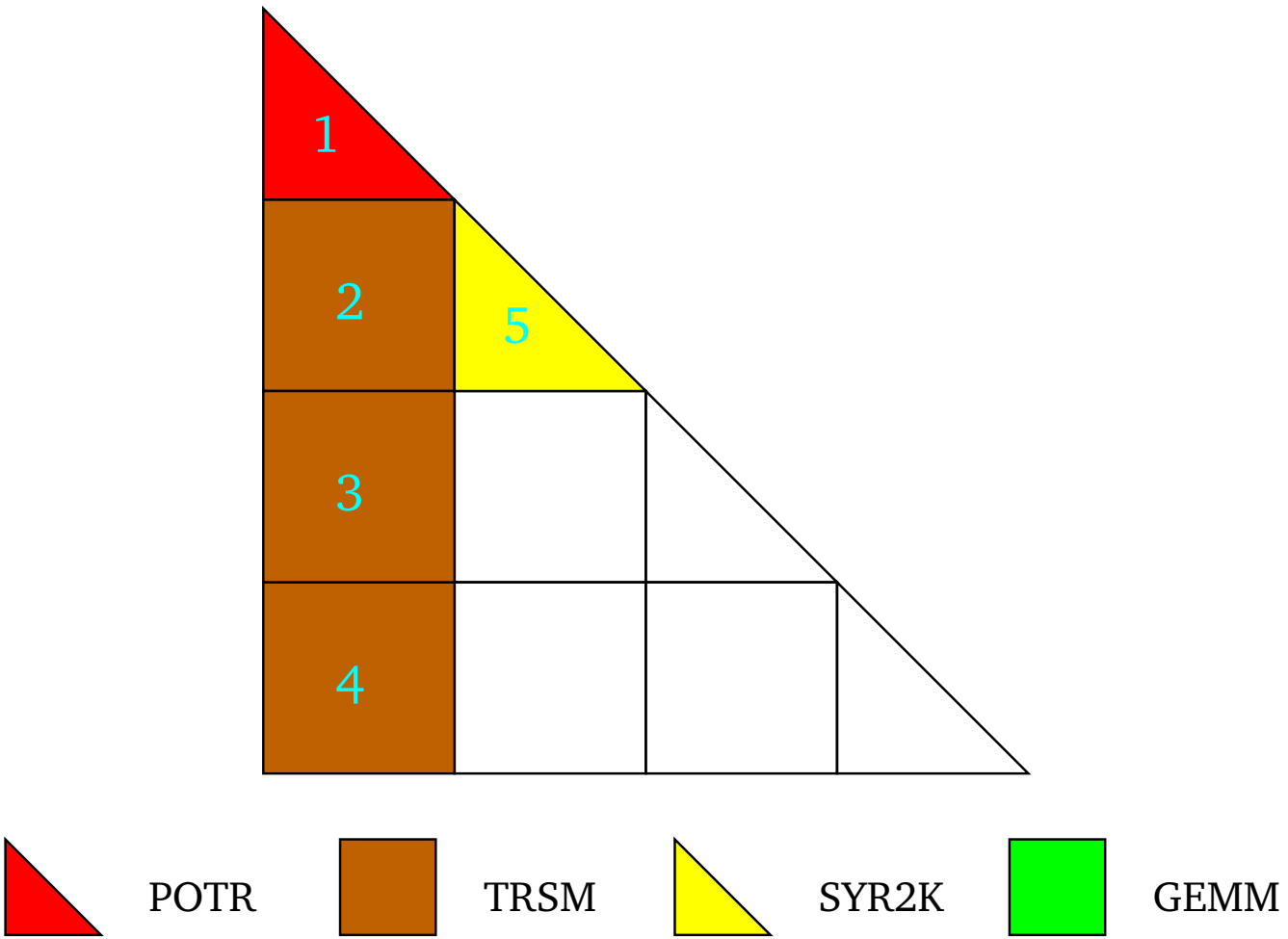
Cholesky por bloques secuencial

iteración `k=0`



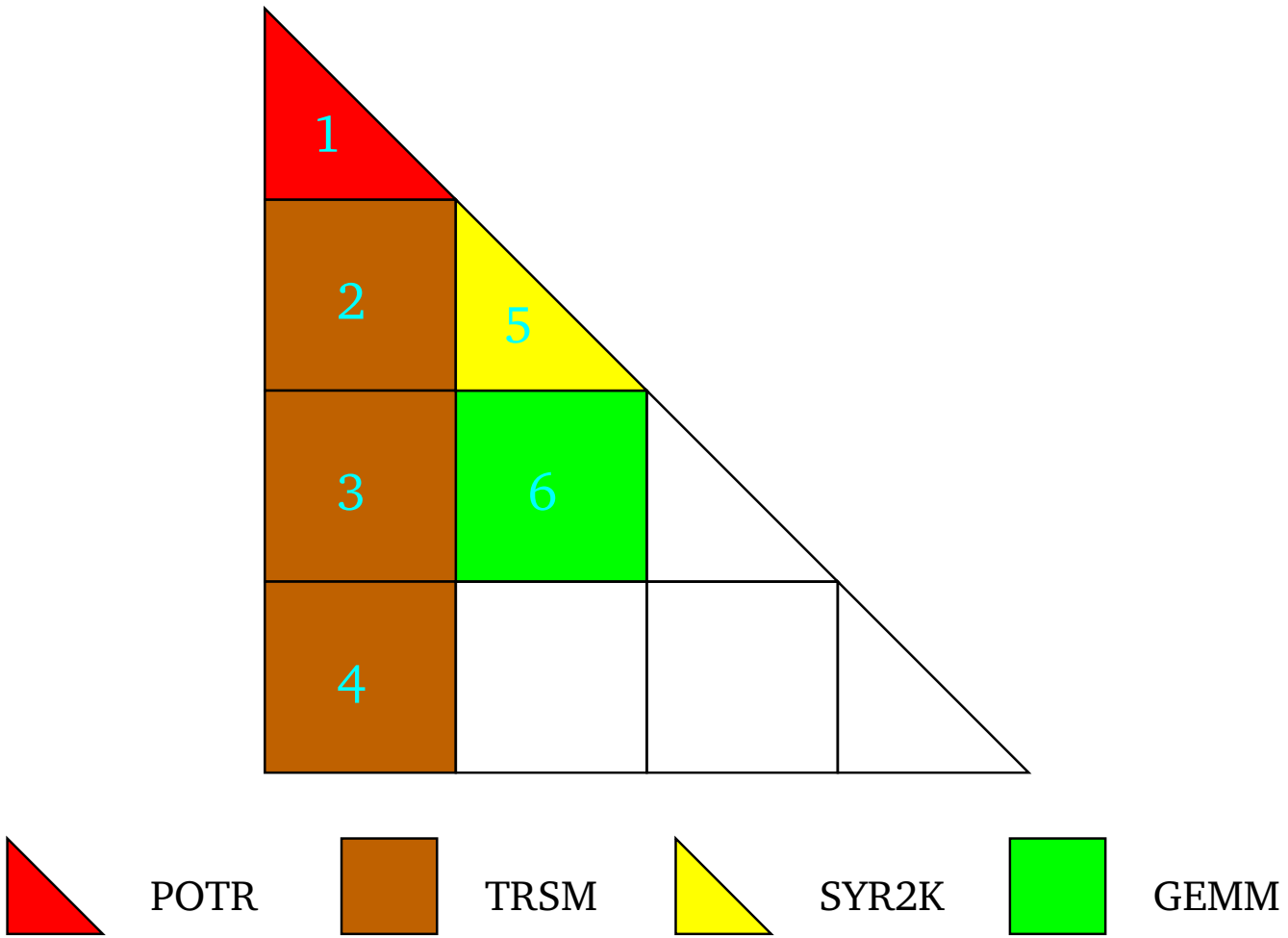
Cholesky por bloques secuencial

iteración `k=0`



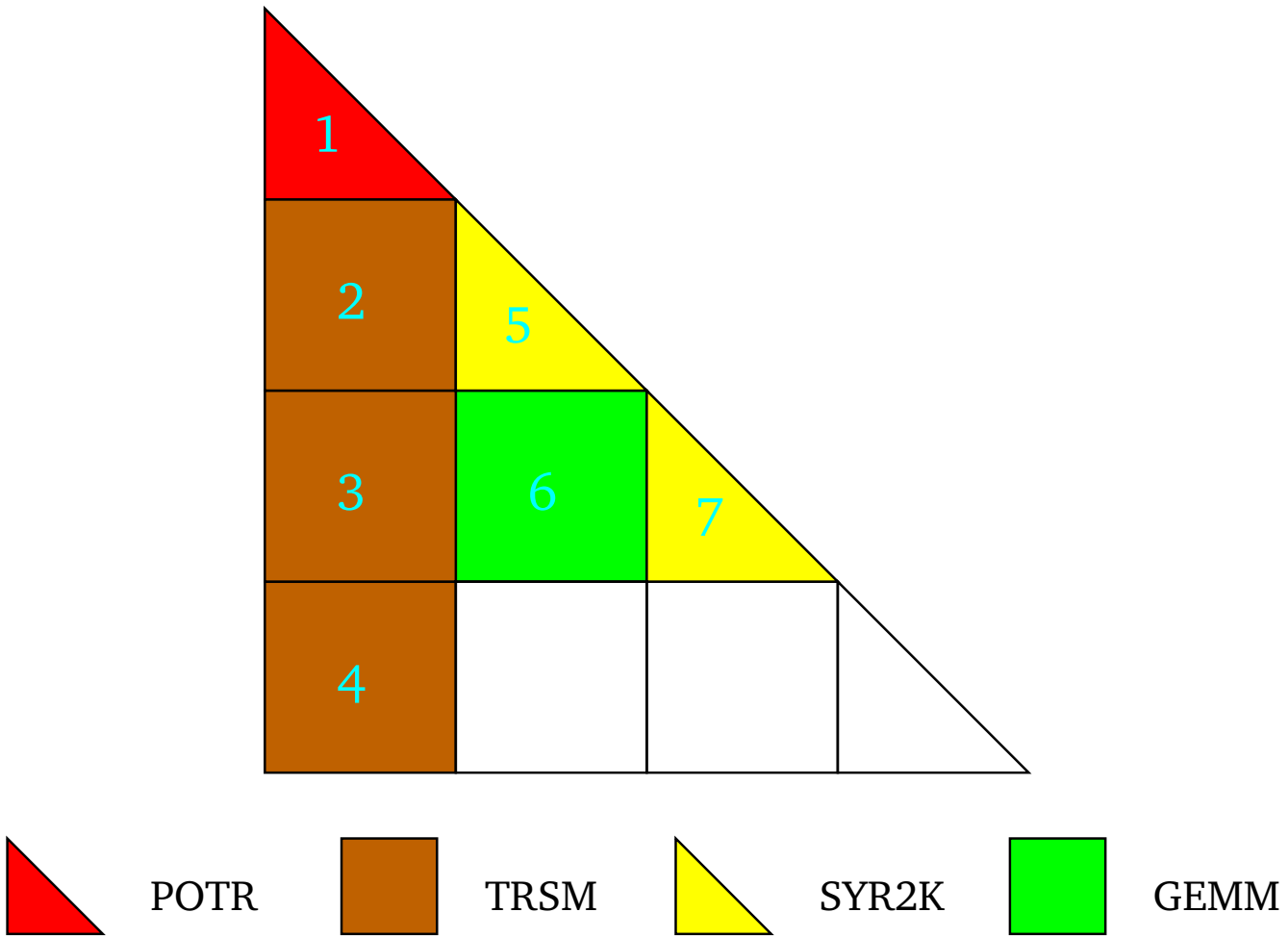
Cholesky por bloques secuencial

iteración `k=0`



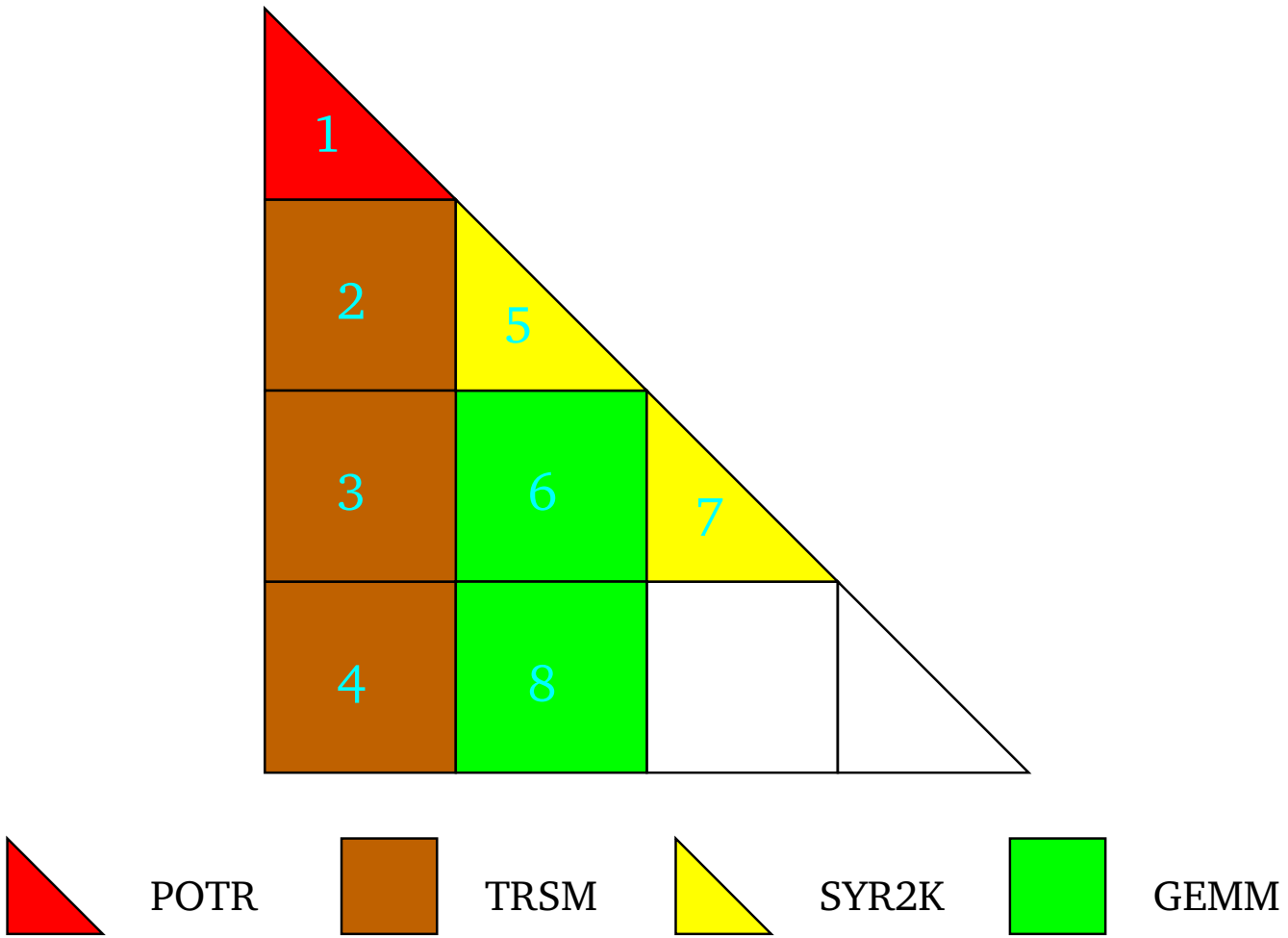
Cholesky por bloques secuencial

iteración `k=0`



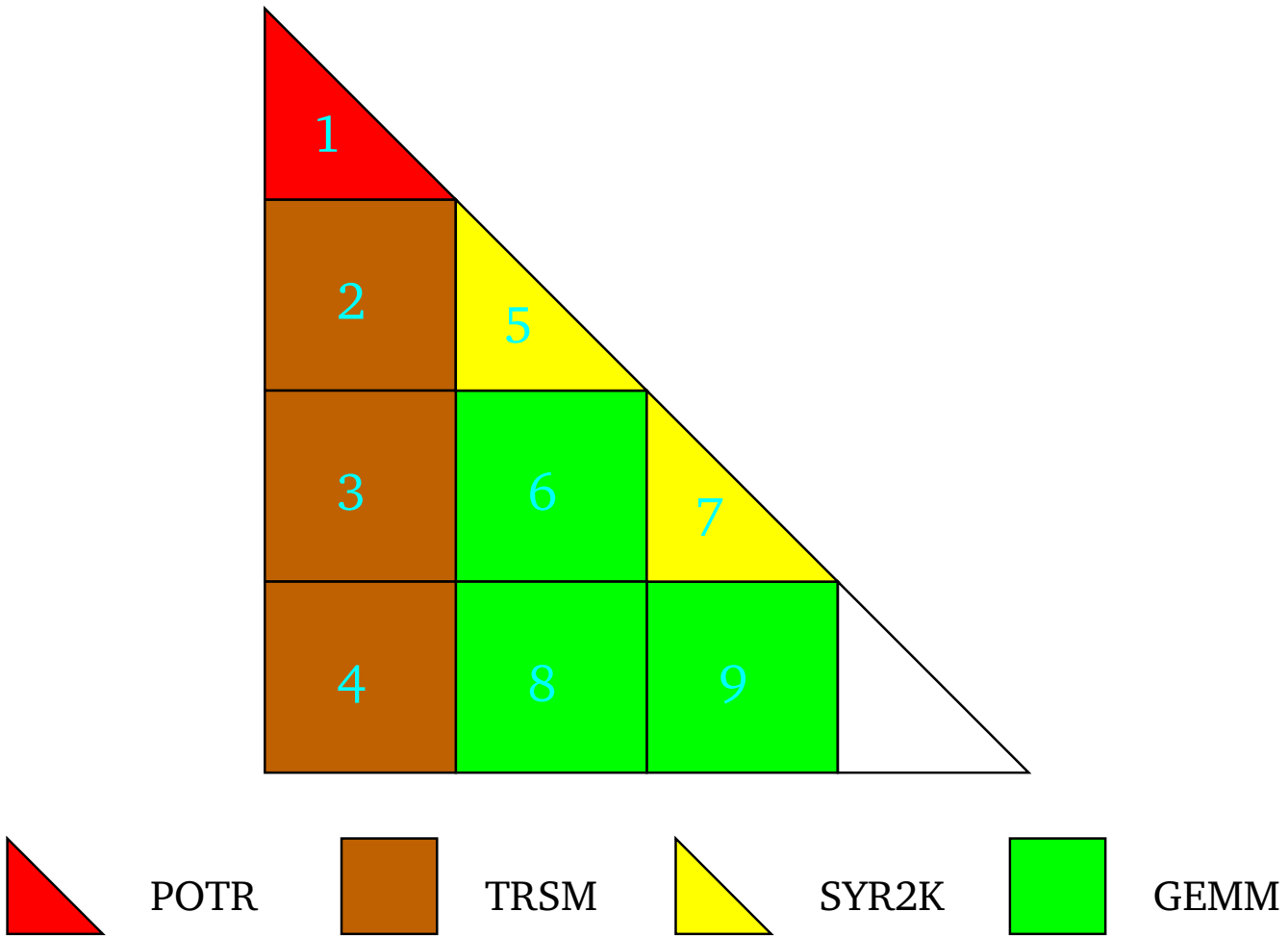
Cholesky por bloques secuencial

iteración `k=0`



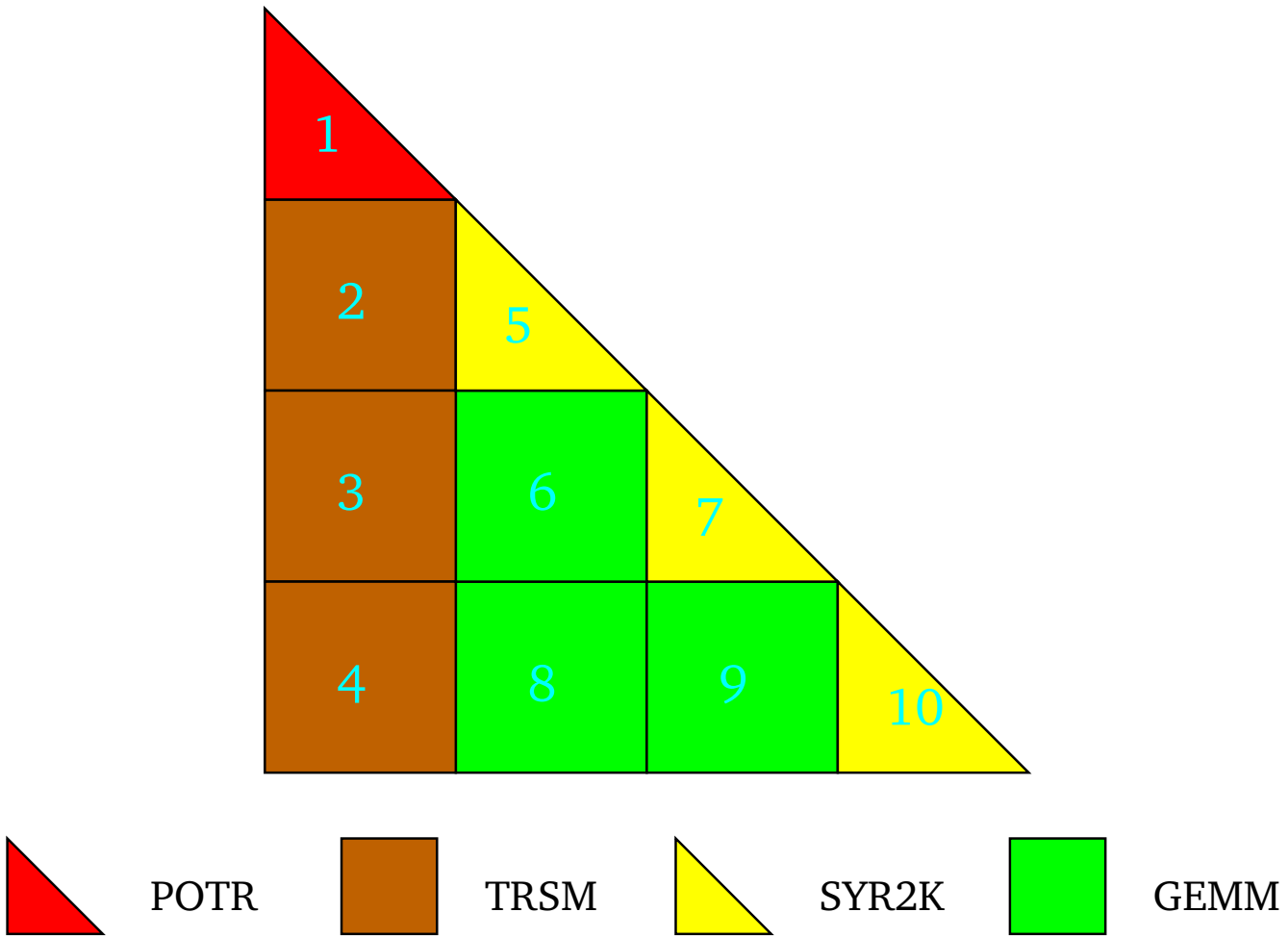
Cholesky por bloques secuencial

iteración `k=0`



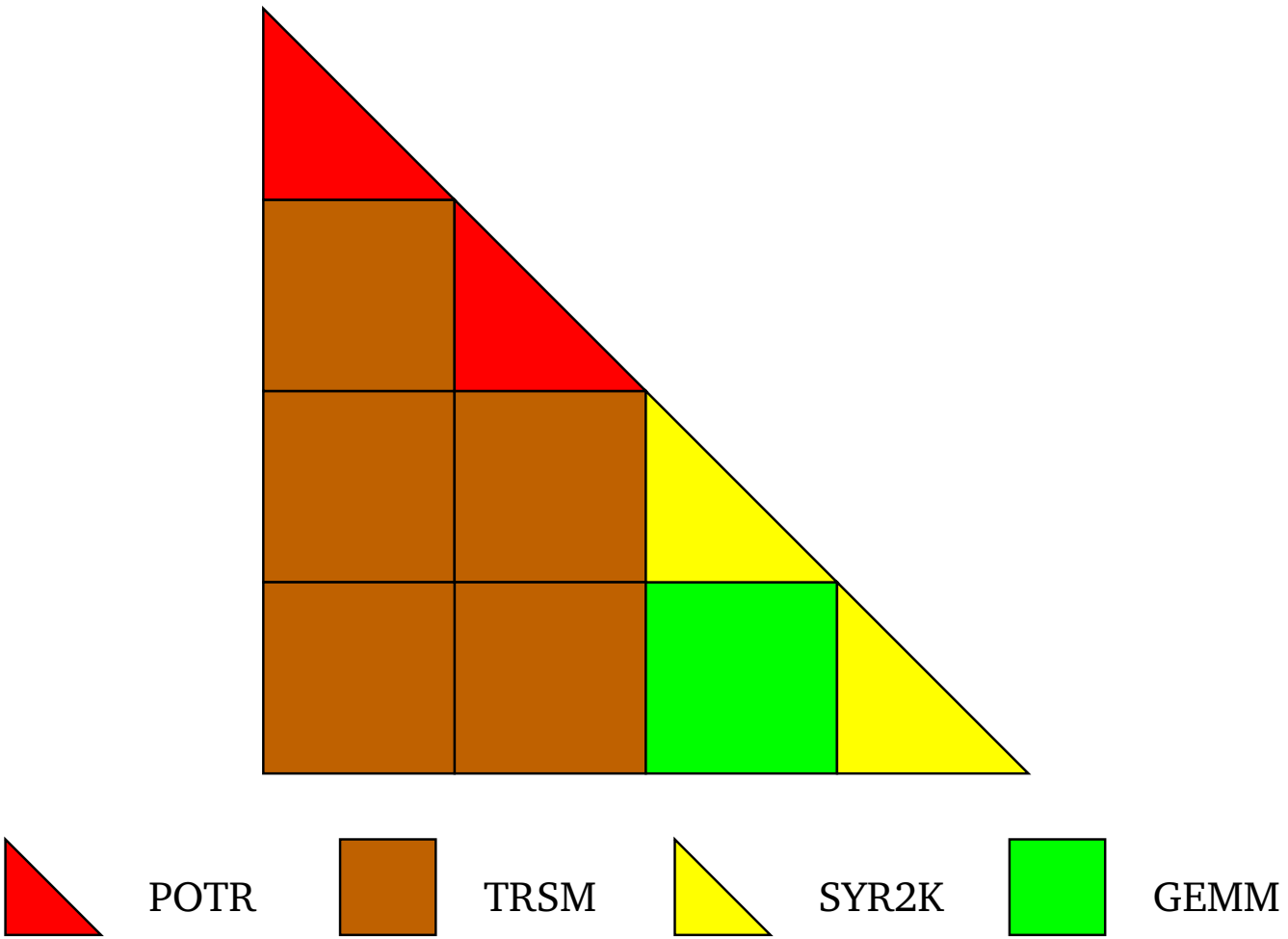
Cholesky por bloques secuencial

iteración `k=0`



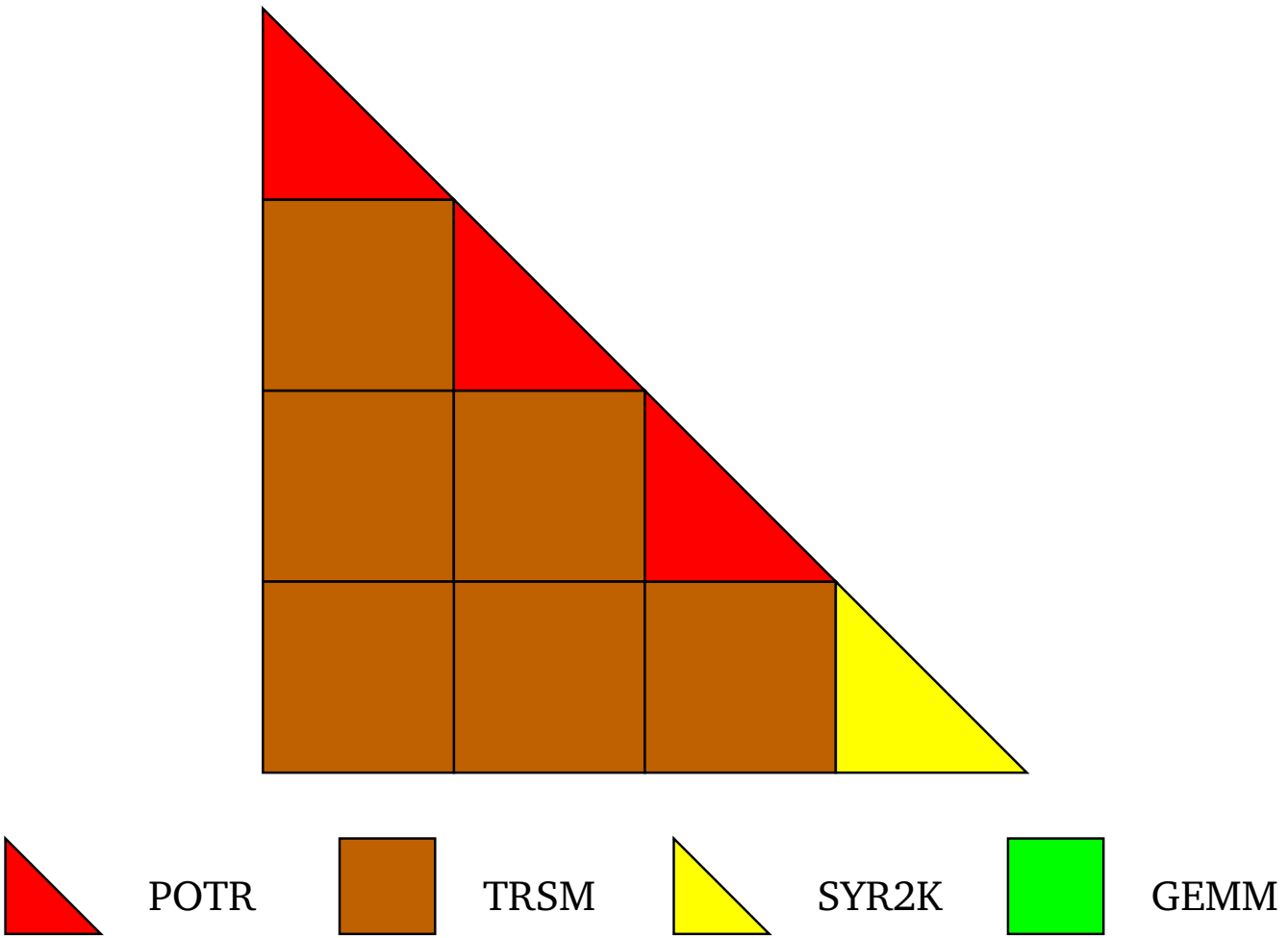
Cholesky por bloques secuencial

iteración `k=1`



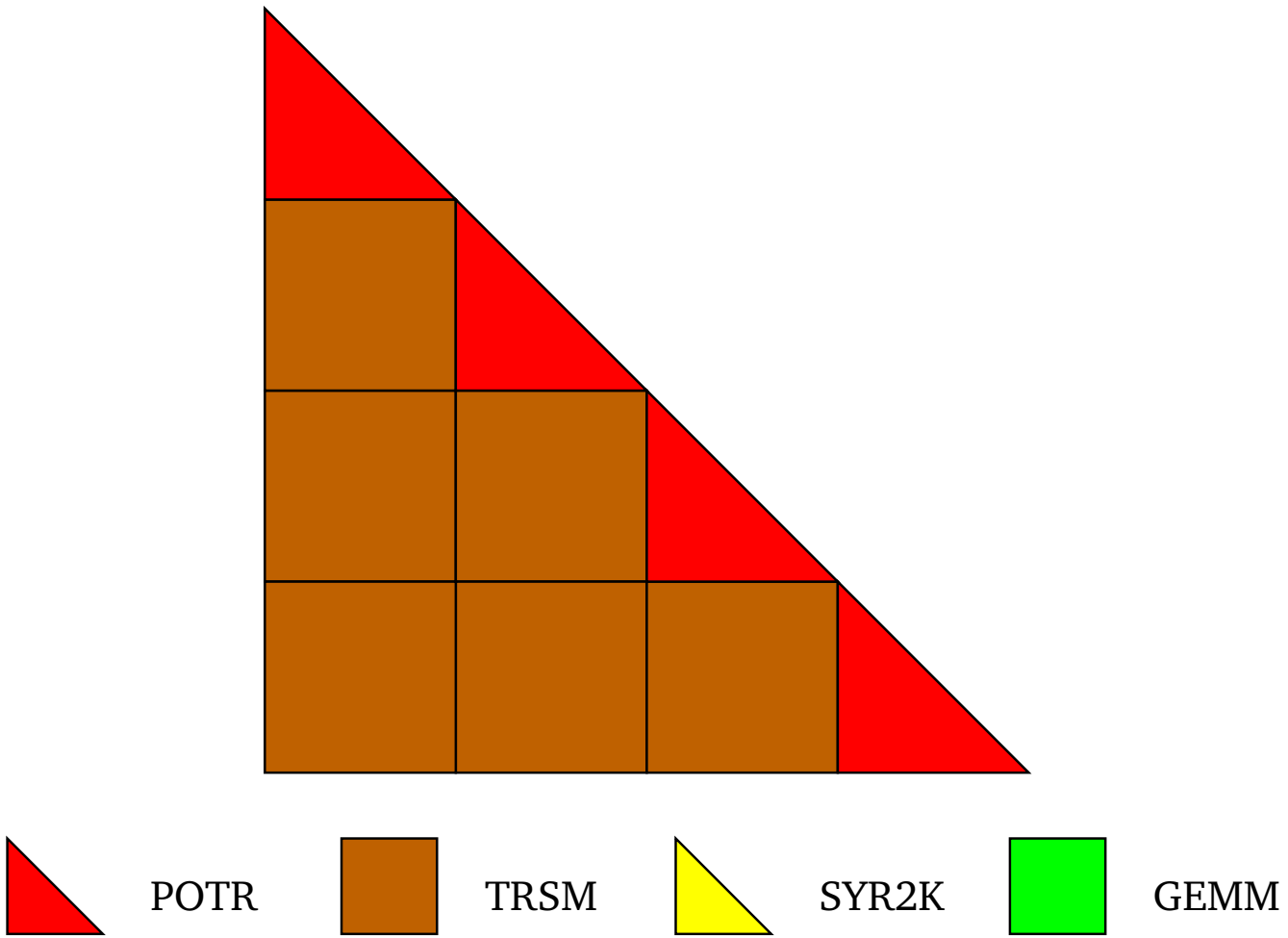
Cholesky por bloques secuencial

iteración `k`=2



Cholesky por bloques secuencial

iteración `k`=3

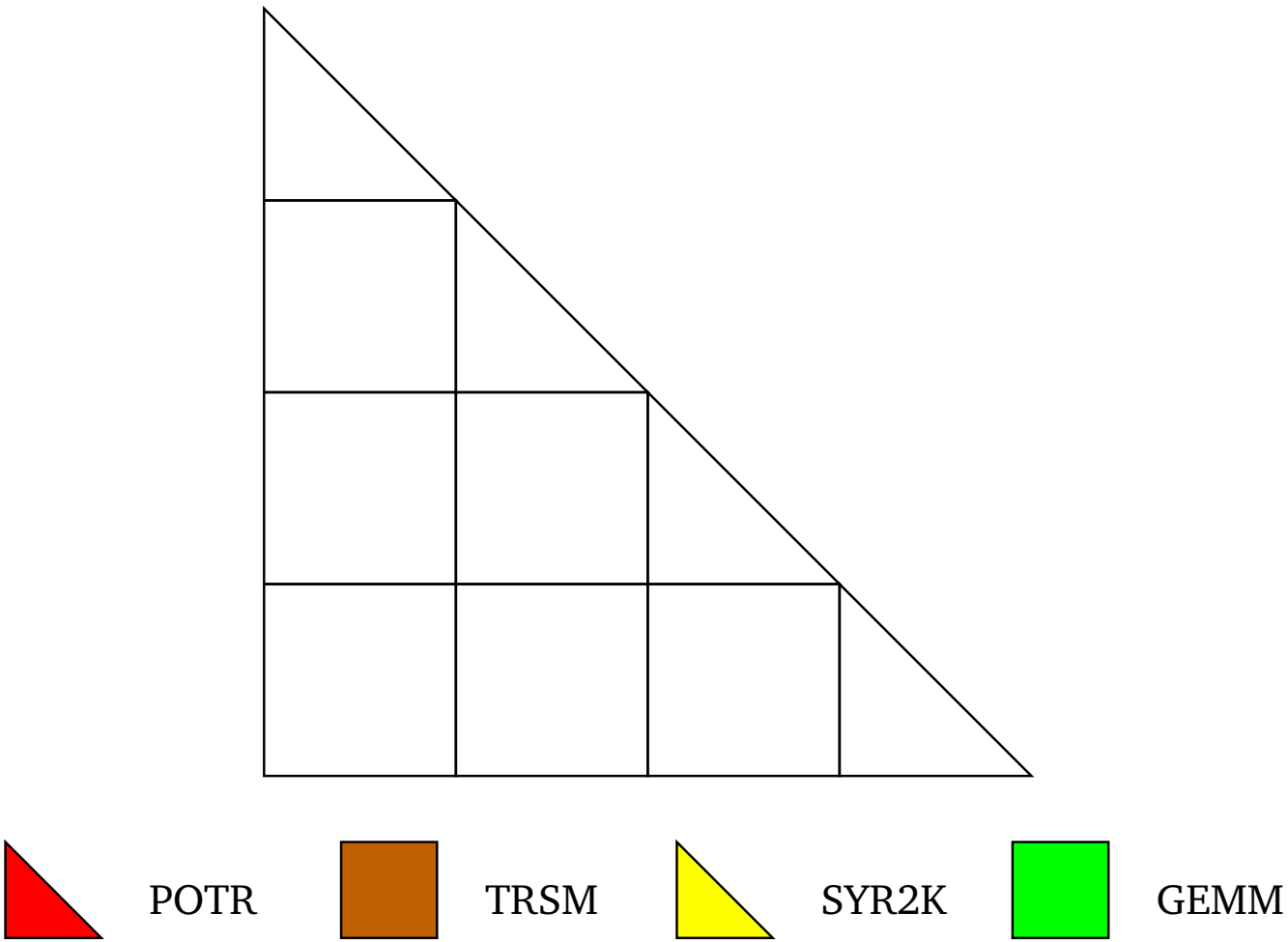


Cholesky: algoritmo por bloques

function CHOL_BLOQUES(A, b) **return** C
  $C \leftarrow A$
  for $k = 1 : b : n$ **do**
    $m_k = \min(k + b - 1, n)$
    $D = \text{CHOL\_ESCALAR}(C(k : m_k, k : m_k))$ ▷ POTRF
    $C(k : m_k, k : m_k) = D$
    FOR $i = k + b : b : n$ **DO**
      $m_i = \min(i + b - 1, n)$
      $C(i : m_i, k : m_k) \leftarrow C(i : m_i, k : m_k)/D^T$ ▷ TRSM
    END FOR
    FOR $i = k + b : b : n$ **DO**
      $m_i = \min(i + b - 1, n)$
      FOR $j = k + b : b : i - 1$ **DO**
        $m_j = \min(j + b - 1, n)$
        $C(i : m_i, j : m_j) \leftarrow C(i : m_i, j : m_j) - C(i : m_i, k : m_k) \cdot C(j : m_j, k : m_k)^T$ ▷ GEMM
      END FOR
      $C(i : m_i, i : m_i) \leftarrow C(i : m_i, i : m_i) - C(i : m_i, k : m_k) \cdot C(i : m_i, k : m_k)^T$ ▷ SYRK
    END FOR
  END FOR
END FUNCTION

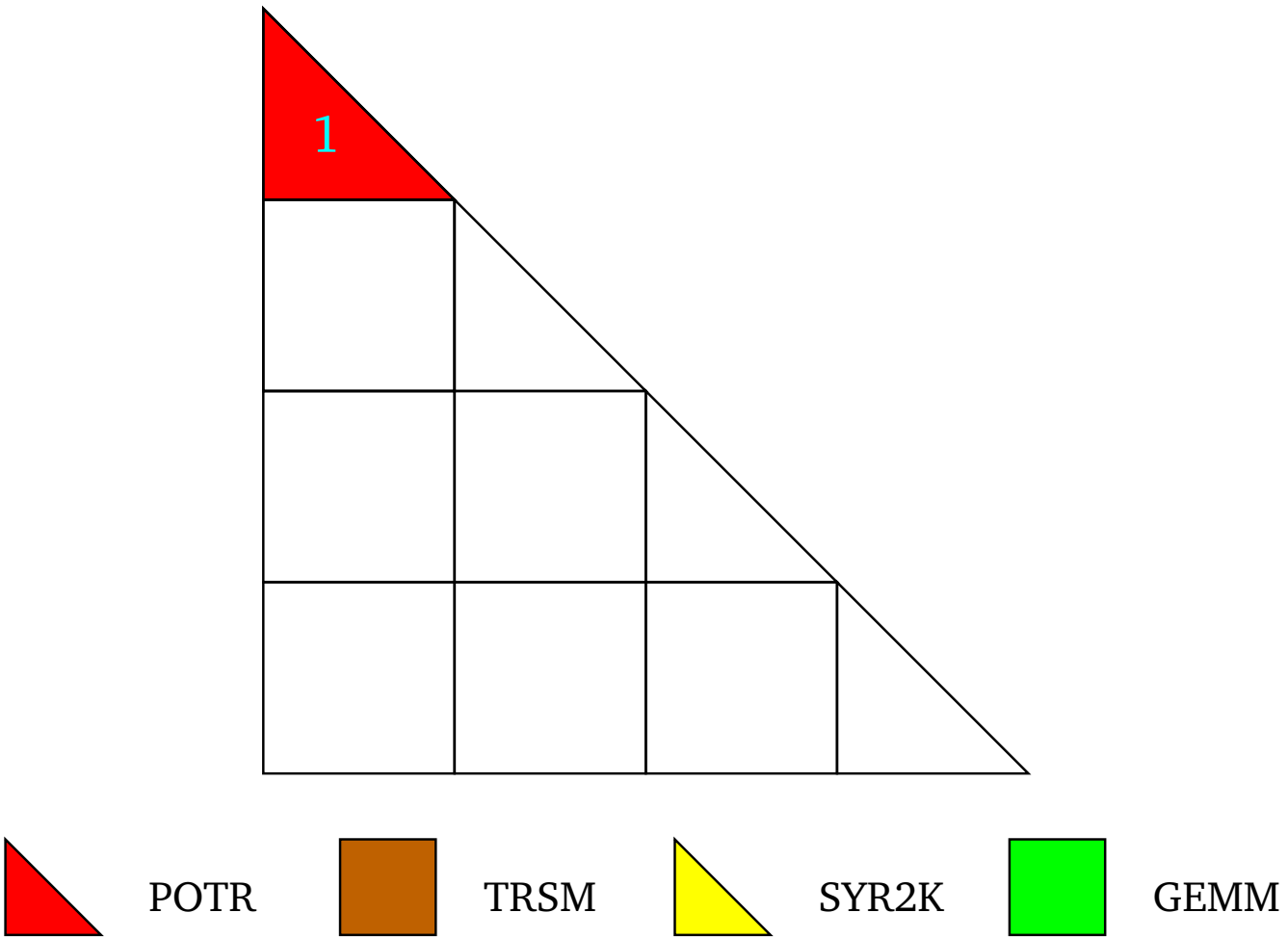
Cholesky por bloques paralelo

iteración k=0

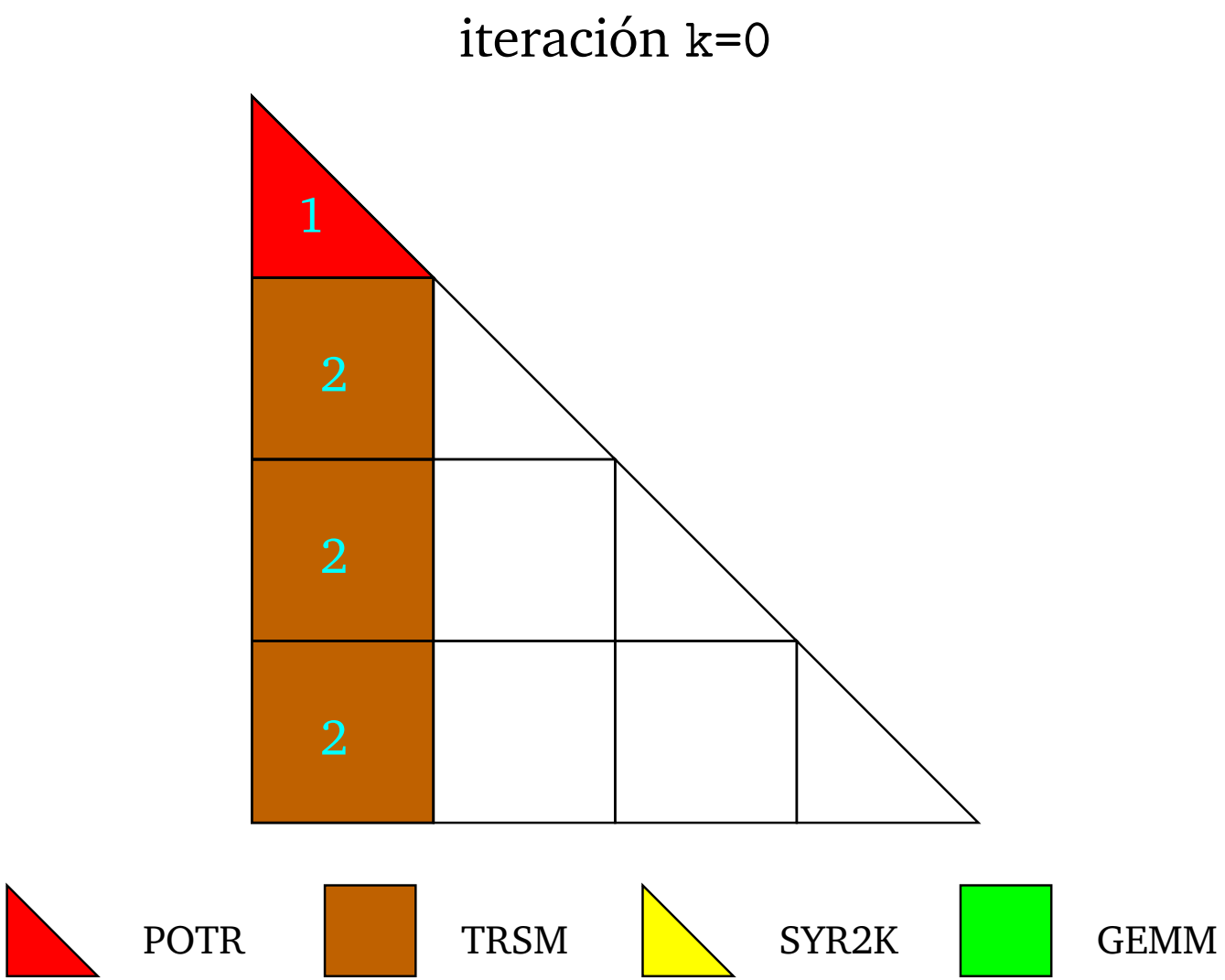


Cholesky por bloques paralelo

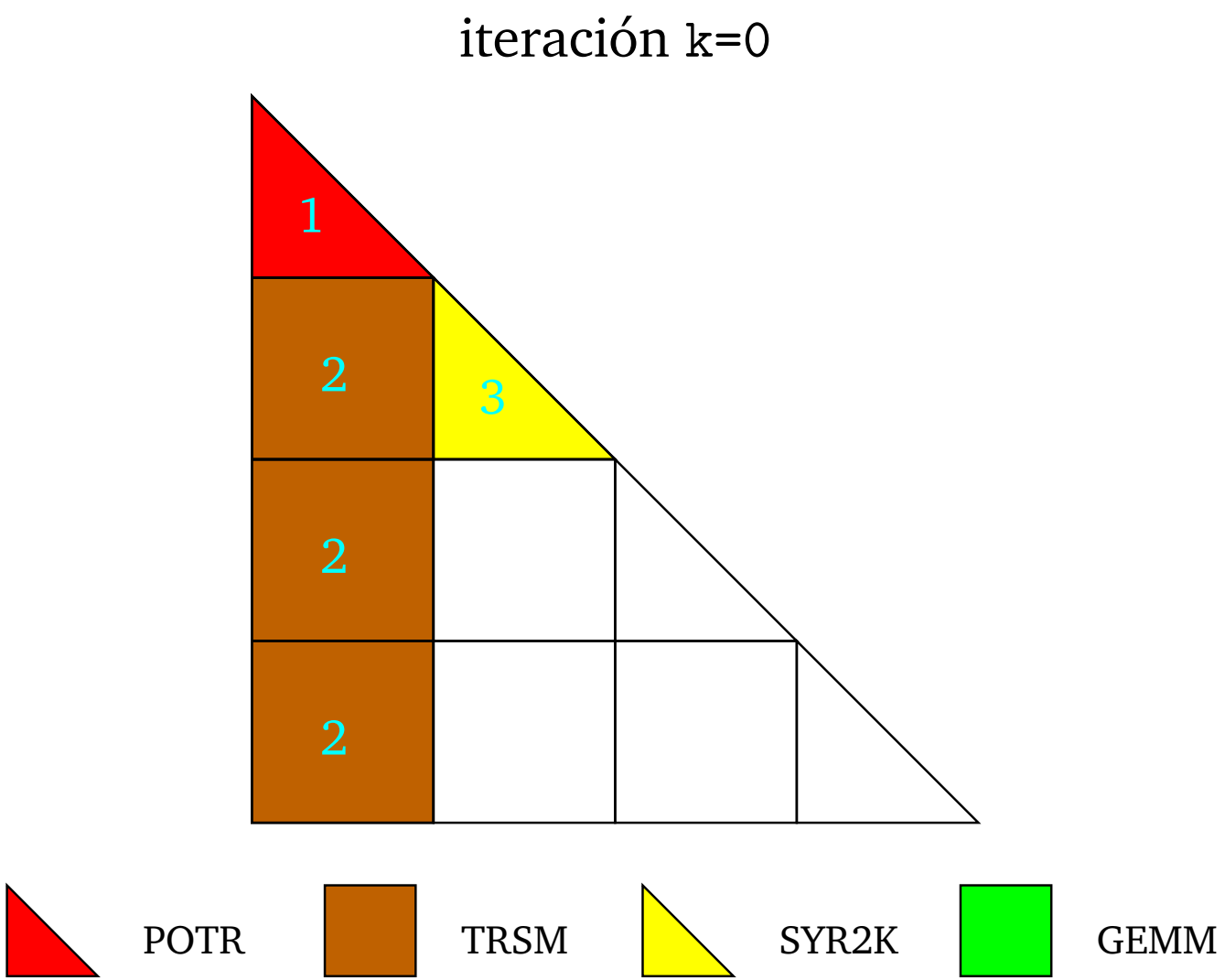
iteración `k=0`



Cholesky por bloques paralelo

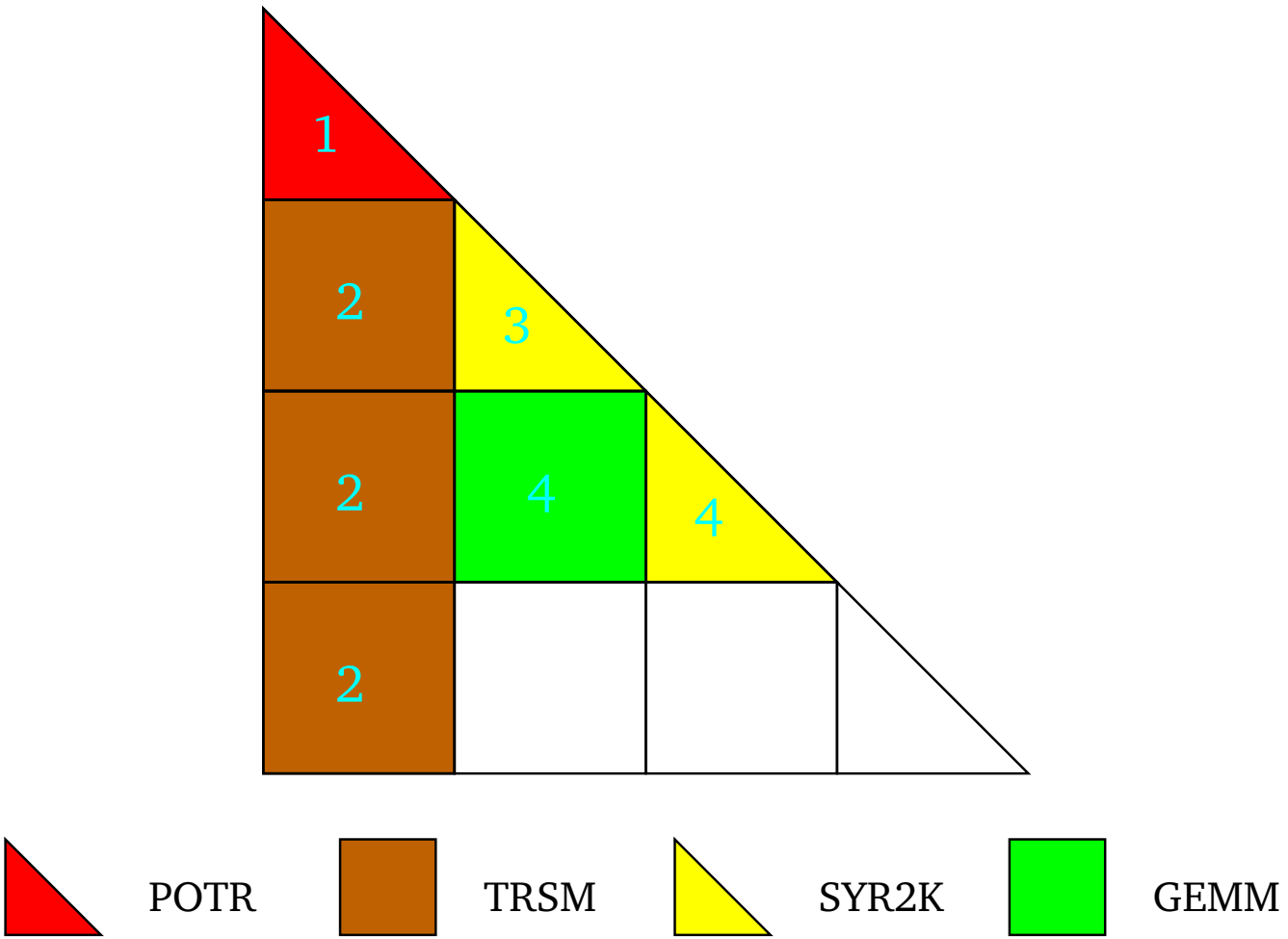


Cholesky por bloques paralelo



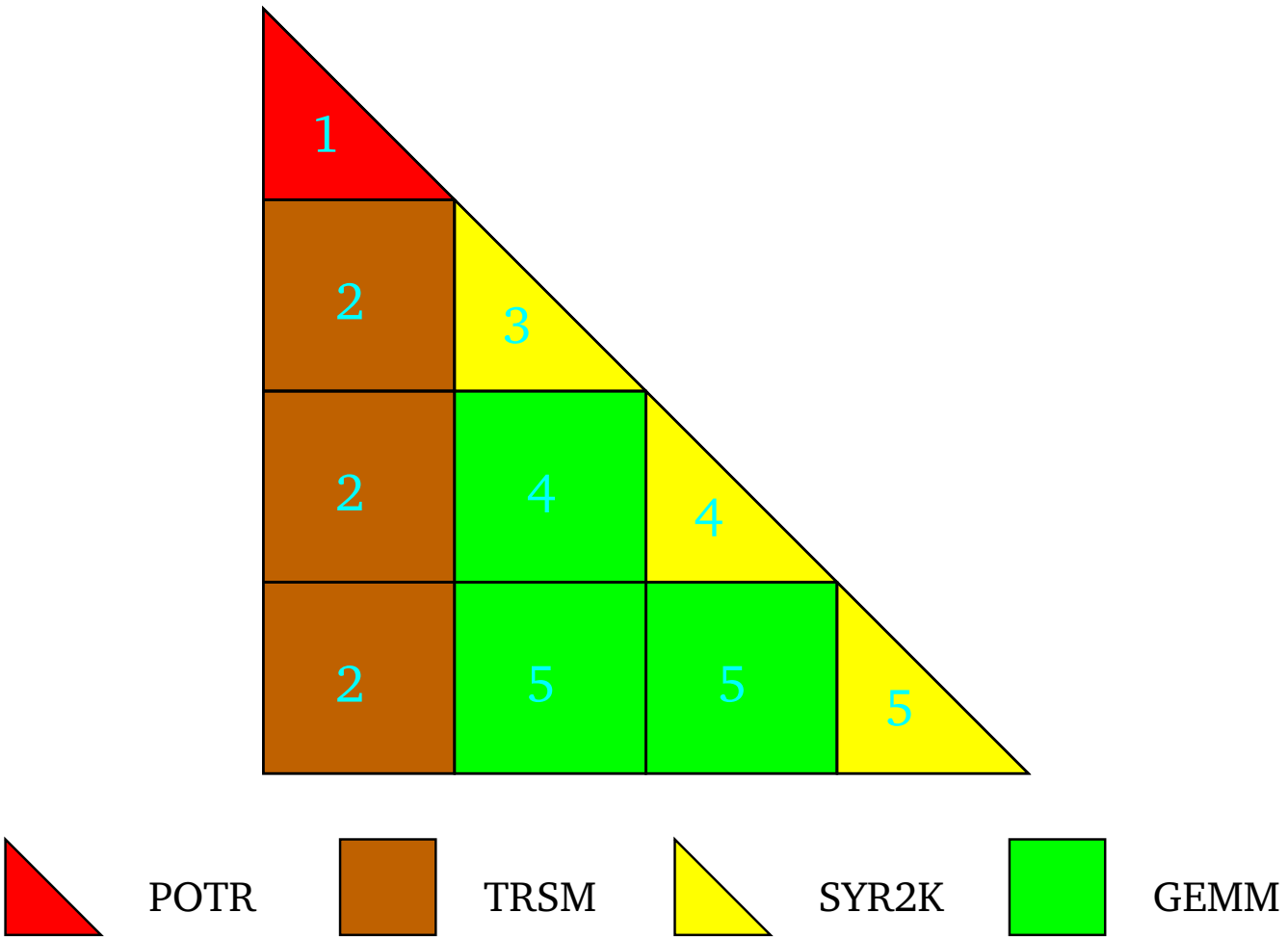
Cholesky por bloques paralelo

iteración `k=0`



Cholesky por bloques paralelo

iteración `k=0`



Cholesky: algoritmo por bloques paralelo

function CHOL_BLOQUES(A, b) **return** C
  $C \leftarrow A$
  for $k = 1 : b : n$ **do**
    $m_k = \min(k + b - 1, n)$
    $D = \text{CHOL\_ESCALAR}(C(k : m_k, k : m_k))$ ▷ POTRF
    $C(k : m_k, k : m_k) = D$
    #pragma omp parallel for
    FOR $i = k + b : b : n$ **DO**
      $m_i = \min(i + b - 1, n)$
      $C(i : m_i, k : m_k) \leftarrow C(i : m_i, k : m_k)/D^T$ ▷ TRSM
    END FOR
    FOR $i = k + b : b : n$ **DO**
      $m_i = \min(i + b - 1, n)$
      #pragma omp parallel for
      FOR $j = k + b : b : i - 1$ **DO**
        $m_j = \min(j + b - 1, n)$
        $C(i : m_i, j : m_j) \leftarrow C(i : m_i, j : m_j) - C(i : m_i, k : m_k) \cdot C(j : m_j, k : m_k)^T$ ▷ GEMM
      END FOR
      $C(i : m_i, i : m_i) \leftarrow C(i : m_i, i : m_i) - C(i : m_i, k : m_k) \cdot C(i : m_i, k : m_k)^T$ ▷ SYRK
    END FOR
  END FOR
END FUNCTION

Cholesky: algoritmo por bloques paralelo

function CHOL_BLOQUES(A, b) **return** C
  $C \leftarrow A$
  for $k = 1 : b : n$ **do**
    $m_k = \min(k + b - 1, n)$
    $D = \text{CHOL\_ESCALAR}(C(k : m_k, k : m_k))$ ▷ POTRF
    $C(k : m_k, k : m_k) = D$
    #pragma omp parallel for
    FOR $i = k + b : b : n$ **DO**
      $m_i = \min(i + b - 1, n)$
      $C(i : m_i, k : m_k) \leftarrow C(i : m_i, k : m_k)/D^T$ ▷ TRSM
    END FOR
    #pragma omp parallel for
    FOR $i = k + b : b : n$ **DO**
      $m_i = \min(i + b - 1, n)$
      FOR $j = k + b : b : i - 1$ **DO**
        $m_j = \min(j + b - 1, n)$
        $C(i : m_i, j : m_j) \leftarrow C(i : m_i, j : m_j) - C(i : m_i, k : m_k) \cdot C(j : m_j, k : m_k)^T$ ▷ GEMM
      END FOR
      $C(i : m_i, i : m_i) \leftarrow C(i : m_i, i : m_i) - C(i : m_i, k : m_k) \cdot C(i : m_i, k : m_k)^T$ ▷ SYRK
    END FOR
  END FOR
END FUNCTION

Cholesky: algoritmo por bloques paralelo

function CHOL_BLOQUES(A, b) **return** C
  $C \leftarrow A$
  for $k = 1 : b : n$ **do**
    $m_k = \min(k + b - 1, n)$
    $D = \text{CHOL\_ESCALAR}(C(k : m_k, k : m_k))$ ▷ POTRF
    $C(k : m_k, k : m_k) = D$
    #pragma omp parallel for
    FOR $i = k + b : b : n$ **DO**
      $m_i = \min(i + b - 1, n)$
      $C(i : m_i, k : m_k) \leftarrow C(i : m_i, k : m_k)/D^T$ ▷ TRSM
    END FOR
    #pragma omp parallel for
    FOR $i = k + b : b : n$ **DO**
      $m_i = \min(i + b - 1, n)$
      #pragma omp parallel for
      FOR $j = k + b : b : i - 1$ **DO**
        $m_j = \min(j + b - 1, n)$
        $C(i : m_i, j : m_j) \leftarrow C(i : m_i, j : m_j) - C(i : m_i, k : m_k) \cdot C(j : m_j, k : m_k)^T$ ▷ GEMM
      END FOR
      $C(i : m_i, i : m_i) \leftarrow C(i : m_i, i : m_i) - C(i : m_i, k : m_k) \cdot C(i : m_i, k : m_k)^T$ ▷ SYRK
    END FOR
  END FOR
END FUNCTION

Caso de Estudio: MIMO

- Sistemas MIMO (Multiple Input Multiple Output) de análisis digital de señales.
- Problema matemático: encontrar $x \in \mathrm{I}^n$ tal que

$$\min_x ||Rx - b||_2 \ .$$

donde $R \in \mathcal{R}^{n \times n}$ es triangular superior, $b \in \mathcal{R}^n$ es un vector independiente de números reales.

- El conjunto discreto I se define, para un valor entero positivo y par t (cardinal del conjunto), como:

$$I = \left\{ \ \frac{1-t}{2}, \quad \frac{3-t}{2}, \quad \frac{5-t}{2}, \quad \cdots \quad \frac{t-3}{2}, \quad \frac{t-1}{2} \ \right\} \ .$$

Ejemplos:

$$I_{t=6} = \left\{ \ -\frac{5}{2}, \quad -\frac{3}{2}, \quad -\frac{1}{2}, \quad \frac{1}{2}, \quad \frac{3}{2}, \quad \frac{5}{2} \ \right\} \ ,$$

y

$$I_{t=8} = \left\{ \ -\frac{7}{2}, \quad -\frac{5}{2}, \quad -\frac{3}{2}, \quad -\frac{1}{2}, \quad \frac{1}{2}, \quad \frac{3}{2}, \quad \frac{5}{2}, \quad \frac{7}{2} \ \right\} \ .$$

Caso de Estudio: MIMO

- La 2-norma de un vector y de n elementos es
  $||y||_2 = \sqrt{y^T \cdot y} = \sqrt{\sum_{i=0}^{n-1} y_i^2}$,
  y se utiliza para obtener el valor de $||Rx - b||_2$.
- Ejemplo:

$$\min_x \left\| \begin{pmatrix} r_{00} & r_{01} & r_{02} & r_{03} \\ & r_{11} & r_{12} & r_{13} \\ & & r_{22} & r_{23} \\ & & & r_{33} \end{pmatrix} \begin{pmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{pmatrix} - \begin{pmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \end{pmatrix} \right\|_2 .$$

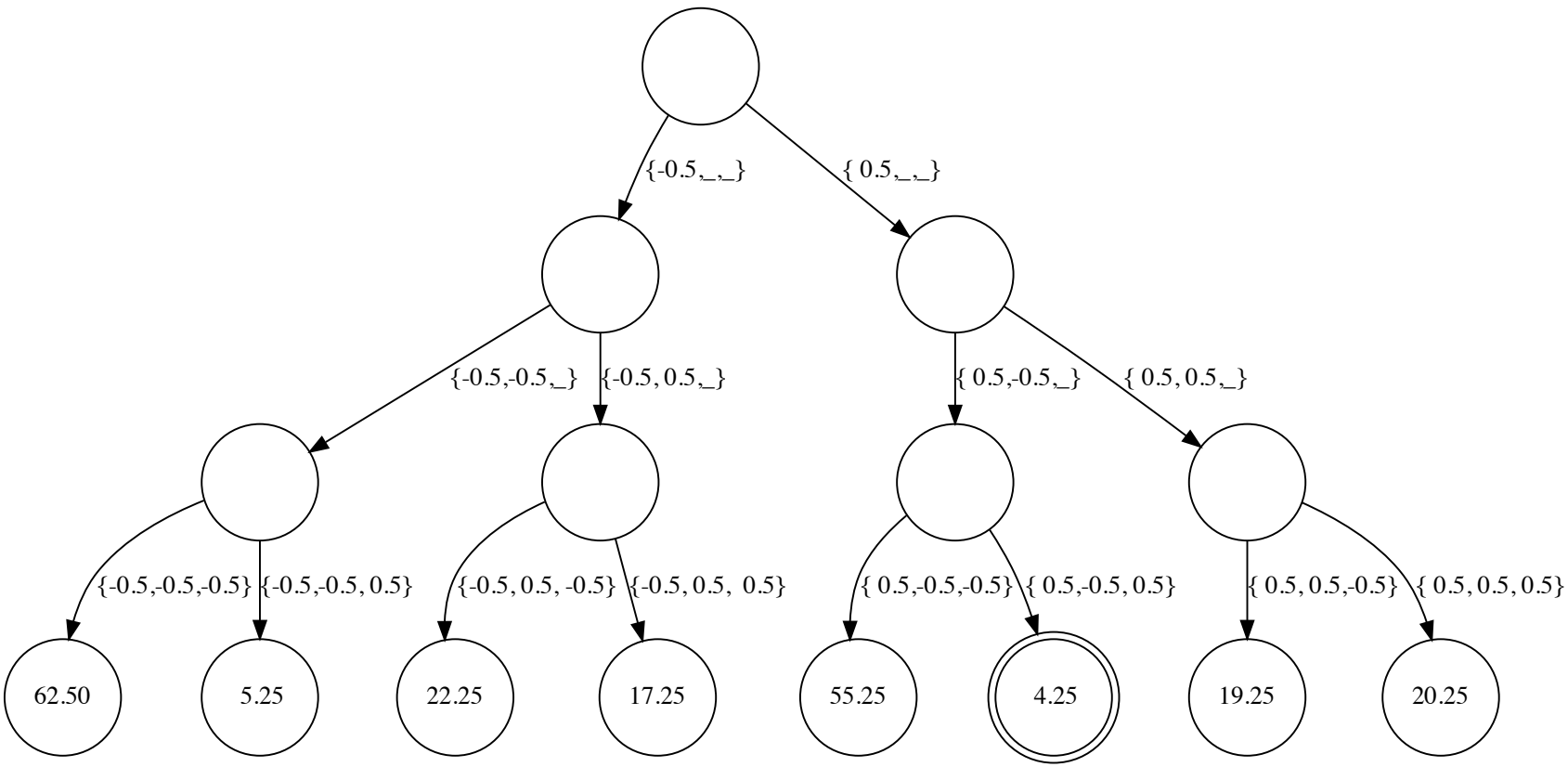
- Ejemplo numérico:

$$\min_x \left\| \begin{pmatrix} 1 & 2 & 3 \\ & 4 & 5 \\ & & 6 \end{pmatrix} \begin{pmatrix} x_0 \\ x_1 \\ x_2 \end{pmatrix} - \begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix} \right\|_2 , \quad I_{t=2} = \{-0{,}5, 0{,}5\}.$$

- Método de prueba y error por fuerza bruta: $O(t^n)$ flops.

Caso de Estudio: MIMO. Resolución por Backtracking

$$R = \begin{pmatrix} 1 & 2 & 3 \\ & 4 & 5 \\ & & 6 \end{pmatrix} \; b = \begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix} \; x = \begin{pmatrix} x_0 \\ x_1 \\ x_2 \end{pmatrix} \; I_{t=2} = \{-0{,}5, 0{,}5\}.$$



(Los números corresponden al cuadrado de la norma de x.)

Caso de Estudio: MIMO propiedad

Propiedad:

$$\min_x \left(\left\| \begin{pmatrix} r_{00} & r_{01} & r_{02} & r_{03} \\ & r_{11} & r_{12} & r_{13} \\ & & r_{22} & r_{23} \end{pmatrix} \begin{pmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{pmatrix} - \begin{pmatrix} b_0 \\ b_1 \\ b_2 \end{pmatrix} \right\|_2^2 + \|r_{33}x_3 - b_3\|_2^2 \right) =$$

$$= \min_x \left(\left\| \begin{pmatrix} r_{00} & r_{01} & r_{02} \\ & r_{11} & r_{12} \\ & & r_{22} \end{pmatrix} \begin{pmatrix} x_0 \\ x_1 \\ x_2 \end{pmatrix} - \begin{pmatrix} \bar{b}_0 \\ \bar{b}_1 \\ \bar{b}_2 \end{pmatrix} \right\|_2^2 + \|r_{33}x_3 - b_3\|_2^2 \right) ,$$

donde

$$\begin{pmatrix} \bar{b}_0 \\ \bar{b}_1 \\ \bar{b}_2 \end{pmatrix} = \begin{pmatrix} b_0 \\ b_1 \\ b_2 \end{pmatrix} - x_3 \begin{pmatrix} r_{03} \\ r_{13} \\ r_{23} \end{pmatrix} .$$

Caso de Estudio: MIMO propiedad

Propiedad:

$$\min_x \left\| \begin{pmatrix} r_{00} & r_{01} & r_{02} \\ & r_{11} & r_{12} \\ & & r_{22} \end{pmatrix} \begin{pmatrix} x_0 \\ x_1 \\ x_2 \end{pmatrix} - \begin{pmatrix} b_0 \\ b_1 \\ b_2 \end{pmatrix} \right\|_2^2 =$$

$$\min_x \left(\left\| \begin{pmatrix} r_{00} & r_{01} & r_{02} \\ & r_{11} & r_{12} \end{pmatrix} \begin{pmatrix} x_0 \\ x_1 \\ x_2 \end{pmatrix} - \begin{pmatrix} b_0 \\ b_1 \end{pmatrix} \right\|_2^2 + \|r_{22}x_2 - b_2\|_2^2 \right) =$$

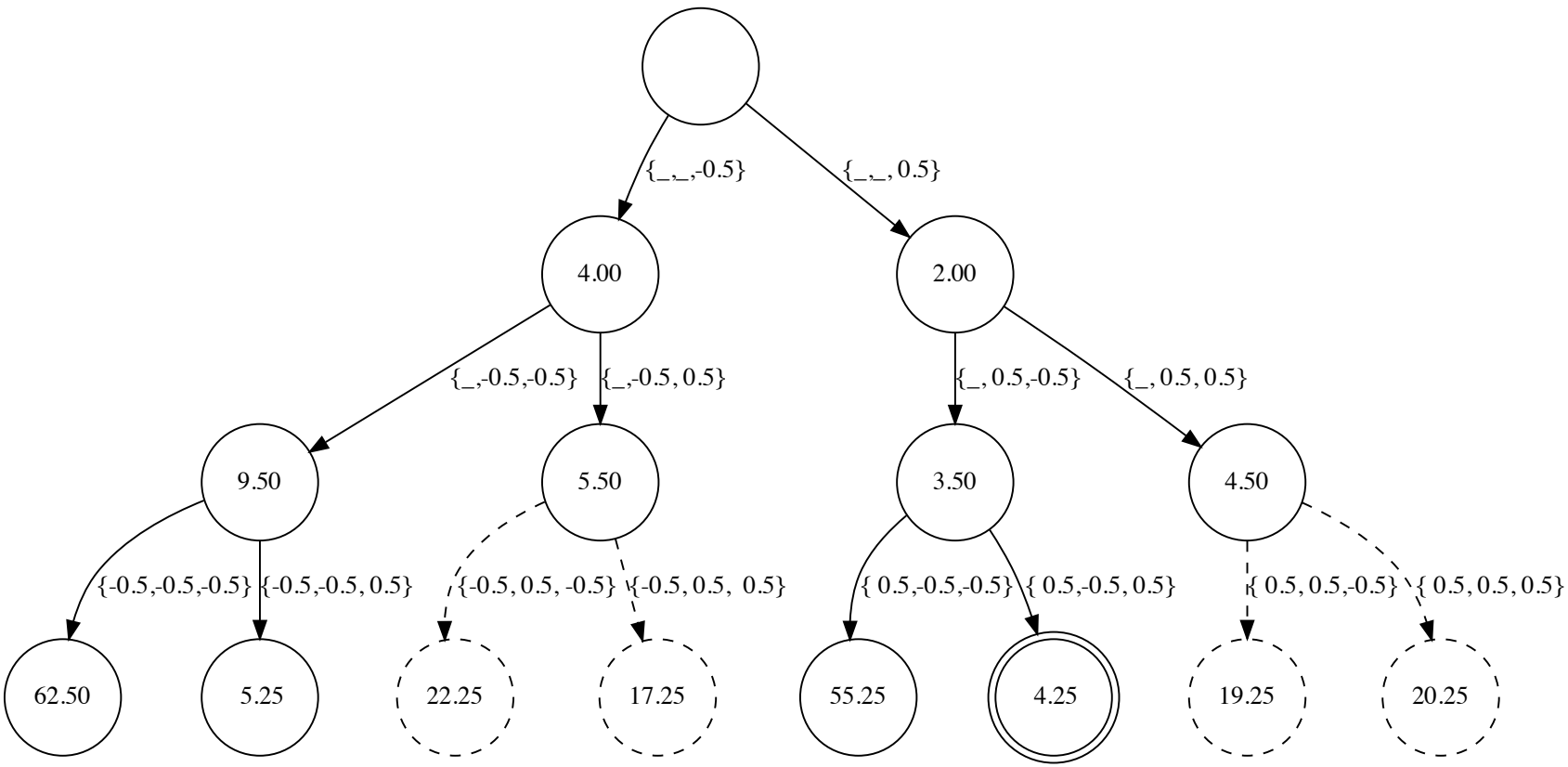
$$\min_x \left(\left\| \begin{pmatrix} r_{00} & r_{01} \\ & r_{11} \end{pmatrix} \begin{pmatrix} x_0 \\ x_1 \end{pmatrix} - \begin{pmatrix} \bar{b}_0 \\ \bar{b}_1 \end{pmatrix} \right\|_2^2 + \|r_{22}x_2 - b_2\|_2^2 \right) ,$$

donde

$$\begin{pmatrix} \bar{b}_0 \\ \bar{b}_1 \end{pmatrix} = \begin{pmatrix} b_0 \\ b_1 \end{pmatrix} - x_2 \begin{pmatrix} r_{02} \\ r_{12} \end{pmatrix} .$$

Caso de Estudio: MIMO. Ramificación y poda

$$R = \begin{pmatrix} 1 & 2 & 3 \\ & 4 & 5 \\ & & 6 \end{pmatrix} \quad b = \begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix} \quad x = \begin{pmatrix} x_0 \\ x_1 \\ x_2 \end{pmatrix} \quad I_{t=2} = \{-0{,}5, 0{,}5\}.$$



(Los números corresponden al cuadrado de la norma de x.)

Caso de Estudio: MIMO. Algoritmo secuencial

Posible solución: algoritmo de backtracking.

```
void mimo( int n, double *x, int t, double *I, double *R, int lda, double *b, double nrm )
{
  int k, inc = 1;
  if( n==0 ) {
    if( nrm < minimo ) {
      minimo = nrm;
      dcopy_( &lda, x, &inc, sol, &inc );
    }
  } else {
    for( k=0; k<t; k++ ) {
      int m = n-1;
      x(m) = I(k);
      double r = R(m,m)*x(m) - b(m);
      double norma = nrm + r*r;
      if( norma < minimo ) {
        double v[m], y[lda];
        dcopy_( &m, b, &inc, v, &inc );
        dcopy_( &lda, x, &inc, y, &inc );
        double alpha = -x(m);
        daxpy_( &m, &alpha, &R(0,m), &inc, v, &inc );
        mimo( m, y, t, I, R, lda, v, norma );
      }
    }
  }
}
```

Caso de Estudio: MIMO. Solución con hilos

Caso de Estudio: MIMO. Solución con hilos

Apartado 5

Diseño de algoritmos paralelos basado en descomposición de tareas

- Caso de Estudio: Sudoku
- Caso de Estudio: Descomposición de Cholesky
- Caso de Estudio: MIMO

- Algoritmos de búsqueda de una solución en un espacio
  - Caso de Estudio: Sudoku
- Algoritmos basados en particionado de datos
  - Caso de Estudio: Descomposición de Cholesky
- Algoritmos de búsqueda de la solución óptima en un espacio
  - Caso de Estudio: MIMO

Diseño de algoritmos paralelos basado en descomposición de tareas

- Algoritmos de búsqueda de una solución en un espacio
  - Caso de Estudio: Sudoku
- Algoritmos basados en particionado de datos
  - Caso de Estudio: Descomposición de Cholesky
- Algoritmos de búsqueda de la solución óptima en un espacio
  - Caso de Estudio: MIMO

Diseño de algoritmos paralelos basado en descomposición de tareas

- El problema de utilizar threads directamente es el de concurrencia versus paralelismo.
- Concurrencia:
  - Concurrencia se refiere a actividades separadas que progresan hacia delante.
  - Los threads del SO están diseñados para esto.
  - Si los threads exceden los threads *hardware*, el SO reparte el "tiempo" entre los threads.
  - Adecuado para obtener buenas respuestas por parte de actividades inconexas.
  - Si la tarea total es única, esta solución no es adecuada.
- Paralelismo:
  - La idea es "dividir el trabajo" (no el tiempo) para mantener el hardware "ocupado" y coordinar este trabajo para un uso eficiente de los recursos.

Diseño de algoritmos paralelos basado en descomposición de tareas

- La concurrencia recursiva crea concurrencia exponencial (explosiva).
- Por ejemplo, ejecutar `fib(25)` en un procesador con 16 cores consumió 0.4 terabytes de memoria virtual antes de abortar.

```
double fib_thread(int n) {
  if (n<2) return n;
  else {
    double x;
    auto t = std::thread( [&]{x=fib_thread(n-2);} );
    double y = fib_thread(n-1);
    t.join();
    return x+y;
  }
}
```

- Elegir el número idóneo de threads a utilizar en cada nivel de concurrencia es muy difícil.
- Es necesario un mecanismo mediante el que especificar el trabajo a ejecutar en paralelo (tareas) y un planificador capaz de distribuir dicho trabajo entre los threads.
- Idealmente, el planificador debería:
  - Ejecutar las tareas de manera que utilicen la memoria eficientemente.
  - No disponer de un control centralizado que suponga un cuello de botella.

Paralelismo de Tareas

En determinados problemas (¿no todos?), el paralelismo de tareas es mejor solución que otras construcciones. Por ejemplo:

- Bucles sin límites condicionales
- Algoritmos recursivos
- Esquemas productor-consumidor

Ejemplo de paralelización basada en hilos

```
void traverse_list(List l) {
  Element e;
  #pragma omp parallel private(e)
  for (e = l->first; e; e = e->next)
    #pragma omp single nowait
    process(e);
}
```

El Modelo de Tareas en OpenMP

Tareas (OpenMP 3.0): unidades de trabajo planificadas para ser ejecutadas en un momento indeterminado.

- También se pueden ejecutar inmediatamente
- Se componen de código y datos

Directiva creación:

```
#pragma omp task [clauses]
```

Cada hilo crea una tarea (empaquetando código y datos)

- Alcance de variables: `shared`, `private`, `firstprivate` (se inicializa en la creación), `default(shared|none)`
- Por defecto `firstprivate`, pero `shared` se hereda
- Las barreras de los hilos afectan a las tareas

Directiva sincronización:

```
#pragma omp taskwait
```

- Bloquea un hilo hasta que acaban sus tareas hijas (directas)

El Modelo de Tareas en OpenMP

Tareas (OpenMP 3.0): unidades de trabajo planificadas para ser ejecutadas en un momento indeterminado.

- También se pueden ejecutar inmediatamente
- Se componen de código y datos

Directiva creación:

```
#pragma omp task [clauses]
```

Cada hilo crea una tarea (empaquetando código y datos)

- Alcance de variables: `shared`, `private`, `firstprivate` (se inicializa en la creación), `default(shared|none)`
- Por defecto `firstprivate`, pero `shared` se hereda
- Las barreras de los hilos afectan a las tareas

Directiva sincronización:

```
#pragma omp taskwait
```

- Bloquea un hilo hasta que acaban sus tareas hijas (directas)

El Modelo de Tareas en OpenMP

Tareas (OpenMP 3.0): unidades de trabajo planificadas para ser ejecutadas en un momento indeterminado.

- También se pueden ejecutar inmediatamente
- Se componen de código y datos

Directiva creación:

```
#pragma omp task [clauses]
```

Cada hilo crea una tarea (empaquetando código y datos)

- Alcance de variables: `shared`, `private`, `firstprivate` (se inicializa en la creación), `default(shared|none)`
- Por defecto `firstprivate`, pero `shared` se hereda
- Las barreras de los hilos afectan a las tareas

Directiva sincronización:

```
#pragma omp taskwait
```

- Bloquea un hilo hasta que acaban sus tareas hijas (directas)

Modelo de Ejecución de Tareas

Las tareas se ejecutan por algún hilo del equipo que la generó

Recorrido de listas con `task`

```
void traverse_list(List l) {
  Element e;
  for (e = l->first; e; e = e->next)
    #pragma omp task
    process(e);       /* la variable e es firstprivate */
  #pragma omp taskwait
}
...
#pragma omp parallel    /* Un hilo crea las tareas y */
#pragma omp single      /* todos cooperan en su ejecución */
traverse_list(l);
```

- La cláusula `if` permite controlar la creación de tareas
- La cláusula `untied` permite migrar tareas entre hilos

Modelo de Ejecución de Tareas

Las tareas se ejecutan por algún hilo del equipo que la generó

Recorrido de listas con `task`

```
void traverse_list(List l) {
  Element e;
  for (e = l->first; e; e = e->next)
    #pragma omp task
    process(e);       /* la variable e es firstprivate */
  #pragma omp taskwait
}
...
#pragma omp parallel    /* Un hilo crea las tareas y */
#pragma omp single      /* todos cooperan en su ejecución */
traverse_list(l);
```

- La cláusula `if` permite controlar la creación de tareas
- La cláusula `untied` permite migrar tareas entre hilos

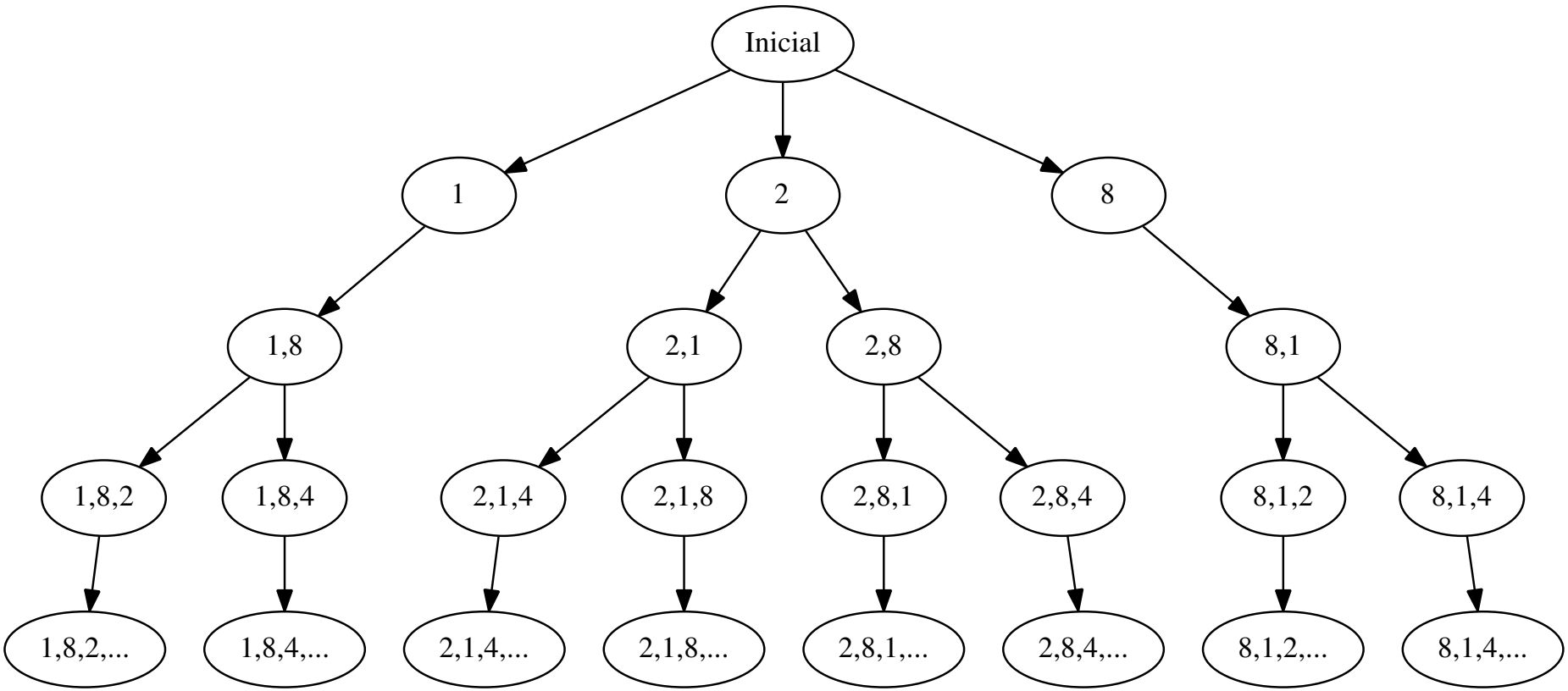
Recursión con directiva `task`

Quicksort con `task`

```
void quick_sort(float *data, int n)
{
    int p;
    if (n < N_MIN) {
        insertion_sort(data, n);
    } else {
        p = partition(data, n);
        #pragma omp task
        quick_sort(data, p);
        #pragma omp task
        quick_sort(data+p+1, n-p-1);
    }
}
...
#pragma omp parallel
#pragma omp single
quick_sort(data, n);
```

Sudoku: DAG y tareas

Cada nodo del árbol va a ser una tarea.



Sudoku: Algoritmo *backtracking* con tareas

- El algoritmo de backtracking para el Sudoku con tareas es sencillo.
- Propuesta: Crear una tarea para cada llamada recursiva a `sudoku_sol`.

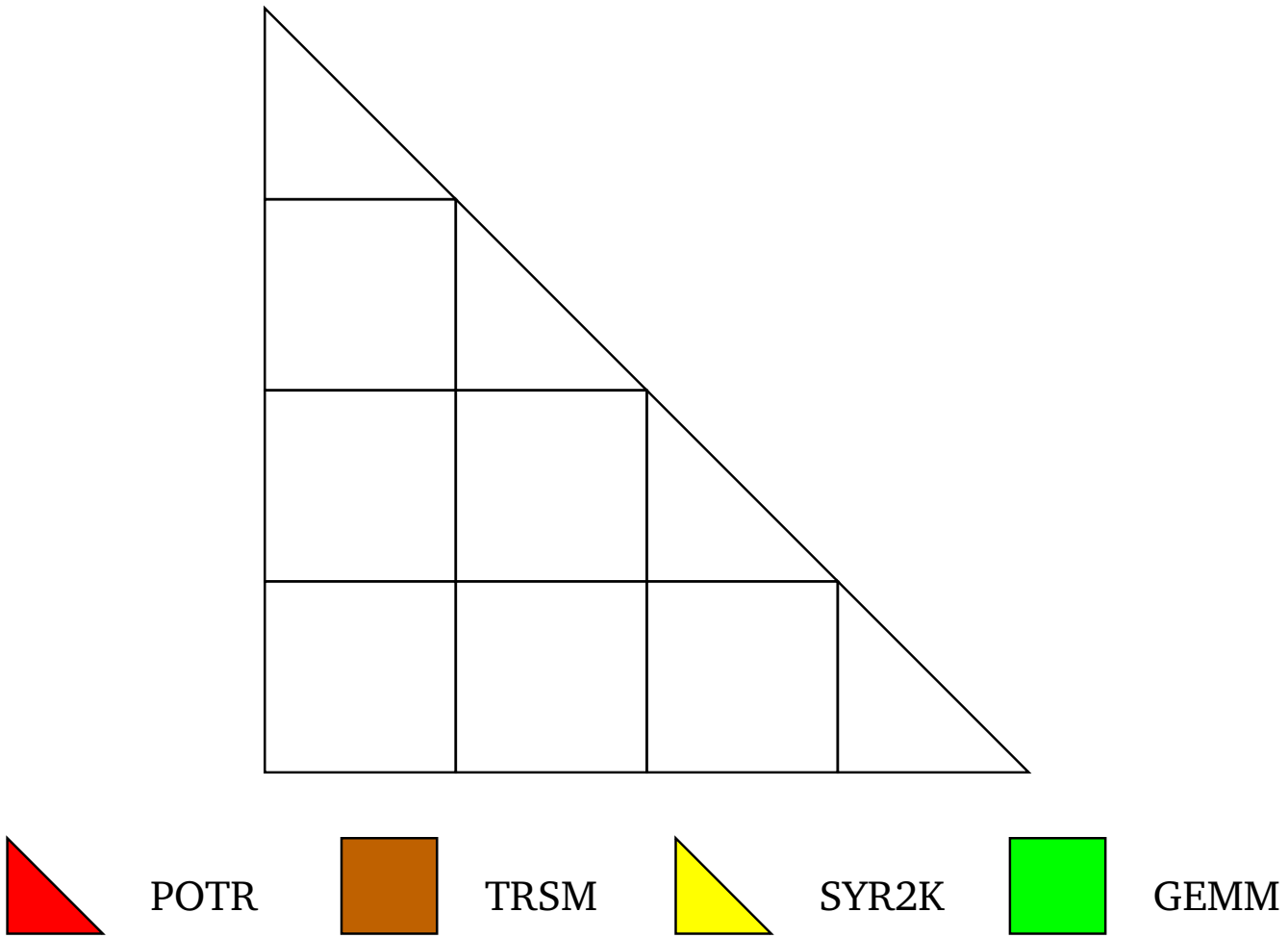
```
. . .
#pragma omp task [clausulas]
sudoku_sol( i, j+1, sol, mascara );
. . .
```

- Hay que reflexionar acerca de las cláusulas en la creación de la tarea. (Observación: `sol` y `mascara` son vectores.)
- En el programa principal hay que crear los hilos

```
. . .
#pragma omp parallel
#pragma omp single
sudoku_sol( i, j+1, sol, mascara );
. . .
```

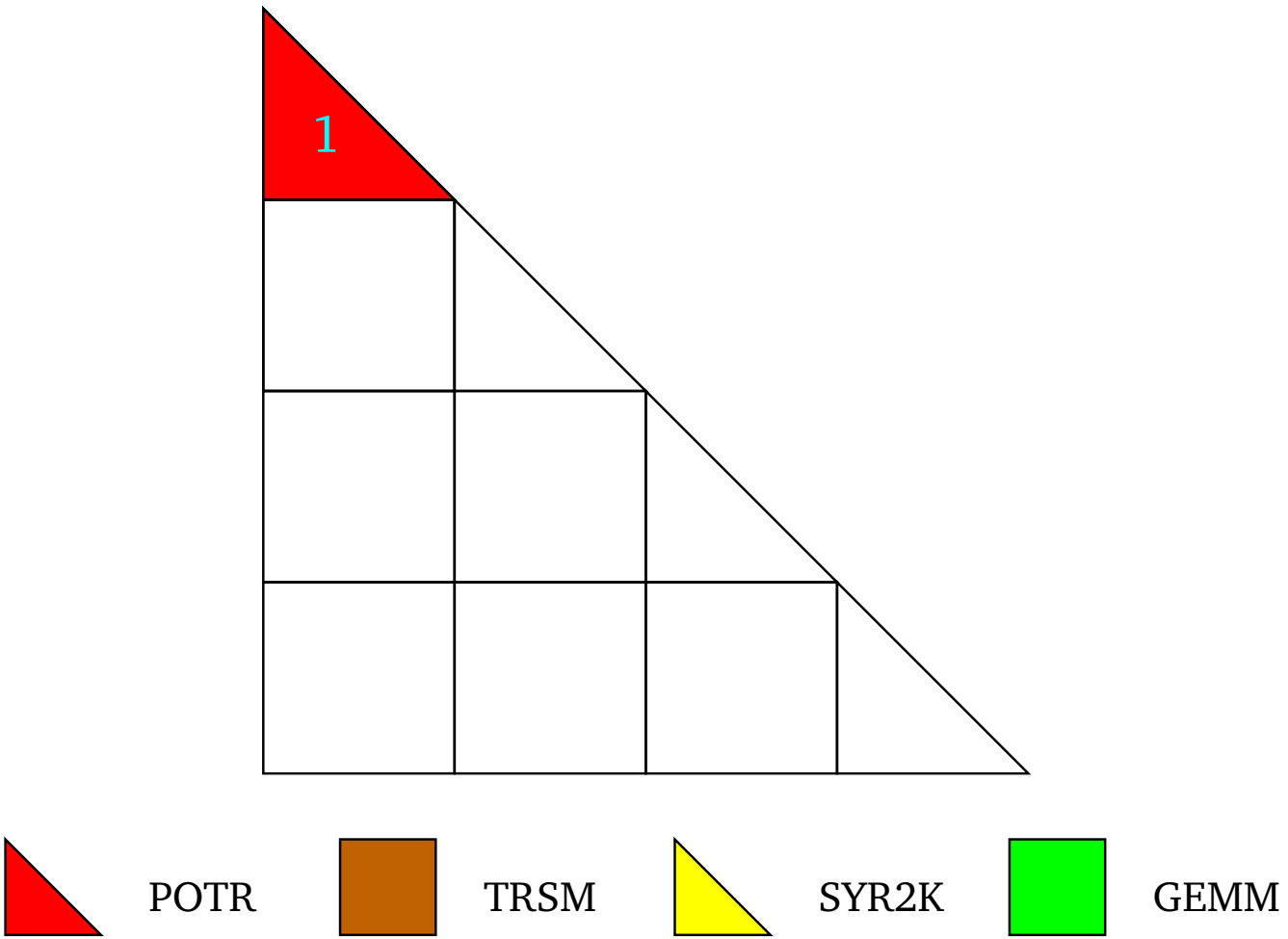
- Hay que considerar la necesidad o no de sincronización de tareas (`taskwait`).

Cholesky por paralelo (tareas)

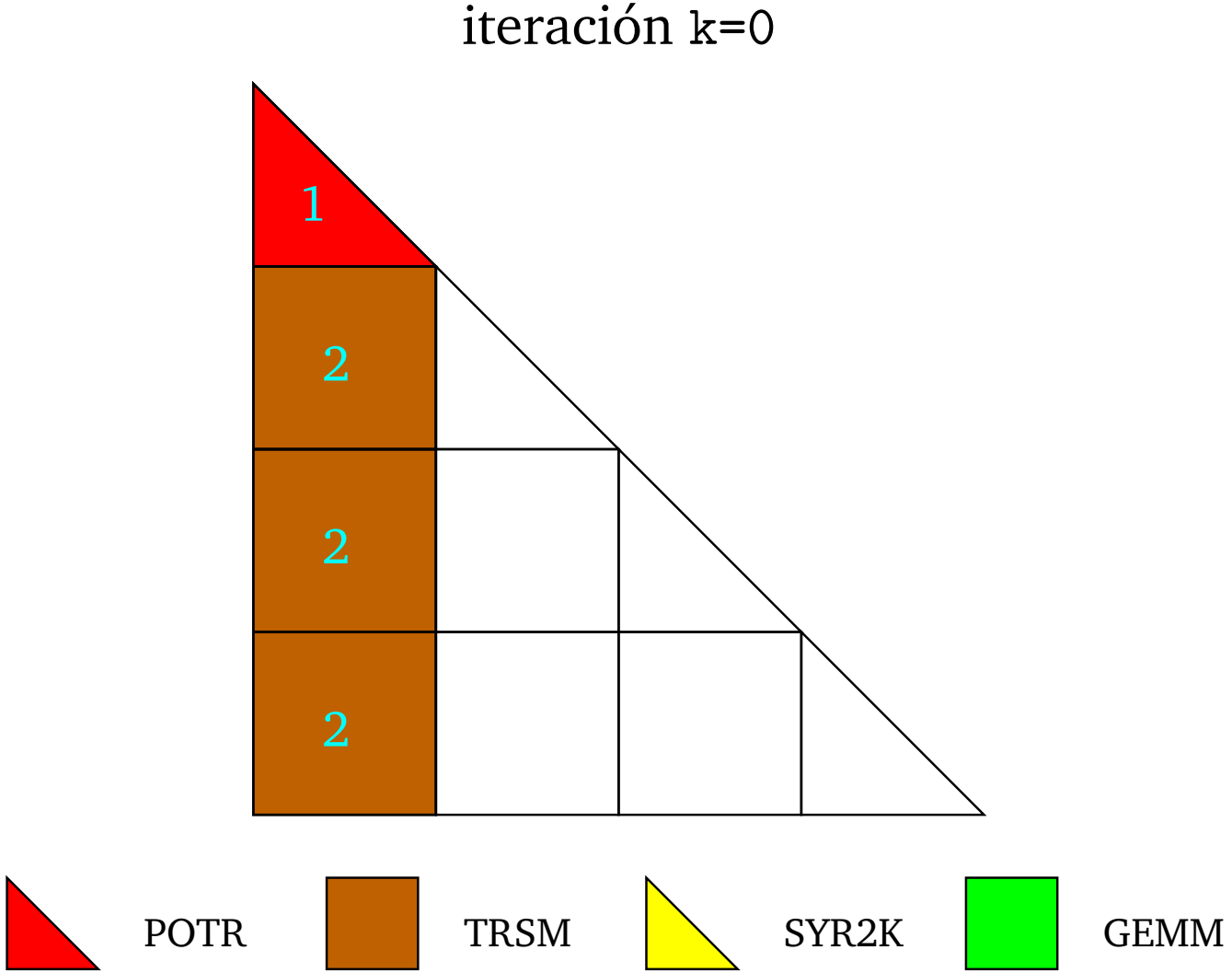


Cholesky por paralelo (tareas)

iteración `k=0`

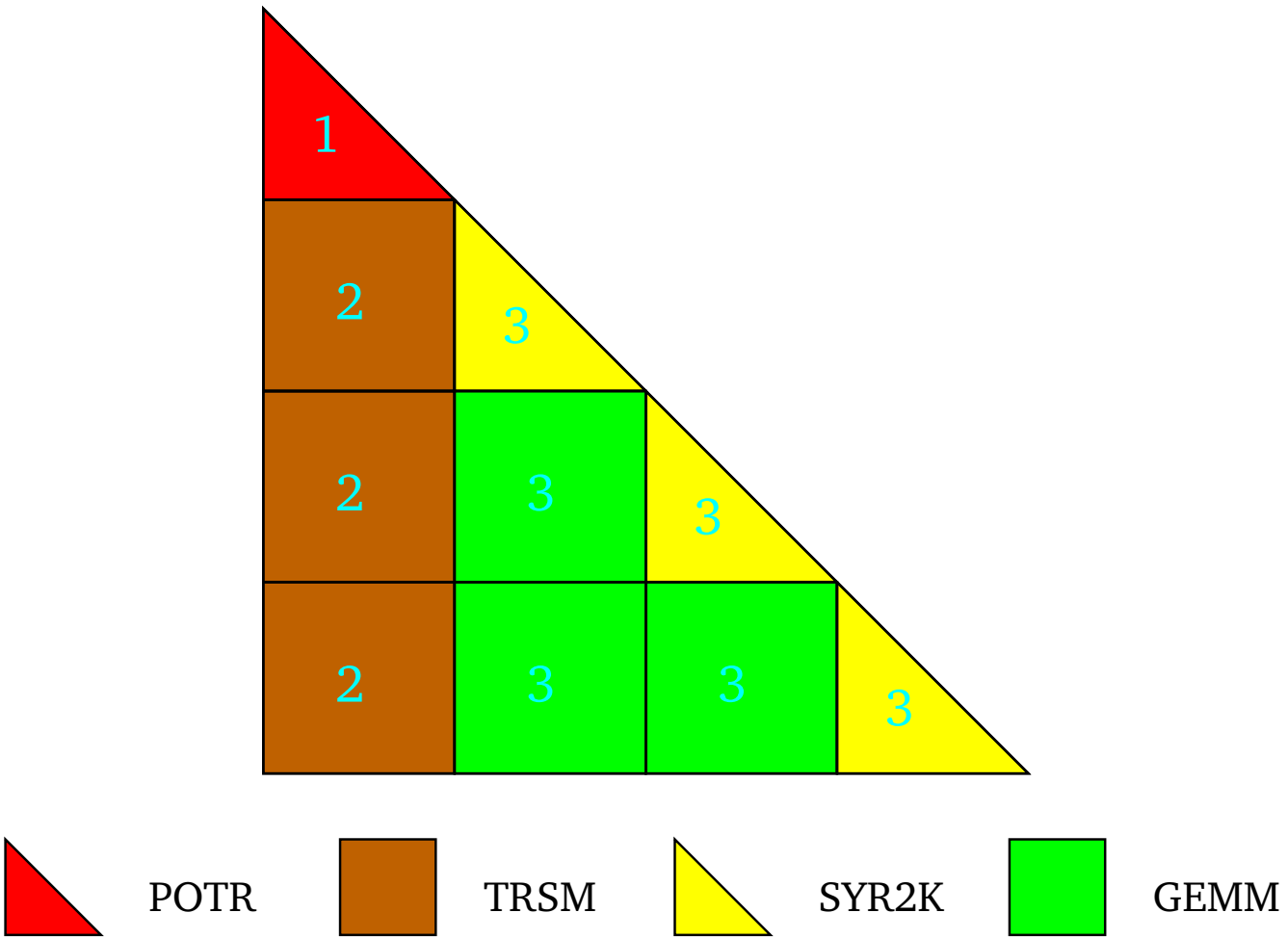


Cholesky por paralelo (tareas)



Cholesky por paralelo (tareas)

iteración `k=0`



Cholesky: algoritmo por bloques

Esquema del algoritmo de Cholesky paralelo con tareas OpenMP

```
function CHOL_TAREAS( A, b ) return C
    N ← n/b
    for k = 1 : N do
        #pragma omp task
        POTRF
        for i = k + 1 : N do
            #pragma omp task
            TRSM
        end for
        for i = k + 1 : N do
            for j = k + 1 : i − 1 do
                #pragma omp task
                GEMM
            end for
            #pragma omp task
            SYR2K
        end for
    end for
end function
```

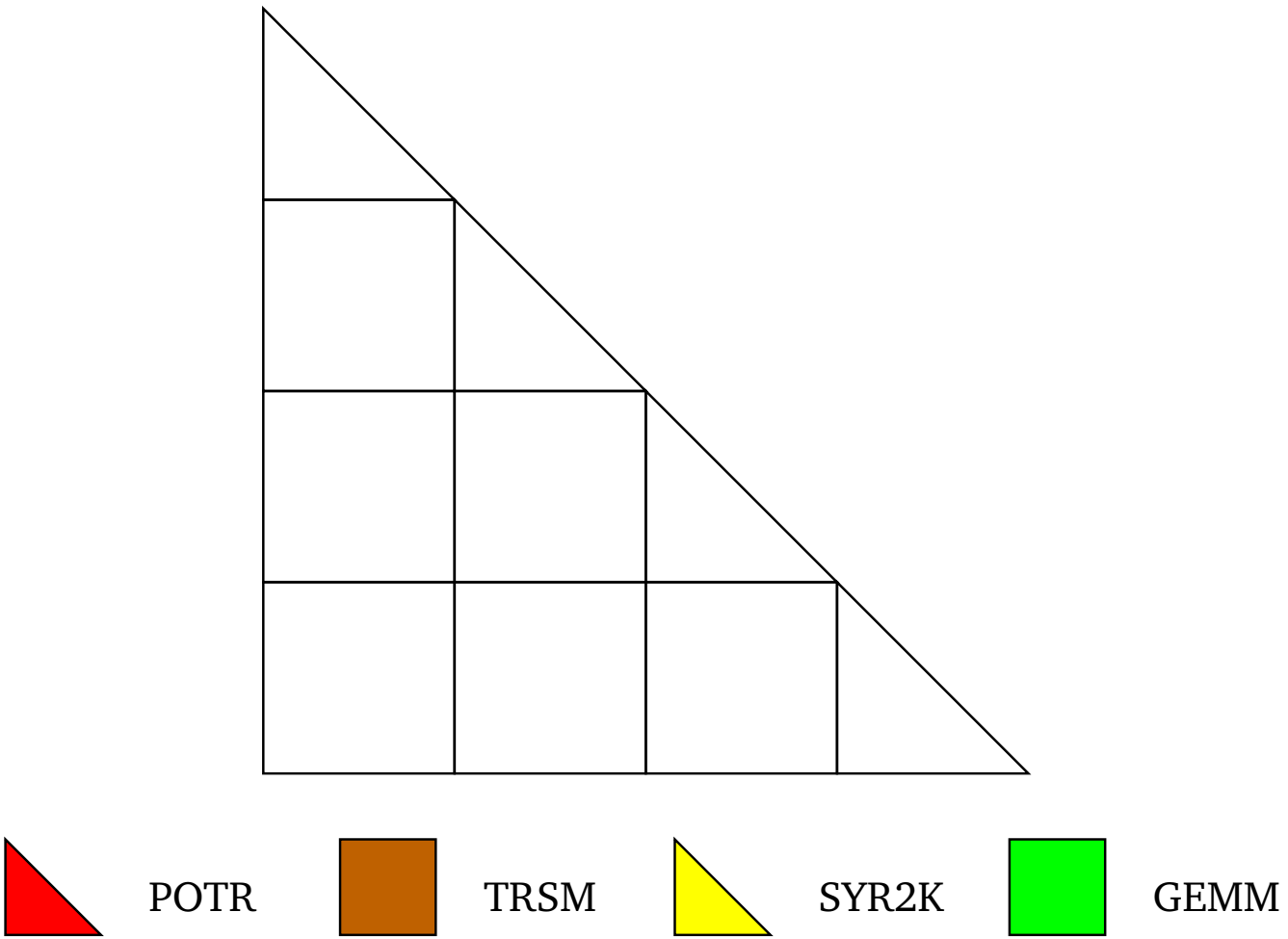
Pregunta: ¿Es correcto el código anterior?

Cholesky: algoritmo por bloques

```
function CHOL_TAREAS( A, b ) return C
    N ← n/b
    for k = 1 : N do
        #pragma omp task
        POTRF
        for i = k + 1 : N do
            #pragma omp task
            TRSM
        end for
        #pragma omp taskwait
        for i = k + 1 : N do
            for j = k + 1 : i − 1 do
                #pragma omp task
                GEMM
            end for
            #pragma omp task
            SYR2K
        end for
        #pragma omp taskwait
    end for
end function
```

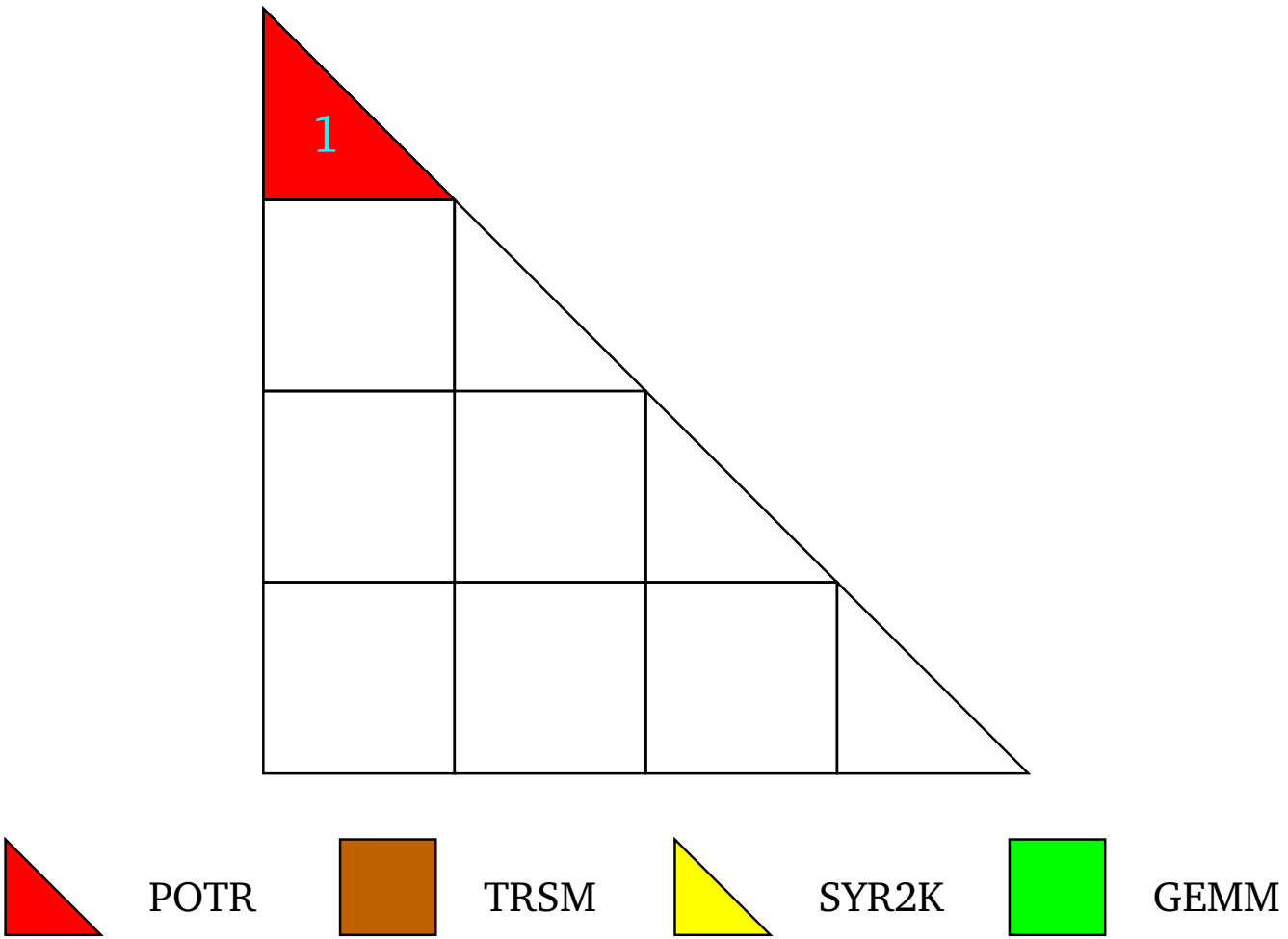
Cholesky por bloques paralelo con DAG

iteración k=0



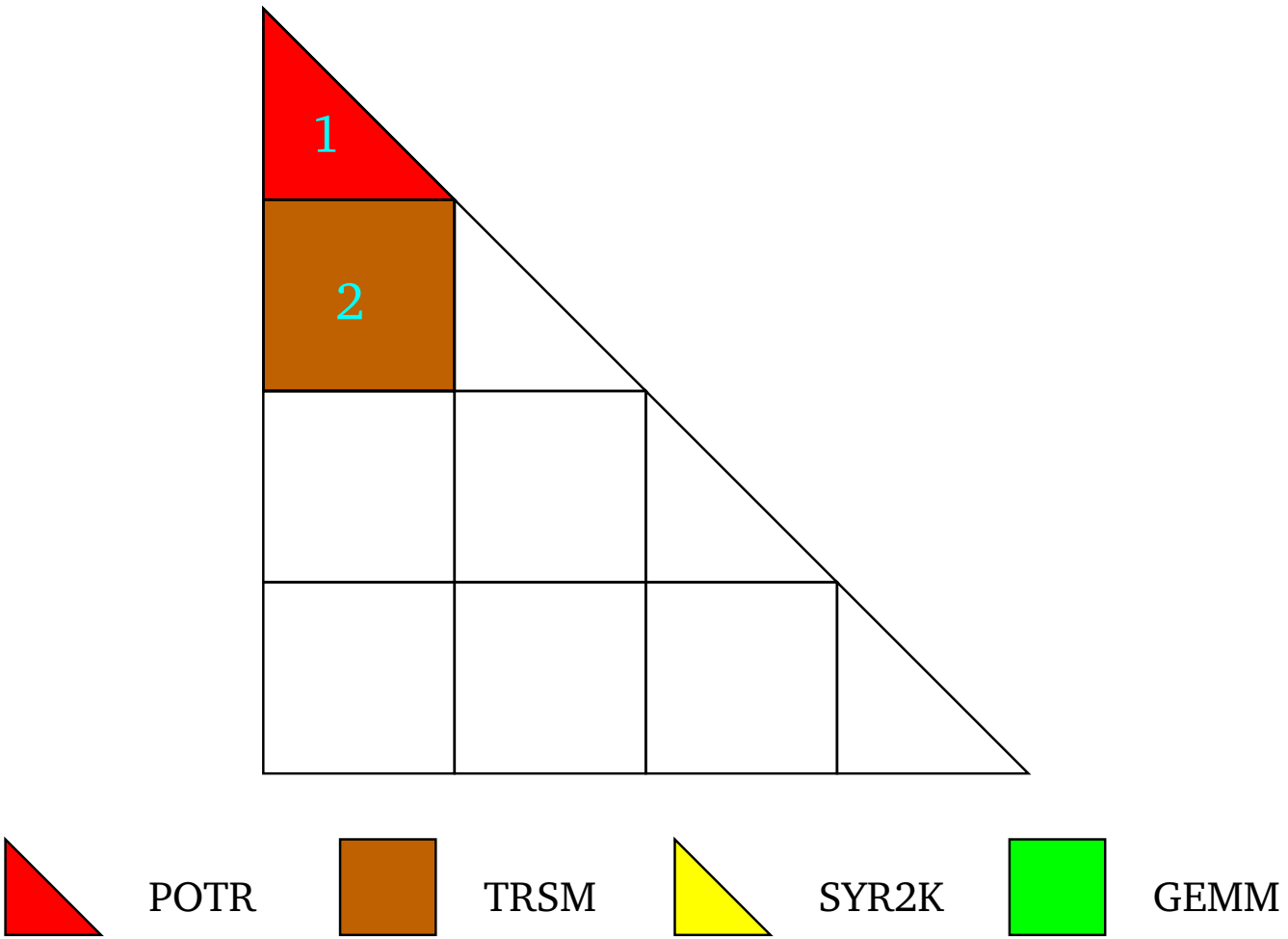
Cholesky por bloques paralelo con DAG

iteración `k=0`



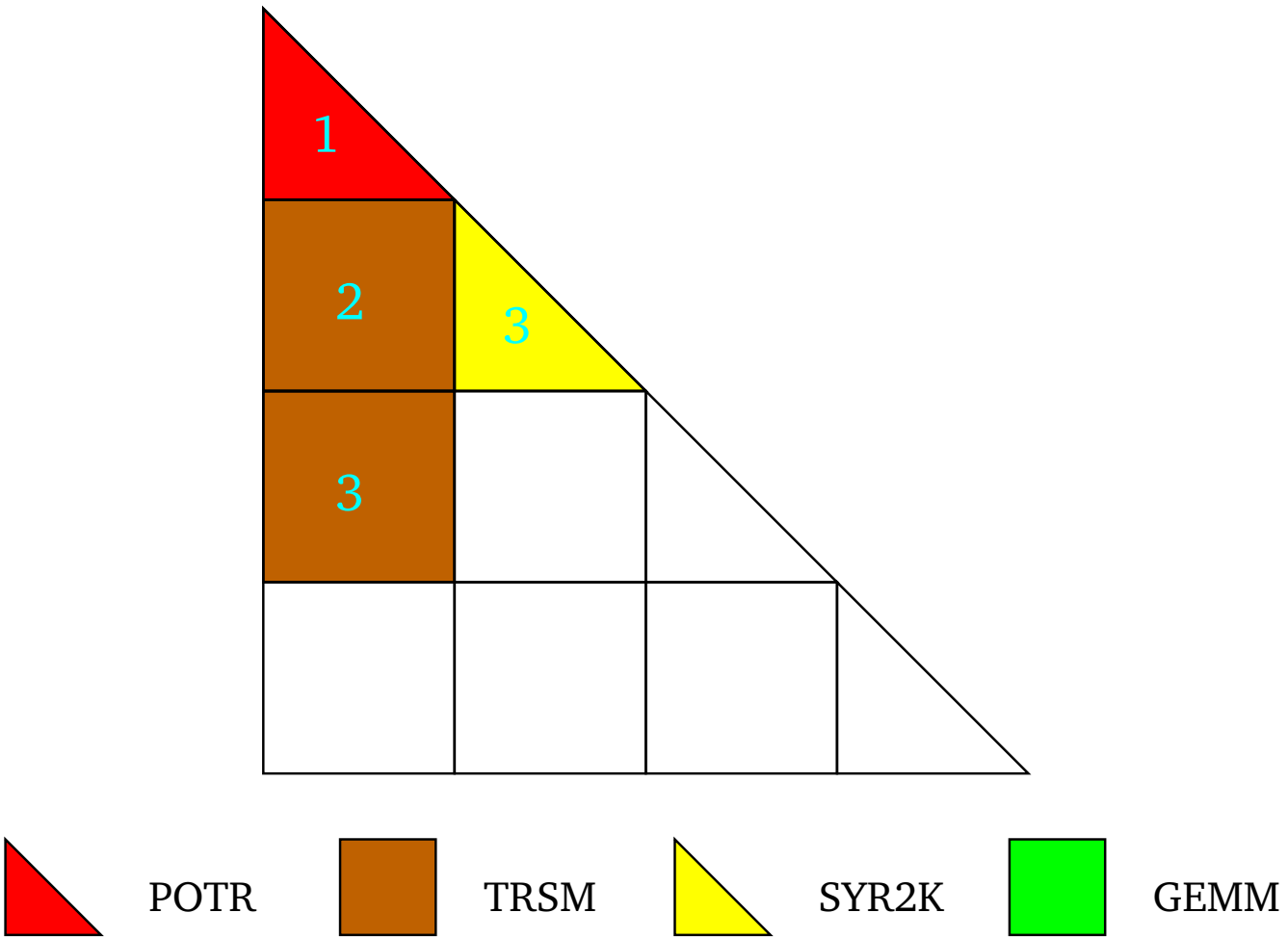
Cholesky por bloques paralelo con DAG

iteración `k=0`



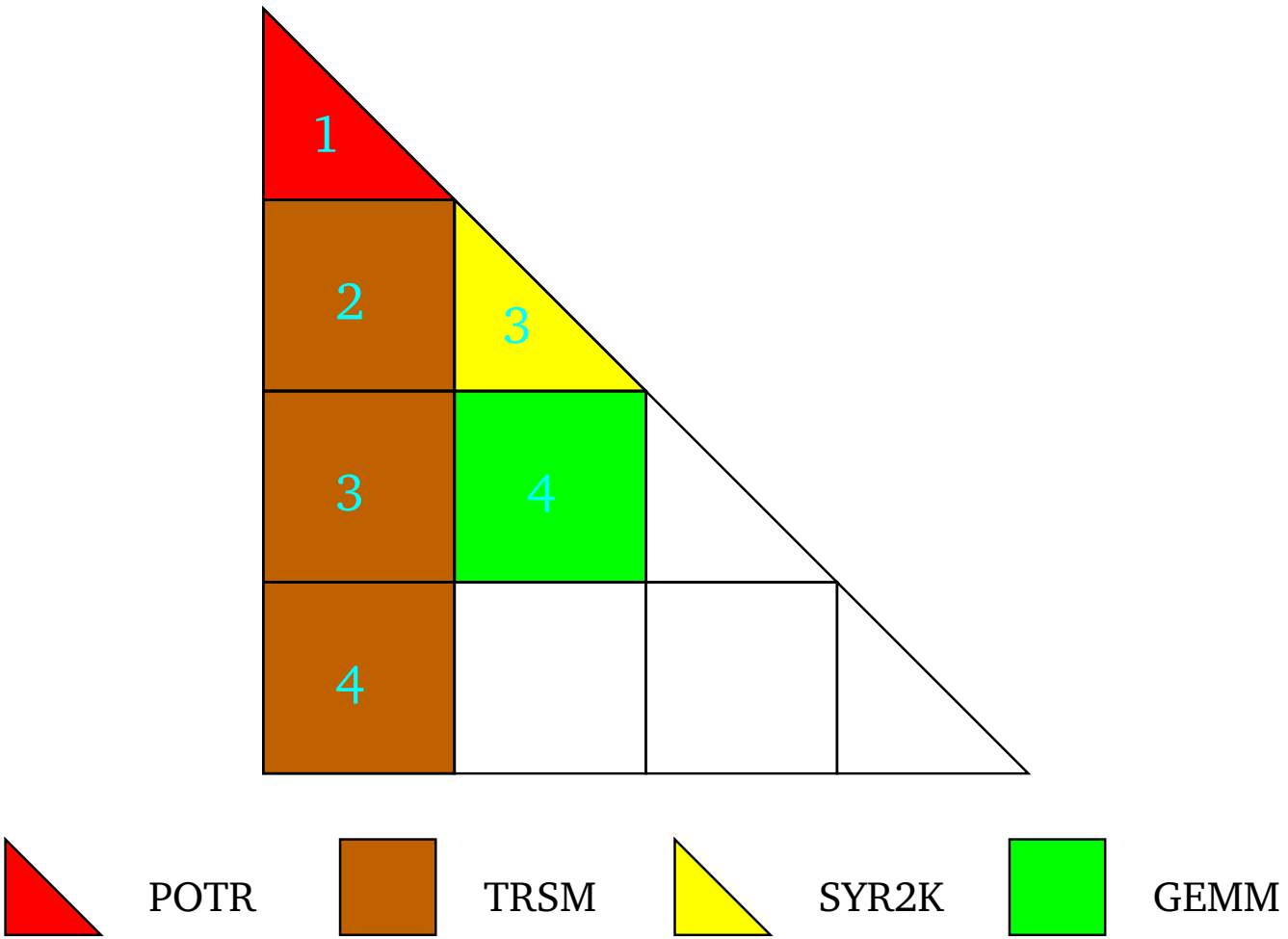
Cholesky por bloques paralelo con DAG

iteración `k=0`



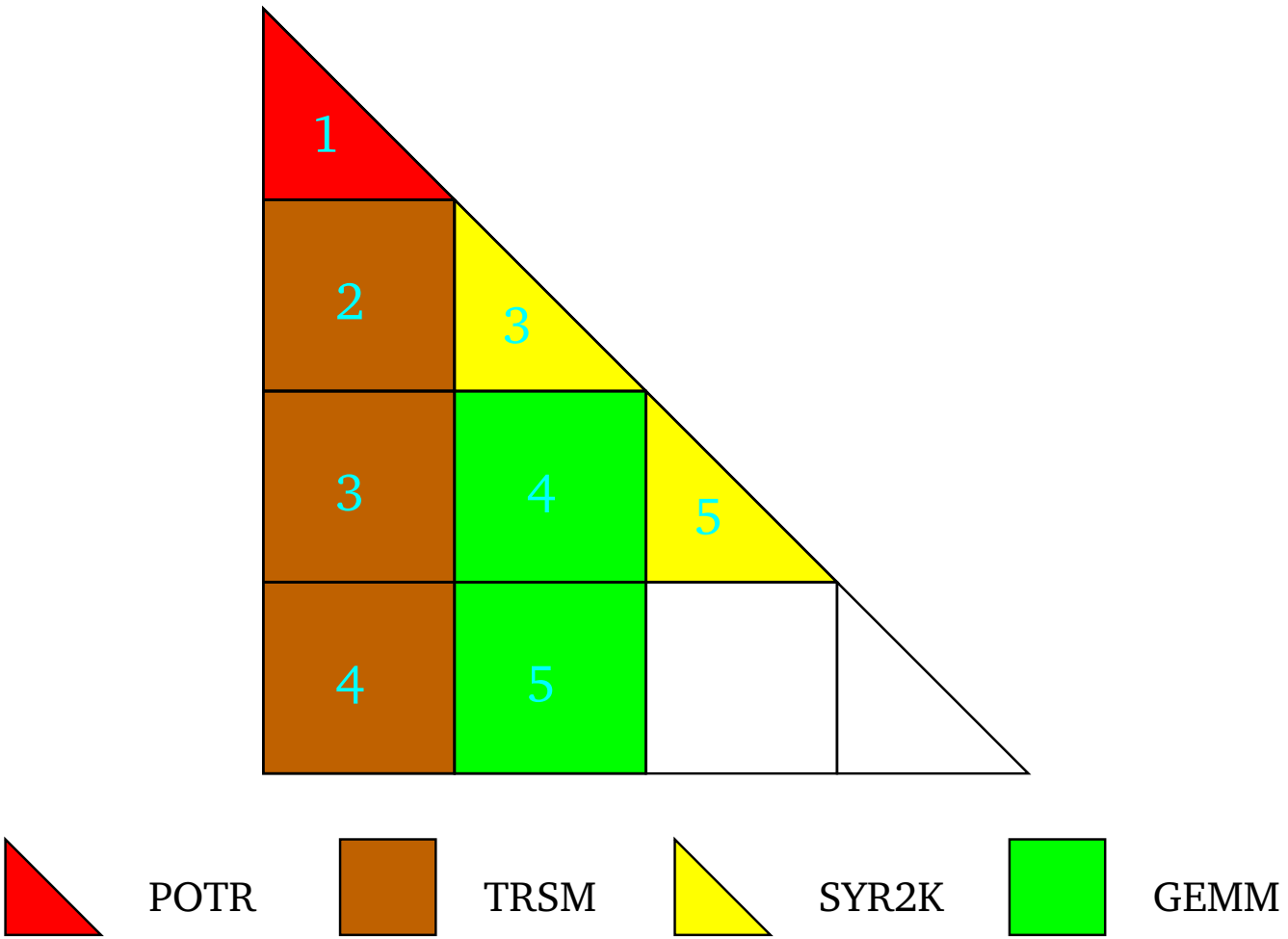
Cholesky por bloques paralelo con DAG

iteración `k=0`



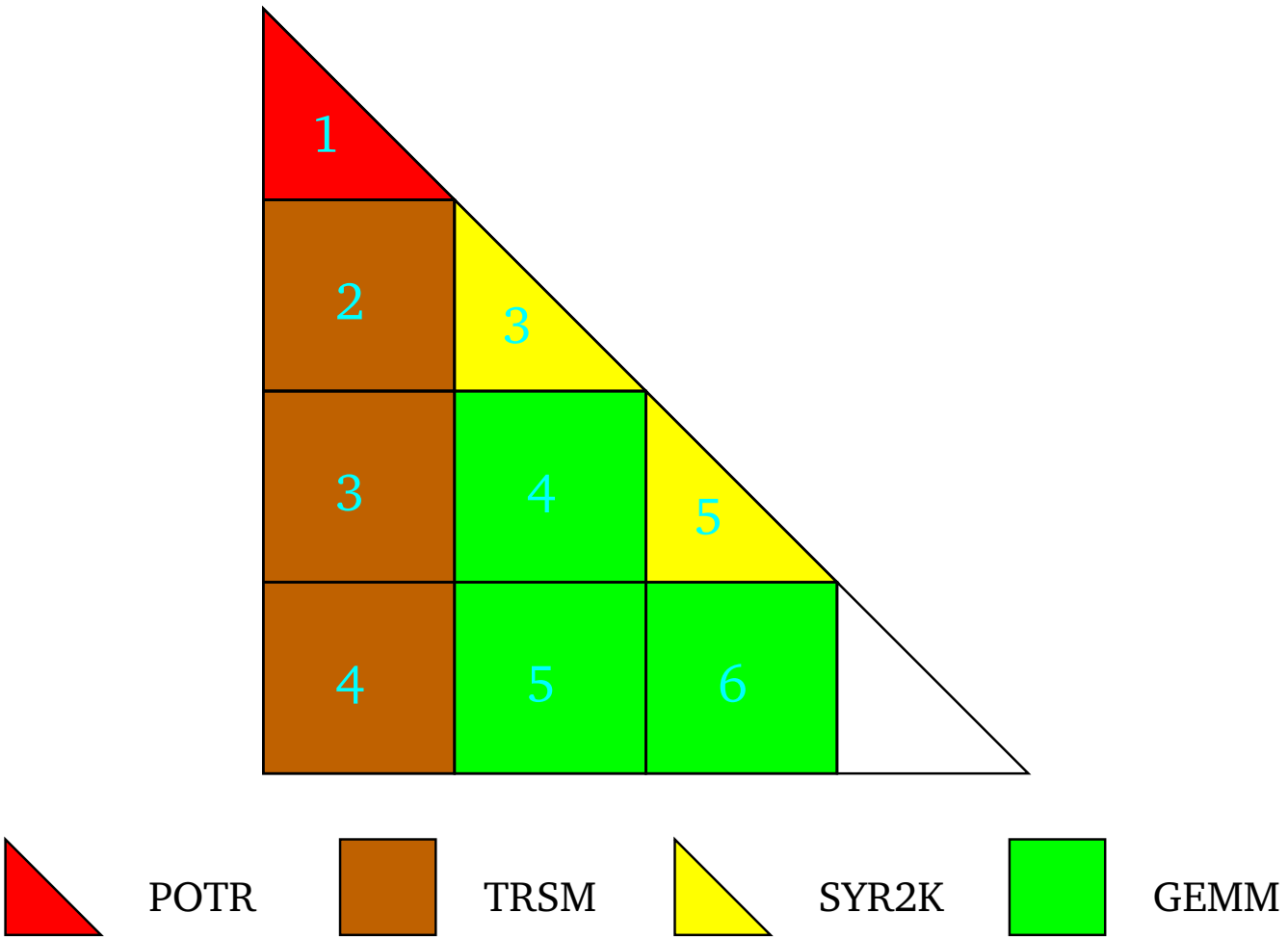
Cholesky por bloques paralelo con DAG

iteración `k=0`



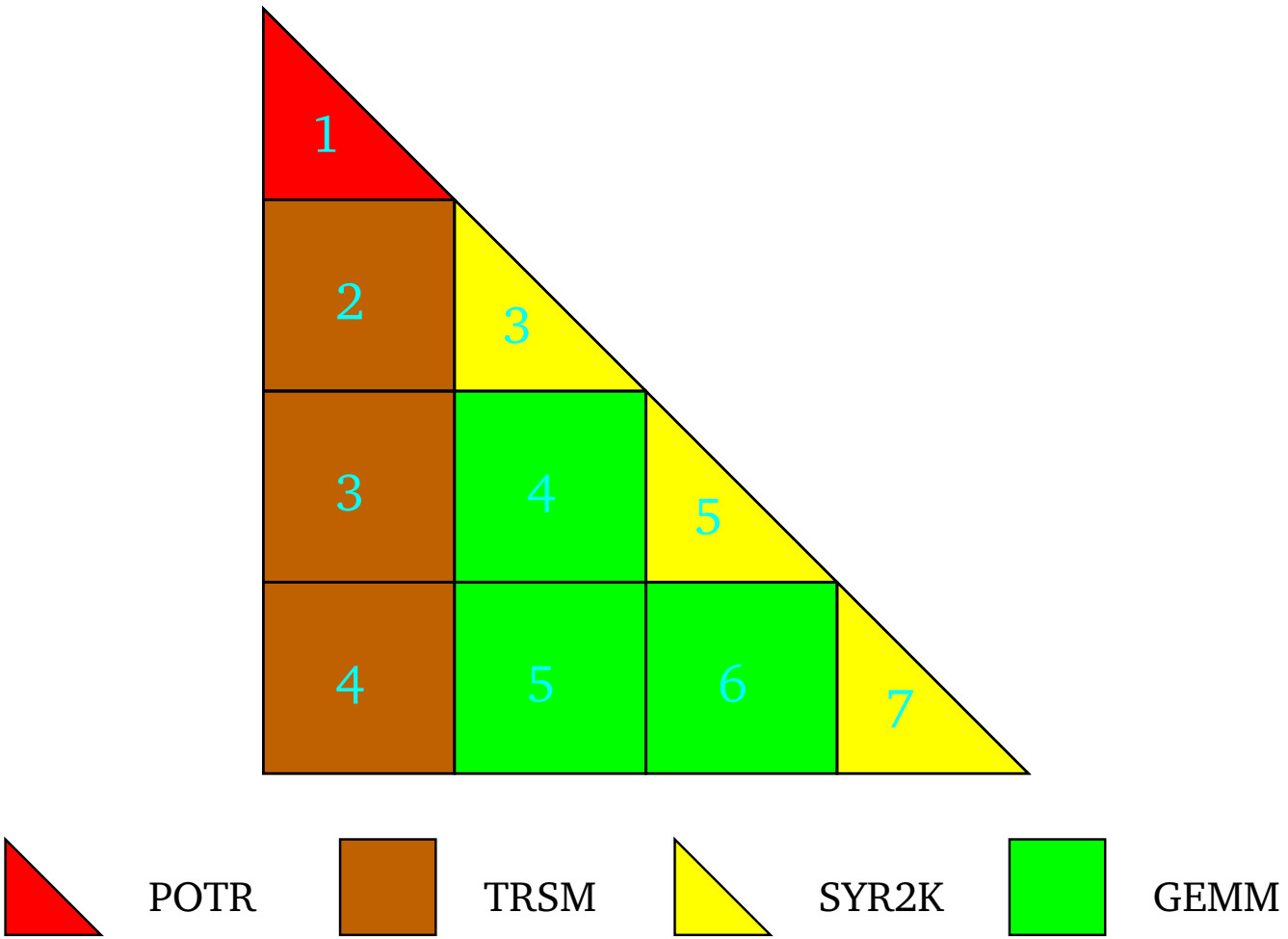
Cholesky por bloques paralelo con DAG

iteración `k=0`

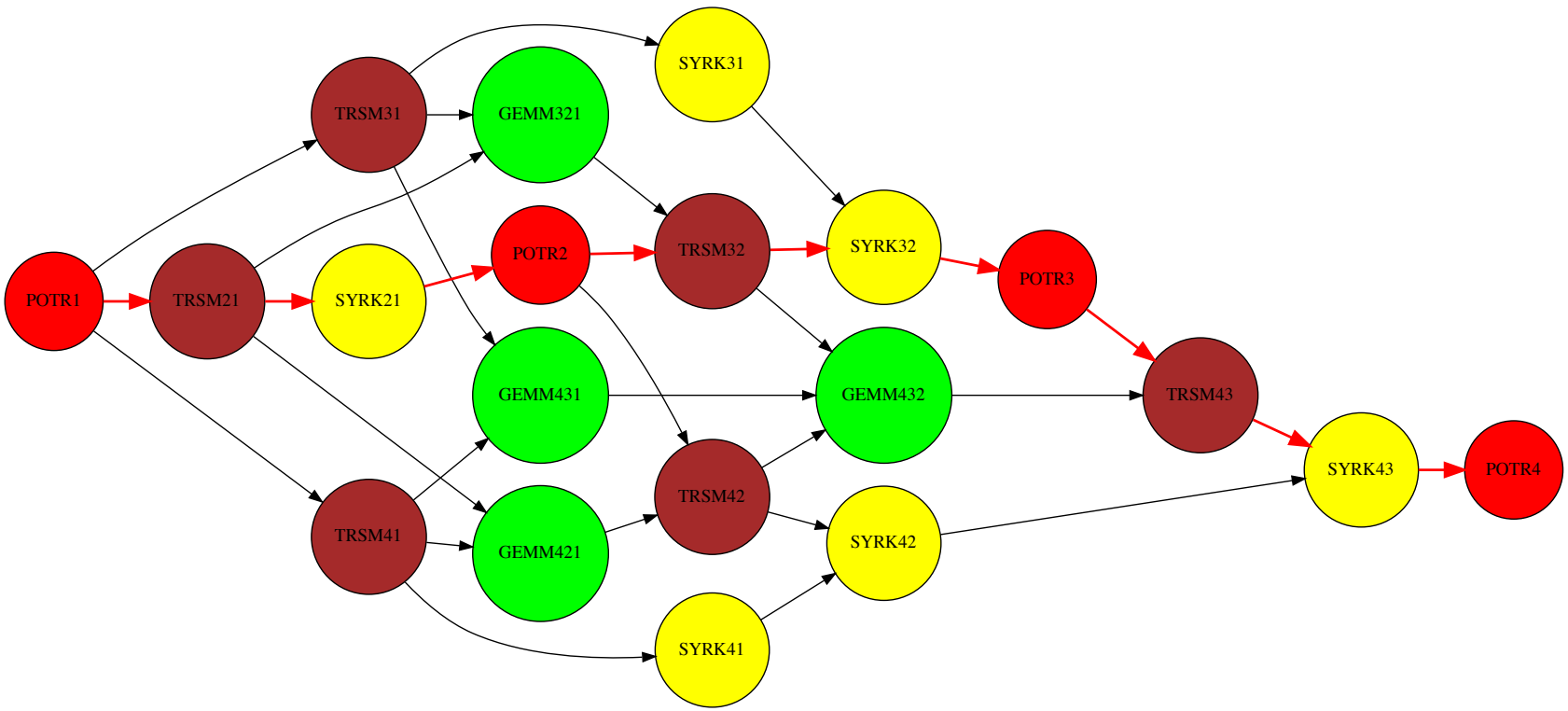


Cholesky por bloques paralelo con DAG

iteración `k=0`



Grafo (DAG) de Cholesky



OpenMP 4.0: La cláusula depend

Cláusula `depend`

```
#pragma omp task depend(dependency-type: list)
... structured block ...
```

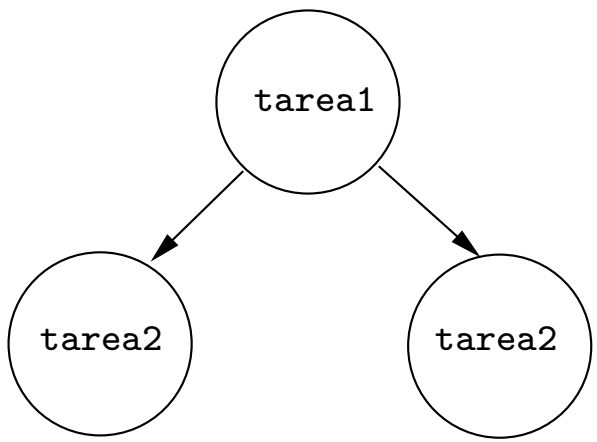
- Una dependencia de tarea se produce cuando la tarea predecesora se ha completado.
- La especificación dice lo siguiente:
  - `in` dependency-type: the generated task will be a dependent task of all previously generated sibling tasks that reference at least one of the list items in an `out` or `inout` clause.
  - `out` and inout dependency-type: The generated task will be a dependent task of all previously generated sibling tasks that reference at least one of the list items in an `in`, `out`, or `inout` clause.
  - The list items in a depend clause may include array sections.

La cláusula depend: ejemplo

```
void process_in_parallel) {
 #pragma omp parallel
 #pragma omp single
 {
   int x = 1;
   ...
   for( int i = 0; i < T; ++i ) {
   #pragma omp task shared(x) depend(out:x)
   tarea1(...);
   #pragma omp task shared(x) depend(in:x)
   tarea2(...);
   #pragma omp task shared(x) depend(in:x)
   tarea3(...);
   }
 } // end omp single, omp parallel
```

La cláusula depend: ejemplo

```
void process_in_parallel) {
 #pragma omp parallel
 #pragma omp single
 {
   int x = 1;
   ...
   for( int i = 0; i < T; ++i ) {
   #pragma omp task shared(x) depend(out:x)
   tarea1(...);
   #pragma omp task shared(x) depend(in:x)
   tarea2(...);
   #pragma omp task shared(x) depend(in:x)
   tarea3(...);
   }
 } // end omp single, omp parallel
```



La cláusula depend: Cholesky

function CHOL_ESCALAR_PARALELO(A) **return** C
  $C \leftarrow A$
  for $k = 1 \rightarrow n$ **do**

    $C(k,k) \leftarrow \sqrt{C(k,k)}$
    for $i = k+1 \rightarrow n$ **do**

      $C(i,k) = C(i,k)/C(k,k)$
    end for
    for $i = k+1 \rightarrow n$ **do**
      for $j = k+1 \rightarrow i-1$ **do**

        $C(i,j) = C(i,j) - C(i,k) \cdot C(j,k)$
      end for

      $C(i,i) = C(i,i) - C(i,k) \cdot C(i,k)$
    end for
  end for
end function

La cláusula depend: Cholesky

function CHOL_ESCALAR_PARALELO(A) **return** C
  $C \leftarrow A$
  for $k = 1 \rightarrow n$ **do**
    `#pragma omp task depend(inout:C(k,k))`
    $C(k,k) \leftarrow \sqrt{C(k,k)}$
    for $i = k+1 \rightarrow n$ **do**

      $C(i,k) = C(i,k)/C(k,k)$
    end for
    for $i = k+1 \rightarrow n$ **do**
      for $j = k+1 \rightarrow i-1$ **do**

        $C(i,j) = C(i,j) - C(i,k) \cdot C(j,k)$
      end for

      $C(i,i) = C(i,i) - C(i,k) \cdot C(i,k)$
    end for
  end for
end function

La cláusula depend: Cholesky

function CHOL_ESCALAR_PARALELO(A) **return** C
  $C \leftarrow A$
  for $k = 1 \rightarrow n$ **do**
    `#pragma omp task depend(inout:C(k,k))`
    $C(k,k) \leftarrow \sqrt{C(k,k)}$
    for $i = k+1 \rightarrow n$ **do**
      `#pragma omp task depend(in:C(k,k)) depend(inout:C(i,k))`
      $C(i,k) = C(i,k)/C(k,k)$
    end for
    for $i = k+1 \rightarrow n$ **do**
      for $j = k+1 \rightarrow i-1$ **do**

        $C(i,j) = C(i,j) - C(i,k) \cdot C(j,k)$
      end for

      $C(i,i) = C(i,i) - C(i,k) \cdot C(i,k)$
    end for
  end for
end function

La cláusula depend: Cholesky

function CHOL_ESCALAR_PARALELO(A) **return** C
  $C \leftarrow A$
  for $k = 1 \rightarrow n$ **do**
    `#pragma omp task depend(inout:C(k,k))`
    $C(k,k) \leftarrow \sqrt{C(k,k)}$
    for $i = k+1 \rightarrow n$ **do**
      `#pragma omp task depend(in:C(k,k)) depend(inout:C(i,k))`
      $C(i,k) = C(i,k)/C(k,k)$
    end for
    for $i = k+1 \rightarrow n$ **do**
      for $j = k+1 \rightarrow i-1$ **do**
        `#pragma omp task depend(in:C(i,k),C(j,k)) depend(inout:C(i,j))`
        $C(i,j) = C(i,j) - C(i,k) \cdot C(j,k)$
      end for

      $C(i,i) = C(i,i) - C(i,k) \cdot C(i,k)$
    end for
  end for
end function

La cláusula depend: Cholesky

function CHOL_ESCALAR_PARALELO(A) **return** C
  $C \leftarrow A$
  for $k = 1 \rightarrow n$ **do**
    `#pragma omp task depend(inout:C(k,k))`
    $C(k,k) \leftarrow \sqrt{C(k,k)}$
    for $i = k+1 \rightarrow n$ **do**
      `#pragma omp task depend(in:C(k,k)) depend(inout:C(i,k))`
      $C(i,k) = C(i,k)/C(k,k)$
    end for
    for $i = k+1 \rightarrow n$ **do**
      for $j = k+1 \rightarrow i-1$ **do**
        `#pragma omp task depend(in:C(i,k),C(j,k)) depend(inout:C(i,j))`
        $C(i,j) = C(i,j) - C(i,k) \cdot C(j,k)$
      end for
      `#pragma omp task depend(in:C(i,k)) depend(inout:C(i,i))`
      $C(i,i) = C(i,i) - C(i,k) \cdot C(i,k)$
    end for
  end for
end function

Caso de Estudio: MIMO. Solución con tareas

```
void mimo( int n, double *x, int t, double *I, double *R, int lda, double *b, double nrm ) {
  int k, inc = 1;
  if( n==0 ) {
    if( nrm < minimo ) {
      minimo = nrm;
      dcopy_( &lda, x, &inc, sol, &inc );
    }
  } else {
    for( k=0; k<t; k++ ) {
      int m = n-1;
      x(m) = I(k);
      double r = R(m,m)*x(m) - b(m);
      double norma = nrm + r*r;
      if( norma < minimo ) {
        double v[m], y[lda];
        dcopy_( &m, b, &inc, v, &inc );
        dcopy_( &lda, x, &inc, y, &inc );
        double alpha = -x(m);
        daxpy_( &m, &alpha, &R(0,m), &inc, v, &inc );
        mimo( m, y, t, I, R, lda, v, norma );
      }
    }
  }
}
```

- Se puede generar una tarea por cada llamada recursiva.
- Atención a:
  - Regiones críticas.
  - Sincronización de tareas (`taskwait`).
  - Vectores “automáticos”.

Caso de Estudio: MIMO. Solución con tareas

```
void mimo( int n, double *x, int t, double *I, double *R, int lda, double *b, double nrm ) {
  int k, inc = 1;
  if( n==0 ) {
    if( nrm < minimo ) {
      minimo = nrm;
      dcopy_( &lda, x, &inc, sol, &inc );
    }
  } else {
    for( k=0; k<t; k++ ) {
      int m = n-1;
      x(m) = I(k);
      double r = R(m,m)*x(m) - b(m);
      double norma = nrm + r*r;
      if( norma < minimo ) {
        double v[m], y[lda];
        dcopy_( &m, b, &inc, v, &inc );
        dcopy_( &lda, x, &inc, y, &inc );
        double alpha = -x(m);
        daxpy_( &m, &alpha, &R(0,m), &inc, v, &inc );
        mimo( m, y, t, I, R, lda, v, norma );
      }
    }
  }
}
```

- Se puede generar una tarea por cada llamada recursiva.
- Atención a:
  - Regiones críticas.
  - Sincronización de tareas (`taskwait`).
  - Vectores “automáticos”.