

Introducción a Cilk y TBB

ECAR 2017

David Alejandro González Márquez

Departamento de Computación
Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires

29/09/17



- Creado en 1994 por el MIT.
- Originalmente escrito en ANSI C, agregando *keywords* para explotar paralelismo.
- Cilk++ 1.0 fue lanzado en diciembre de 2008 por Cilk Arts.
- En julio de 2009 Cilk Arts se integra a Intel.
- Intel Cilk Plus es lanzado en Intel Composer XE 2010.
- En mayo de 2017 Intel Cilk Plus pasa a ser parte de TBB.

- Extensiones simples a C y C++.
- Permiten identificar tareas y paralelismo sobre datos.
- Fácil de aprender y utilizar.
- Semántica secuencial, determinístico.

- *Keywords:*
Permiten expresar oportunidad de paralelismo.
- *Reducers:*
Elimina la contención de variables compartidas.
Automáticamente crea vistas donde las variables sean necesarias.
- *Array Notation:*
Crea paralelismo sobre arreglos o secciones de arreglos.
- *SIMD-Enabled Functions:*
Define funciones vectorizables cuando son llamadas desde un *Array Notation* o `#pragma simd loop`.
- `#pragma simd:`
Especifica que un ciclo será vectorizado.

Introduce tres nuevas palabras reservadas a C y C++:

- *cilk_spawn*: Especifica que una función puede ser ejecutada de forma asincrónica, sin esperar por el final para continuar con la función llamadora.
- *cilk_sync*: Especifica que todas las funciones ejecutadas asincrónicamente deben completar su ejecución antes de continuar. El mismo es implícito al final de cada función.
- *cilk_for*: Permite que ciclos sean ejecutados en paralelo.

cilk_spawn y *cilk_for* expresan la **oportunidad** de paralelismo.

Qué parte será ejecutada en paralelo es determinada por el runtime.

Utilizar locks para proteger variables compartidas puede ser problemático.

Su uso incorrecto lleva a *deadlocks* o incluso a afectar el rendimiento.

No se puede asegurar el orden, generando resultados no determinísticos.

Los *reducers* crear “vistas” de variables por medio de mecanismos de *lock-free*, integrados en cada etapa de sincronización.

La integración se da de forma ordenada, para mantener la semántica secuencial.

Permite expresar operaciones de alto nivel sobre arreglos.

Ayudan al compilador a aplicar vectorizaciones efectivas.

Las operaciones son aplicadas sobre elementos del arreglo en paralelo.

Además provee funciones especiales *builtin*.

`SIMD-enabled:`

Una función SIMD-enabled, puede ser llamada con argumentos escalares o un arreglo.

`#pragma simd:`

Permite que el compilador vectorice un ciclo.

Aplica vectorización de forma manual.

El estándar de Cilk Plus define las keywords:

`_Cilk_spawn`, `_Cilk_sync`, y `_Cilk_for`.

Respetan el estándar para extensiones sobre el lenguaje C/C++.

`<cilk/cilk.h>` define macros más simples:

`cilk_spawn`, `cilk_sync`, y `cilk_for`.

Función ejemplo para calcular el i-ésimo número de Fibonacci.

```
int fib(int n) {  
    if (n < 2)  
        return n;  
    int x = fib(n-1);  
    int y = fib(n-2);  
    return x + y;  
}
```

(No es la mejor implementación, pero provee un ejemplo recursivo simple)

Cilk Plus: Ejemplo básico

El cálculo de `fib(n-1)` puede ser ejecutado en paralelo junto con `fib(n-2)`.

Versión en *Cilk*:

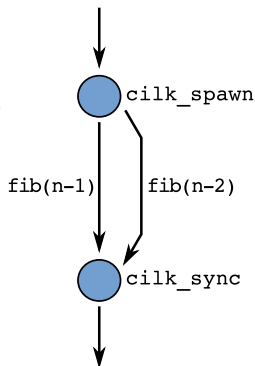
```
int fib(int n) {  
    if (n < 2)  
        return n;  
    int x = cilk_spawn fib(n-1);  
    int y = fib(n-2);  
    cilk_sync;  
    return x + y;  
}
```

- `cilk_spawn` especifica que la función puede ser ejecutada en paralelo.
- El código luego de un `cilk_spawn` corresponde a una **continuation**.
- `cilk_spawn` no crea threads, indica la oportunidad de paralelizar.
- `cilk_sync` indica que todas las funciones ejecutadas en paralelo deben terminar antes de continuar la ejecución (alcance limitado).

Un **strand** es una secuencia de instrucciones que comienza o termina en un cambio de paralelismo.

La función `fib()` contiene los siguientes **strand**.

- Desde el comienzo de la función hasta el `cilk_spawn`.
- El primer `cilk_spawn`, la llamada recursiva a `fib()`.
- Desde el `cilk_spawn` hasta el `cilk_sync`.
- Desde el `cilk_sync` hasta el final de la función.



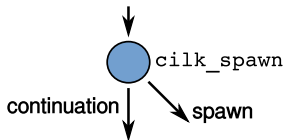
Cilk Plus: *strand*, hebra o filamento

El modelo de paralelismo fork/join, permite ver a una aplicación como un DAG (*Directed Acyclic Graph*).

Cada arco representa un *strand* y cada nodo una sentencia de cilk.

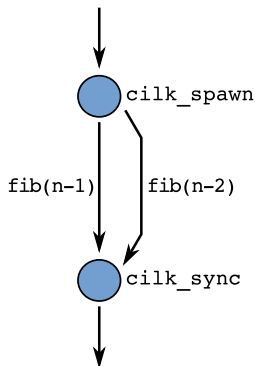
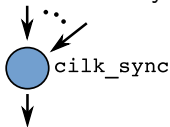
cilk_spawn

exactamente una entrada y dos salidas.



cilk_sync

dos o más entradas y exactamente una salida.



`cilk_spawn` y `cilk_sync` funcionan muy bien para expresar paralelismo.

Pero son ineficientes para paralelizar ciclos.

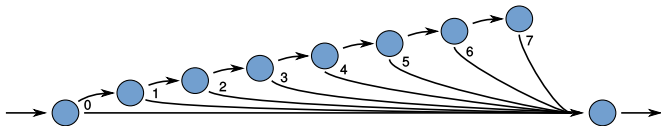
Supongamos el siguiente ejemplo,

```
for (int i = 0; i < 8; ++i)
{
    do_work(i);
}
```

Cilk Plus: cilk_for

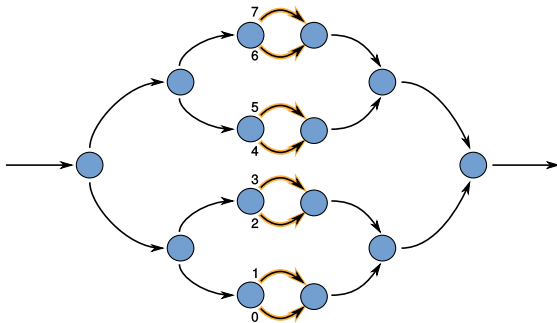
Si se lo expresa con `cilk_spawn` y `cilk_sync`.

```
for (int i = 0; i < 8; ++i)
{
    cilk_spawn do_work(i);
}
cilk_sync;
```



Ahora, si se utiliza cilk_for.

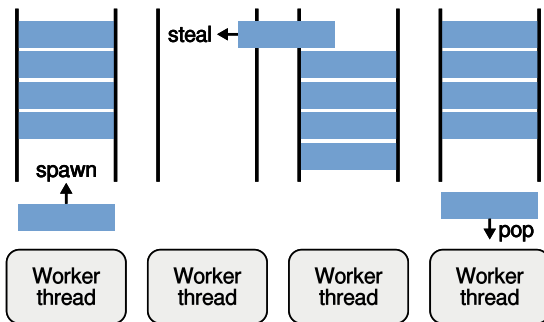
```
cilk_for (int i = 0; i < 8; ++i)
{
    do_work(i);
}
```



Cilk Plus: Scheduler work-stealing

Cada thread tiene su propia cola de tareas.

Si un cola se queda sin tareas, se selecciona aleatoriamente una cola donde “robar” tareas.



Permitir paralelismo es un aspecto clave de la semántica de Cilk.

Usar un solo *worker* generará las mismas operaciones que el caso secuencial.

La cantidad de trabajo suficiente: $tareas > cores \cdot 10$

Las aplicaciones no deben adaptarse al número de cores disponibles.

No se debe convertir todos los llamados a `cilk_spawn`, el costo de un `cilk_spawn` es 10 veces el de un `call`.

El costo estimado de *work-stealing* es de 15000 instrucciones.

Cilk Plus: Reducers

Al modificar una variable compartida se crea una condición de carrera.
Locks o *mutexes* permiten resolver estos problemas.

Considerar el código que genera una lista de letras de “a” hasta “z”

```
void locked_list_test()
{
    mutex m;
    std::list<char> letters;

    cilk_for(char ch = 'a'; ch <= 'z'; ch++)
    {
        work();
        m.lock();
        letters.push_back(ch);
        m.unlock();
    }
    PRINT(letters);
}
```

Las listas de STL no son *thread-safe*, entonces se deben usar *mutex*.
Garantizan acceso seguro, pero no orden.

Resultado: y g n d t a w x e z q j o h b u f v c k i r p l m s

Reescribiendo el código usando *reducers* (reducer_list)

```
void reducer_list_test()
{
    cilk::reducer< cilk::op_list_append<char> > letters_reducer;

    cilk_for(char ch = 'a'; ch <= 'z'; ch++)
    {
        work();
        letters_reducer->push_back(ch);
    }

    const std::list<char> &letters = letters_reducer.get_value();

    PRINT(letters);
}
```

Los *reducer* garantizan acceso seguro y orden, sin importar la cantidad de *workers*.

Resultado: a b c d e f g h i j k l m n o p q r s t u v w x y z

Cilk Plus: Reducers

Propiedades de los reducers:

- No requieren de *mutexes*, cada *strand* tiene su propia vista del *reducer*.
- Las vistas son combinadas por el runtime de Cilk.
- La reducción se realiza en el orden del programa secuencial.

Los reducers no están limitados a ciclos.

Para el ejemplo de Fibonacci.

```
cilk::reducer< cilk::op_add<int> > fib_sum(0);
```

```
void fib_with_reducer_internal(int n)
{
    if (n < 2) {
        *fib_sum += n;
    }
    else {
        cilk_spawn fib_with_reducer_internal(n-1);
        fib_with_reducer_internal(n-2);
        cilk_sync;
    }
}
```

Cilk Plus: Reducer Library

Cilk provee una biblioteca de *reducers* y además es posible desarrollar nuevos.

Lists	
<code>reducer_list_append</code>	Crea una lista agregando elementos por detrás.
<code>reducer_list_prepend</code>	Crea una lista agregando elementos por delante.
Min/Max	
<code>reducer_max</code>	Calcula el máximo valor.
<code>reducer_max_index</code>	Calcula el máximo valor y su índice.
<code>reducer_min</code>	Calcula el mínimo valor.
<code>reducer_min_index</code>	Calcula el mínimo valor y su índice.
Math Operators	
<code>reducer_opadd</code>	Calcula la suma de los valores.
Bitwise Operators	
<code>reducer_opand</code>	Calcula el AND binario.
<code>reducer_opor</code>	Calcula el OR binario.
<code>reducer_opxor</code>	Calcula el XOR binario.
String Operators	
<code>reducer_string</code>	Acumula una string usando la operación concatenación.
<code>reducer_wstring</code>	Acumula una wstring usando la operación concatenación.
Files	
<code>reducer_ostream</code>	<i>output stream</i> para escribir en paralelo.

Cilk Plus: Array Notation

Cilk incluye extensiones para operaciones con arreglos. Permiten expresar operaciones vectoriales de alto nivel, paralelizables sobre arreglos.

- Ayudan al compilador en una vectorización eficiente.
- Se puede usar en arreglos estáticos y dinámicos.
- Se puede usar dentro de sentencias “if” o “switch”.
- Semántica paralela, no implican orden.

El nuevo operador es [:]

`array-expression`[`lower-bound` : `length` : `stride`]

- `array-expression`: arreglo
- `lower-bound`: límite inferior
- `length`: tamaño
- `stride`: paso

- Los tres valores en la notación son expresiones enteras.
- El usuario es responsable de mantener las secciones dentro de los límites del arreglo.
- Los argumentos pueden ser omitidos.
 - `lower-bound` default es 0.
 - `length` default es la longitud del arreglo.
 - `stride` default es 1 (se puede omitir el ":").
- `[:]` denota todo el arreglo.
- Se pueden usar arreglos de múltiples dimensiones (`[:] [:]`).

Cilk Plus: Array Notation (Ejemplos)

Caso	Array Notation	Equivalente C/C++
Setea todos los elementos de A en 5.	<code>A[:] = 5;</code>	<pre>for (i = 0; i < 10; i++) A[i] = 5;</pre>
Setea los primeros 7 elementos de A en 5 y los últimos 3 en 4.	<code>A[0:7] = 5;</code> <code>A[7:3] = 4;</code>	<pre>for (i = 0; i < 7; i++) A[i] = 5; for (i = 7; i < (7+3); i++) A[i] = 4;</pre>
Setea los elementos en índices pares en 5 y los en índices impares en 4.	<code>A[0:5:2] = 5;</code> <code>A[1:5:2] = 4;</code>	<pre>for (i = 0; i < 10; i += 2) A[i] = 5; for (i = 1; i < 10; i += 2) A[i] = 4;</pre>

A, B y D arreglos enteros de tamaño 10.
C es un arreglo de dos dimensiones de tamaño 10x10.

Cilk Plus: Array Notation (Ejemplos)

Caso	Array Notation	Equivalente C/C++
Setea cada elemento de arreglo A con el correspondiente en el arreglo B.	$A[:] = B[:];$	<pre>for (i = 0; i < 10; i++) A[i] = B[i];</pre>
Setea cada elemento de arreglo A con el correspondiente en el arreglo B y le suma 5 a cada uno.	$A[:] = B[:] + 5;$	<pre>for (i = 0; i < 10; i++) A[i] = B[i] + 5;</pre>
Suma cada elemento de A con B y guarda el resultado en D.	$D[:] = A[:] + B[:];$	<pre>for (i = 0; i < 10; i++) D[i] = A[i] + B[i];</pre>
Setea los primeros n elementos del arreglo A en 5.	$A[0:n] = 5;$	<pre>for (i = 0; i < n; i++) A[i] = 5;</pre>

A, B y D arreglos enteros de tamaño 10.
C es un arreglo de dos dimensiones de tamaño 10x10.

Cilk Plus: Array Notation (Ejemplos)

Caso	Array Notation	Equivalente C/C++
Setea todos los elementos del arreglo bidimensional C en 12.	<code>C[:, :] = 12;</code>	<pre>for (i = 0; i < 10; i++) for (j = 0; j < 10; j++) C[i][j] = 12;</pre>
Setea todos los elementos en filas pares del arreglo C en 12.	<code>C[0:5:2][:] = 12;</code>	<pre>for (i = 0; i < 10; i += 2) for (j = 0; j < 10; j++) C[i][j] = 12;</pre>
Setea los elementos de la fila X en C con los correspondientes de A.	<code>C[X][:] = A[:];</code>	<pre>for (j = 0; j < 10; j++) C[X][j] = A[j];</pre>
Llama a la función <code>func()</code> para todos los elementos del arreglo A.	<code>func (A[:]);</code>	<pre>for (i = 0; i < 10; i++) func (A[i]);</pre>

A, B y D arreglos enteros de tamaño 10.
C es un arreglo de dos dimensiones de tamaño 10x10.

Los arreglos generados dinámicamente pueden usar la notación de arreglos con una excepción:

- Para acceder a los elementos se debe especificar el **índice** inicial y el **tamaño**. No se puede usar la notación `A[:]`.

Para los arreglos de múltiples dimensiones, se debe conocer la longitud de todos los arreglos involucrados, con excepción del más externo.

Cilk Plus: Array Notation en condiciones

La notación de arreglos se puede usar dentro de condiciones.

Ejemplos:

Caso	Array Notation	Equivalente C/C++
Para cada elemento de A, guarda "Yes" si el valor es 5, y "No" en caso contrario.	<pre>if (5 == A[:]) r[:] = "Yes"; else r[:] = "No";</pre>	<pre>for (int i=0; i<size; i++){ if (5 == A[i]) r[i] = "Yes"; else r[i] = "No"; }</pre>
Determina el tipo de cada letra en A.	<pre>if (('a'==A[:]) ('e'==A[:]) ('i'==A[:]) ('o'==A[:]) ('u'==A[:])) types[:] = 'V'; else types[:] = 'C';</pre>	<pre>for (i = 0; i < 10; i++){ if (('a'==A[i]) ('e'==A[i]) ('i'==A[i]) ('o'==A[i]) ('u'==A[i])) types[i] = 'V'; else types[i] = 'C'; }</pre>

Una función *SIMD-enabled* puede ser llamada con argumentos escalares o con arreglos de elementos en paralelo.

Se especifica anotando al comienzo de la aridad de la función `__declspec(vector)` o `__attribute__((vector))`.

El compilador crea dos versiones de la misma función, una escalar y otra vectorial.

Ejemplo:

```
__declspec(vector) double func(double a, double b);
```

El uso de `#pragma simd`, obliga al compilador a utilizar instrucciones de SIMD.

No deja decidir al compilador si vectorizar automáticamente.
(hint: `#pragma ivdep`)

Ejemplo:

```
void add_floats(float *a, float *b, float *c,  
               float *d, float *e, int n) {  
    int i;  
    #pragma simd  
    for (i=0; i<n; i++) {  
        a[i] = a[i] + b[i] + c[i] + d[i] + e[i];  
    }  
}
```

- 1 Construir una función que imprima “cuatro”, 4 veces en paralelo, luego “tres”, 3 veces en paralelo y así sucesivamente.
- 2 Con notación de arreglos, construir dos arreglo A y B de N elementos. Inicializarlos con 5 y 7 respectivamente. Crear un arreglo C de $2*N$ elementos, la primera mitad completarla con los valores de A y B intercalados en índices pares e impares. La segunda mitad con la suma de los valores de A y B.
- 3 Construir una función con `cilk_for` que obtenga la cantidad de números primos menores a 10000000. Considerar usar `cilk::reducer_opadd<int>`.

Recreo...



- Biblioteca basada en templates de C++
- Desarrollada por Intel
- Versión 1.0 publicada el 29 de agosto de 2006
- Licencia doble: Comercial / Código abierto (Apache 2.0)

TBB: Threading Building Blocks

Biblioteca que provee soluciones para facilitar la escritura de programas paralelos.

Hace foco en un buen rendimiento, gran escalabilidad y fácil utilización.

Incluye:

- Generic Parallel Algorithms
- Concurrent Containers
- Flow Graph
- Scalable Memory Allocation
- Thread Local Storage
- Task Scheduler
- Synchronization Primitives

TBB: Generic Parallel Algorithms

TBB mapea patrones de paralelismo con algoritmos.

- `parallel_for`: map
- `parallel_reduce`, `parallel_scan`: reduce, scan
- `parallel_do`: workpile
- `parallel_pipeline`: pipeline
- `parallel_invoke`, `task_group`: fork-join
- `flow_graph`: plumbing for reactive and streaming apps

TBB: Generic Parallel Algorithms

Considerando el ejemplo de patrón map.

Se aplica la función Foo a cada elemento del arreglo A.

```
void SerialApplyFoo( float A[], size_t n ) {  
    for( size_t i=0; i!=n; ++i )  
        Foo(A[i]);  
}
```

`parallel_for` puede ser usado para implementar la misma funcionalidad, pero de forma paralela.

Divide las iteraciones del ciclo en tareas las ejecuta en paralelo

```
#include "tbb/tbb.h"  
using namespace tbb;  
  
void ParallelApplyFoo( float A[], size_t n ) {  
    tbb::parallel_for( size_t(0), n,  
        [&]( size_t i ) {  
            Foo(A[i]);  
        } );  
}
```

TBB provee contenedores altamente concurrentes. Pueden ser usados con algoritmos de TBB o de forma independiente.

Un contenedor concurrente permite que múltiples threads puedan acceder y modificar elementos.

Dos características:

- *Fine-grained locking*: Se generan locks solamente en las partes del contenedor realmente necesarias por el thread.
- *Lock-free techniques*: Diferentes threads consideran el efecto de otros threads para no interferir.

TBB: Concurrent Containers

Considerar el ejemplo de una cola (queue).

Inicialmente chequea si la cola esta vacía, y quita un elemento para procesar.

Si se ejecutará en paralelo, otro thread puede quitar el último elemento después del chequeo.

```
extern std::queue<T> MySerialQueue;  
T item;  
if( !MySerialQueue.empty() ) {  
    item = MySerialQueue.front();  
    MySerialQueue.pop_front();  
    ... process item...  
}
```

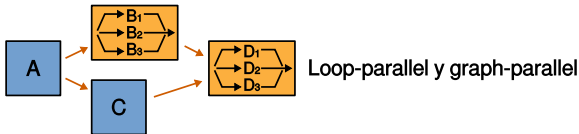
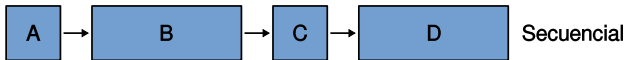
La interfaz del contenedor de TBB provee una función `try_pop`.

Quita el elemento chequeando si la cola está vacía de forma atómica.

```
extern concurrent_queue<T> MyQueue;  
T item;  
if( MyQueue.try_pop(item) ) {  
    ... process item...  
}
```

TBB: Flow Graph

TBB provee una interfaz para definir un grafo de dependencias, permitiendo expresar mayor nivel de paralelismo.



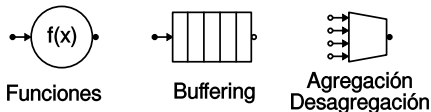
TBB: Flow Graph

En un grafo los cálculos son representados por nodos y la comunicación o dependencias por ejes.

El usuario es responsable de definir las dependencias que deben ser respetadas.

El scheduler de TBB analiza el paralelismo sobre la topología del grafo.

Distintos tipos de nodos son soportados.



El usuario conecta instancias de nodos entre sí y especifica sus dependencias.

TBB: Flow Graph

```
#include "tbb/flow_graph.h"
#include <iostream>

using namespace std;
using namespace tbb::flow;

int main() {
    graph g;
    continue_node< continue_msg> hello( g,
        []( const continue_msg &) {
            cout << "Hello";
        }
    );
    continue_node< continue_msg> world( g,
        []( const continue_msg &) {
            cout << "\nWorld\n";
        }
    );
    make_edge(hello, world);
    hello.try_put(continue_msg());
    g.wait_for_all();
    return 0;
}
```

Ejemplo “Hello World” de grafos en TBB.
No contiene paralelismo.
Crea un nodo que muestra la palabra “Hello”
y un segundo nodo que muestra la palabra
“world”.

TBB: Scalable Memory Allocator

Utilizar *memory allocator* diseñados para el acceso secuencial en múltiples threads presenta problemas.

TBB provee dos *memory allocator templates* similares a `class std::allocator`.

- `scalable_allocator<T>`:
Soluciona el acceso concurrente y escalabilidad.
- `cache_aligned_allocator<T>`:
Soluciona problemas de *false sharing*, dos objetos no pueden compartir la misma línea de cache.

Por ejemplo, utilizar el `cache_aligned_allocator` para `std::vector`

```
std::vector<int, cache_aligned_allocator<int> >;
```

TBB provee dos clases templates para almacenamiento local en threads.

Ambos permiten acceso a objetos locales por threads y crear objetos sobre demanda.

Difieren en el modelo de funcionamiento:

- Clase `combinable`:
provee almacenamiento local por thread para mantener los sub-cálculos que serán reducidos luego como resultado.
- Clase `enumerable_thread_specific`:
provee almacenamiento local por thread que actúa como un contenedor STL, uno por thread.

TBB: Thread Local Storage

```
typedef enumerable_thread_specific< std::pair<int,int> > CounterT;
CounterT MyCounters (std::make_pair(0,0));

struct Body {
    void operator()(const tbb::blocked_range<int> &r) const {
        CounterT::reference my_counter = MyCounters.local();
        ++my_counter.first;
        for (int i = r.begin(); i != r.end(); ++i)
            ++my_counter.second;
    }
};

int main() {
    parallel_for( blocked_range<int>(0, 100000000), Body());

    for (CounterT::const_iterator i = MyCounters.begin();
         i != MyCounters.end(); ++i)
    {
        printf("Thread stats:\n");
        printf("calls to operator(): %d", i->first);
        printf("total # of iterations: %d\n\n", i->second);
    }
}
```

TBB: Task Based Programming

Provee dos interfaces:

- high-level interface: Clase `task_group`
Permite crear grupos de tareas potencialmente paralelas desde *functores* o expresiones *lambda*.
- low-level interface: Clase `task`
Permite un control más detallado de tareas, como propagación de excepciones o afinidad de procesadores.

Ejemplo del uso de `task_group` para Fibonacci.

```
#include "tbb/task_group.h"
using namespace tbb;
int Fib(int n) {
    if( n<2 ) {
        return n;
    } else {
        int x, y;
        task_group g;
        g.run( [&]{ x=Fib(n-1); } ); // spawn a task
        g.run( [&]{ y=Fib(n-2); } ); // spawn another task
        g.wait();                    // wait for both tasks
        return x+y;
    }
}
```

TBB provee primitivas de sincronización.

- Exclusión mútua, permite controlar cuantos threads pueden acceder simultaneamente a una región.
- Operaciones atómicas, los threads ven operaciones como si sucedieran instantaneamente.

TBB: Synchronization Primitives

```
Node* FreeList;
typedef tbb::spin_mutex FreeListMutexType;
FreeListMutexType FreeListMutex;

Node* AllocateNode() {
    Node* n;
    {
        FreeListMutexType::scoped_lock lock(FreeListMutex);
        n = FreeList;
        if( n )
            FreeList = n->next;
    }
    if( !n )
        n = new Node();
    return n;
}

void FreeNode( Node* n ) {
    FreeListMutexType::scoped_lock lock(FreeListMutex);
    n->next = FreeList;
    FreeList = n;
}
```


TBB vs OpenMP vs Threads

	TBB	OpenMP	Threads
Task level parallelism	+	+	-
Data decomposition support	+	+	-
Complex parallel patterns (non-loops)	+	-	-
Broadly applicable generic parallel patterns	+	-	-
Scalable nested parallelism support	+	-	-
Built-in load balancing	+	+	-
Affinity support	-	+	+
Static scheduling	-	+	-
Concurrent data structures	+	-	-
Scalable memory allocator	+	-	-
I/O dominated tasks	-	-	+
User-level synchronization primitives	+	+	-
Compiler support is not required	+	-	+
Cross OS support	+	+	-

Fuente: <https://software.intel.com/en-us/intel-threading-building-blocks-openmp-or-native-threads>

- 1 Construir una función para obtener el i -ésimo número de Fibonacci utilizando `task_group`.
- 2 Modificando el ejemplo de `flow_graph`, construir tres nodos conectados por ejes de continuación que impriman cada uno las letras HPC.
- 3 Modificar la función `haceTiempo` y explicar los resultados generados por el código.

Material utilizado en la clase:

- Tutorial de Cilk:
<https://www.cilkplus.org/cilk-plus-tutorial>
- Tutorial de TBB:
<https://www.threadingbuildingblocks.org/intel-tbb-tutorial>
- Tabla comparación:
<https://software.intel.com/en-us/intel-threading-building-blocks-openmp-or-native-threads>

Libros utiles:

- *Intel Threading Building Blocks (Outfitting C++ for Multi-Core Processor Parallelism)*. **James Reinders**. O'Reilly Media.
- *Structured Parallel Programming (Patterns for Efficient Computation)*. **Michael McCool, Arch D. Robison y James Reinders**. Morgan Kaufmann

Preguntas...

Gracias...