Proyecto de programación paralela

J. Daniel Garcia (coordinador)
Arquitectura de Computadores
Departamento de Informática
Universidad Carlos III de Madrid

2025

1. Objetivo

El objetivo de este proyecto es enfrentarse a la conversión de un programa secuencial a su correspondiente versión paralela. De esta manera los estudiantes podrán explorar alternativas de mejora de prestaciones usando técnicas de programación que permitan aprovechar los recursos de un multiprocesador de memoria compartida.

2. Visión general

En este proyecto partirás del proyecto realizado en la primera parte de la asignatura y construirás una versión paralela con el objetivo de mejorar el rendimiento para ejecutarlo en una máquina de memoria compartida con un elevado número de unidades de procesamiento.

Para ello, y partiendo de la versión anterior de tu proyecto, tendrás que acometer una serie de tareas:

1. Preparar el entorno de desarrollo para el uso de la biblioteca TBB.
2. Adaptar la generación de números aleatorios para el uso en un programa paralelo.
3. Identificar las zonas de código donde es más adecuado aplicar la programación paralela.
4. Identificar los parámetros de la paralelización que tienen impacto sobre el rendimiento global.
5. Evaluar el impacto de cada uno de estos parámetros sobre el rendimiento de la aplicación y seleccionar los más adecuados.
6. Generar la versión final del programa.

2.1. Preparación del entorno

2.1.1. Modificación de la imagen Docker

Para preparar el entorno de desarrollo para el uso de la biblioteca TBB deberás realizar modificaciones que afectan a la imagen de **Docker**.

Para ello debes editar el archivo **.devcontainer/Dockerfile** y añadir a la lista de paquetes a instalar, el paquete **libtbb-dev** como se muestra en el listado 1.

Listado 1: Modificación de archivo Dockerfile

```
...
# Install all necessary packages in a single RUN command to reduce image layers
RUN apt-get update && apt-get install -y \
    software-properties-common \
    wget \
    curl \
    gnupg \
    build-essential \
    libtbb-dev \
    git \
    sudo \
    ninja-build \
    python3 \
    python3-pip \
    python3-full \
    pipx \
    lsb-release \
    # The following tools are often included in build-essential, but it is good to be explicit
    gdb \
    valgrind \
    cppcheck \
    doxygen \
    graphviz \
    ccache \
    pkg-config \
    make \
    # Code coverage tools
    lcov \
    && rm -rf /var/lib/apt/lists/*
...
```

2.1.2. Utilización de TBB en CMake

Para asegurar que tienes disponible la biblioteca TBB en la compilación con CMake debes incluir el paquete correspondiente. Para ello en el archivo **CMakeLists.txt** principal debes incluir una línea con el correspondiente **find_package** como se indica en el listado 2.

Listado 2: Inclusión del paquete TBB en CMakeLists.txt

```
...
FetchContent_MakeAvailable(GSL googletest)

find_package(TBB REQUIRED)

# Enable testing
enable_testing()
...
```

Para hacer que se encuentren los archivos de cabecera de TBB y para conseguir que se enlace con la biblioteca, debes establecer una dependencia con esta biblioteca. La forma más sencilla es establecer una dependencia con la biblioteca **common**, que se aplicará transitivamente a todos los ejecutables que dependan de dicha biblioteca.

En el archivo **CMakeLists.txt** del directorio **common** modifica la lista de bibliotecas de las que depende **common** y añade TBB tal y como se indica en el listado 3.

Listado 3: Dependencia de la biblioteca TBB.

```
...
target_link_libraries(common PUBLIC Microsoft.GSL::GSL TBB::tbb)
```

2.2. Generación paralela de números aleatorios

2.2.1. Generadores compartidos

Hay varias razones por las que un generador de números aleatorio plantea problemas en un modelo de programación paralelo.

Por una parte, si distintos hilos de ejecución acceden de forma no sincronizada a un objeto compartido se produce una carrera de datos, por lo que el programa pasa a tener un comportamiento no definido.

Para evitar este problema, puedes proteger los accesos al generador de números aleatorios con una mecanismo de sincronización (como un **mutex** o un **semáforo**). Sin embargo, si los accesos al objeto compartido son muy frecuentes, y la longitud de la sección crítica es corta puede ser que esta estrategia provoque mucha contención y se limite el paralelismo.

2.2.2. Privatización de objetos compartidos

Otra posible estrategia es que cada hilo disponga de su propio generador de números aleatorios local distinto. Este evita los problemas de tener un objeto compartido. Cada hilo de ejecución accede a su propia copia de la objeto sin necesidad de sincronización y sin problemas de carreras de datos.

En el caso de los generadores de números aleatorios un problema lo constituyen las semillas. Si todos los generadores de números aleatorios comparten la misma semilla se generaría exactamente la misma secuencia en cada hilo, existiendo entonces una alta correlación entre las secuencias generadas en cada hilo.

Una solución a este problema es la utilización de un generador de números aleatorios basado en una semilla, para generar las semillas de los distintos hilos.

2.2.3. Almacenamiento local al hilo en C++

En C++ una variable declarada como **thread_local** tiene almacenamiento local al hilo. Esto quiere decir que existe una copia de esta variable en cada hilo de ejecución.

El listado 4 muestra como se puede tener un identificador distinto y consecutivo en cada hilo de ejecución.

Listado 4: Identificadores de hilos consecutivos C++23.

```
1 std::size_t my_id() {
2   static std::atomic<std::size_t> counter{0};
3   thread_local id = counter++;
4   return id;
5 }
```

En este ejemplo, la variable **counter** es una variable atómica (ver https://en.cppreference.com/w/cpp/atomic/atomic.html) y estática. Es decir, es una variable de la que existe una única copia global compartida por todos los hilos. La inicialización a **0** solamente ocurre la primera vez que se entra en la función **my_id()**. En las siguientes invocaciones a la función **my_id()** la variable **recuerda** el valor que tenía al finalizar al ejecución anterior de la variable.

Por otra parte, la variable **id** está marcada con el calificador **thread_local**, por lo que se genera una copia de esta variable por cada hilo de ejecución. Cada copia de la variable **id** se inicializa la primera vez que se entra en la función **my_id()** desde cada hilo de ejecución.

2.2.4. Almacenamiento local al hilo en TBB

La biblioteca TBB ofrece un mecanismo de gestión de variables con almacenamiento local al hilo a través de el tipo **tbb::enumerable_thread_specific<T>**, (ver https://oneapi-spec.uxlfoundation.org/

specifications/oneapi/v1.2-rev-1/elements/onetbb/source/thread_local_storage/enumerable_thread_specific_cls) que se inicializa con una expresión lambda.

El listado 5 presenta un ejemplo de uso de almacenamiento local al hilo con TBB. La función **f()** tiene dos partes: la inicialización de variables locales al hilo y el acceso a dichas variables.

Listado 5: Identificadores de hilos consecutivos TBB.

```
1  void f() {
2    tbb::enumerable_thread_specific<std::size_t> id{[] {
3      static std::atomic_<std::size_t> counter{0};
4      auto next_id = counter++;
5      return next_id;
6    }};
7
8    tbb::parallel_for(std::size_t{0}, std::size_t{1000}, [&]{
9      auto value_id = id.local();
10   });
11 }
```

En la primera parte se inicializa una variable del tipo especificado por TBB para objetos con almacenamiento local al hilo, que usa como tipo base **std::size_t**. En este caso, **id** se inicializa con una expresión lambda. Cada vez que se necesite una copia de **id** para un hilo determinado se invocará a esta expresión lambda. Dentro se utiliza un contador atómico y estático para generar identificadores secuenciales. Cada vez que se invoca a la expresión lambda se genera un identificador numérico consecutivo.

Dentro de un bloque paralelo de TBB (como por ejemplo, **tbb::parallel_for()**) se puede acceder la correspondiente variable local al hilo invocando a la función miembro **local()** de **tbb::enumerable_thread_specific**.

2.2.5. Generadores aleatorios locales al hilo

Si se desea tener varios generadores de números aleatorios a partir de una única semilla se pueden conseguir en varios pasos:

1. Determinar el número de hilos que se va a utilizar.
2. Generar un vector de semillas.
3. Iniciar los generadores locales a cada hilo.

2.2.6. Determinación del número de hilos

La biblioteca estándar de C++23 ofrece un mecanismo general para determinar el número de hilos disponibles es la función **std::hardware_concurrency()** (ver https://en.cppreference.com/w/cpp/thread/thread/hardware_concurrency.html) que devuelve el número de hilos soportado por la implementación. Si no se puede determinar el valor podría devolver **0**.

La biblioteca TBB ofrece un mecanismo para obtener el nivel de concurrencia mediante **tbb::this_task_arena::max_concurrency()** (ver https://oneapi-spec.uxlfoundation.org/specifications/oneapi/v1.2-rev-1/elements/onetbb/source/task_scheduler/task_arena/task_arena_cls?highlight=max_concurrency#_CPPv4NK15max_concurrencyEv).

2.2.7. Generación de un vector de semillas

Una vez conozcas el número de hilos que se pueden ejecutar, puedes generar un vector con varias semillas (una para cada hilo) a partir de una semilla inicial.

Por ejemplo si tienes la semilla inicial **s** y quieres generar **32** semillas en el vector **v**, puede hacer algo como lo que se muestra en el listado 6.

Listado 6: Inicialización de un vector de semillas

```
1 void f(std::size_t s) {
2   std::vector<std::uint64_t> v(32);
3   std::mt19934_64 seed_gen{s};
4   std::ranges::generate(v, seed_gen);
5   //...
6 }
```

La función **f()** define un vector **v** en el que cada elemento es un entero sin signo de 64 bits (tipo **std::uint64_t**). El vector se define con 32 posiciones.

A continuación, se inicializa la variable **seed_gen** (que es un generador de números aleatorios) con el valor de semilla **s**. Este generador se utilizará para generar las distintas semillas que se van a almacenar en el vector **v**.

Por último, la función **std::ranges::generate()** (ver `https://en.cppreference.com/w/cpp/algorithm/ranges/generate`) inicializa cada posición del vector **v** con el resultado de invocar al generador **seed_gen()**.

2.2.8. Iniciación de generadores locales al hilo

Una vez se tienen semillas independientes para cada hilo se pueden iniciar los generadores de números aleatorios usados en cada hilo. Para ello, las correspondientes variables de tipo **std::mt19934_64** deben utilizar almacenamiento local al hilo.

2.3. Ejecución paralela con TBB

Al considerar la versión paralela de un programa con TBB deben considerarse varias dimensiones:

1. Algoritmos paralelos: Deben seleccionarse los algoritmos paralelos que se van a utilizar para mejorar el rendimiento de la aplicación.

2. Limitación del nivel de concurrencia: Se puede limitar el número máximo de hilos que se utilizarán. Esto será útil para analizar el impacto de utilizar más o menos hilos en el programa.

3. Estrategia de división del trabajo: TBB ofrece diversas estrategias de división del trabajo. Esto será útil para estudiar cuál es la estrategia más adecuada para cada programa.

4. Tamaño del grano: Se puede definir un umbral para limitar la división del trabajo de elementos más pequeños.

2.3.1. Algoritmos paralelos en TBB

La biblioteca TBB ofrece distintos algoritmos paralelos (puedes ver la lista completa en `https://oneapi-spec.uxlfoundation.org/specifications/oneapi/latest/elements/onetbb/source/algorithms`).

Entre ellos pueden ser de interés los siguientes:

- **tbb::parallel_invoke()**: Evalúa varias funciones en paralelo.
- **tbb::parallel_for()**: Realiza una iteración en paralelo sobre un rango de valores.
- **tbb::parallel_reduce()**: Calcula una operación de reducción sobre un rango.
- **tbb::parallel_for_each()**: Realiza una iteración en paralelo sobre un par de iteradores o sobre un contenedor.

También existen otros algoritmos como **tbb::parallel_deterministic_reduce()**, **tbb::parallel_scan()**, **tbb::parallel_sort()** o **tbb::parallel_pipeline()**.

2.3.2. Limitación del nivel de concurrencia

La biblioteca TBB permite establecer una limitación sobre el número máximo de hilos subyacentes.

Se puede limitar el número máximo de hilos definiendo un objeto del tipo **tbb::global_control** (puedes ver los detalles en https://oneapi-spec.uxlfoundation.org/specifications/oneapi/latest/elements/onetbb/source/task_scheduler/scheduling_controls/global_control_cls). El listado 7 muestra un ejemplo de limitación global del número de hilos. En este ejemplo, se limita el número de hilos para TBB a **8** durante toda la vida de la función **main()**.

Listado 7: Limitación del número de hilos con un objeto de control global.

```
1 #include <tbb/global_control.h>
2
3 int main() {
4   tbb::global_control limit_threads{
5         tbb::global_control::max_allowed_parallelism, 8
6   };
7
8   f();
9 }
```

2.3.3. Estrategias de división del trabajo

La biblioteca TBB ofrece varias estrategias de división del trabajo, que pueden utilizarse con los algoritmos paralelos. En el proyecto se van a evaluar tres de ellas: **simple_partitioner**, **static_partitioner** y **auto_partitioner**.

simple_partitioner Especifica que el bucle paralelo debe dividir de forma recursiva su rango hasta que no pueda dividirse más. Puedes encontrar más información en https://oneapi-spec.uxlfoundation.org/specifications/oneapi/latest/elements/onetbb/source/algorithms/partitioners/simple_partitioner.

static_partitioner Especifica que el bucle paralelo debe distribuir el trabajo uniformemente entre los hilos sin realizar equilibrio de carga adicional. El número de subrangos generados es igual al número de hilos que participan en la ejecución de la tarea. Puedes encontrar más información en https://oneapi-spec.uxlfoundation.org/specifications/oneapi/latest/elements/onetbb/source/algorithms/partitioners/static_partitioner

auto_partitioner Especifica que el bucle paralelo debe optimizar la subdivisión de rangos para el robo de trabajos. En este caso se intenta minimizar el número de divisiones de rangos al mismo tiempo que se ofrece oportunidades para el robo de trabajo. Puedes encontrar más información en https://oneapi-spec.uxlfoundation.org/specifications/oneapi/latest/elements/onetbb/source/algorithms/partitioners/auto_partitioner.

2.3.4. Rangos y tamaño de grano

Existen distintos tipos para representar un rango de iteración para rangos de una dimensión (**tbb::blocked_range<T>**), de dos dimensiones (**tbb::blocked_range2d<T>**), de tres dimensiones (**tbb::blocked_range3d<T>**) o de N dimensiones (**tbb::blocked_nd_range<T>**).

Todos estos tipos pueden representar cada dimension por dos valores (inicio y fin) o por tres valores. En este último caso el tercer valor es el tamaño de grano. Es decir, el tamaño a partir del cual el subrango no debería dividirse más. No obstante, la interpretación de este valor depende de cada estrategia de partición.

3. Tareas

3.1. Desarrollo de aplicación paralela

Se desarrollará una única aplicación con el nombre **render-par**. En esta aplicación deberás elegir entre una de las dos estrategias ya implementadas en el primer proyecto: la estrategia AOS (*array of structures*) o SOA (*structure of arrays*). Deberás identificar cuál de las dos estrategias es la que mejor rendimiento puede ofrecer.

Esta aplicación deberá ser una versión paralela utilizando la biblioteca TBB. Deberás seleccionar los parámetros más adecuados para reducir al máximo posible el tiempo de ejecución.

3.1.1. Estructura del proyecto

El proyecto partirá como versión inicial del código que has entregado en el proyecto número 1. Esta será la primera versión que deberás subir al repositorio de GitHub para este proyecto.

El repositorio deberá contener los siguientes directorios:

- **.devcontainer**: Deberá de modificarse el archivo **Dockerfile** para incluir en la imagen la instalación del paquete **libtbb-dev**.
- El directorio **common** deberá contener inicialmente los mismos archivos que en la versión secuencial ya entregada. Aunque posteriormente se podrán realizar las modificaciones pertinentes para la versión paralela.
- El directorio **utcommon** seguirá conteniendo las pruebas unitarias para la biblioteca **common**.
- Los directorios **aos** y **soa** se remplazarán por un único directorio **par** con la versión paralela del programa. El ejecutable se llamará **render-par**.
- El software deberá seguir cumpliendo con todas las reglas de calidad establecidas en el primer proyecto de la asignatura.

3.1.2. Selección de configuración óptima

En el programa entregado deberás seleccionar los mejores valores de diversos parámetros para que la ejecución de tu programa sea óptima.

Debes considerar los siguientes parámetros:

- **Número de hilos**: La biblioteca TBB selecciona automáticamente un cierto número de hilos a partir de las características del hardware en el que se ejecuta. No obstante deberás decidir si es conveniente dejar esta decisión a la biblioteca o si es más adecuado fijar el número hilos a algún valor.
- **Estrategia de división del trabajo**: Deberás decidir entre las tres estrategias de división del trabajo descritas anteriormente (**simple_partitioner**, **static_partitioner** y **auto_partitioner**).
- **Tamaño de grano**: En cada algoritmo que lo admita deberás seleccionar, si es pertinente, el tamaño de grano más adecuado. Ten en cuenta que el caso de rangos multidimensionales se puede seleccionar un tamaño de grano distinto para cada dimensión.

3.2. Evaluación del rendimiento y la energía

Esta tarea consiste en una evaluación detallada del rendimiento y la energía y la justificación de la selección de la configuración óptima que se entrega.

Para cada una de las tres estrategias de división del trabajo, se realizará una evaluación detallada con distinto número de hilos fijado evaluando con número de hilos igual a todas las potencias de 2 hasta 256. Es decir: 1, 2, 4, 8, 16, ... 256.

En cada caso se realizarán evaluaciones con distinto tamaño de grano para determinar si el tamaño de grano es relevante y seleccionar el valor óptimo.

Con toda esta información se justificará la selección óptima y se derivarán conclusiones de los resultados obtenidos con distintas configuraciones.

Todas las evaluación se realizarán en el clúster **avignon** utilizando la cola de procesamiento **stan**.

Recuerda que la memoria del proyecto debe incluir conclusiones de los resultados obtenidos que deben ir más allá de una mera descripción, tratando de encontrar explicaciones convincentes de los resultados obtenidos.

4. Estructura de la memoria a entregar

Deberá redactarse una memoria del trabajo realizado que deberá entregarse en formato PDF. Deberá contener las siguientes secciones:

- **Página de título**: contendrá los siguientes datos.
  - Nombre del proyecto.
  - Número de equipo asignado.
  - Nombre, NIA y grupo reducido de los autores.

  Longitud máxima: 1 página.
- **Modificaciones de diseño**: Deberán explicarse las modificaciones al diseño general que se hayan realizado, identificando los archivos fuente afectados por las modificaciones.

  Longitud máxima: 3 páginas.
- **Paralelización**: Se incluirá una explicación de las modificaciones realizadas en el código para conseguir una ejecución paralela.

  Longitud máxima: 3 páginas.

- **Evaluación del rendimiento y la energía**: Deberá incluirse las evaluaciones de rendimiento y energía llevadas a cabo y la justificación de los parámetros óptimos seleccionados.

  Longitud máxima: 5 páginas.

- **Organización del trabajo**: Deberá incluir una descripción de las tareas realizadas por cada integrante del equipo. Ten cuidado en asegurarte que la información es suficientemente detallada e individualizada.

  Longitud máxima: 3 páginas.

- **Conclusiones**: Se valorarán especialmente las conclusiones derivadas de los resultados obtenidos en las evaluaciones realizadas, las conclusiones sobre lo aprendido en la realización del trabajo y las conclusiones que relacionen el trabajo con el contenido de la asignatura.

  Longitud máxima: 2 páginas.

5. Procedimiento de entrega

La entrega de la memoria del proyecto se entregará a través de un entregador habilitado al efecto en Aula Global. En este entregador se colocará un único archivo que deberá llamarse **report.pdf**.

Todo el proyecto desarrollado en el proyecto de GitHub asignado se entregará además en un archivo ZIP a través de Aula Global en un entregador habilitado al efecto. En este entregador se colocará un único archivo comprimido en formato ZIP que deberá llamarse **render.zip**.

6. Calificación

Las calificaciones finales para este proyecto se obtienen de la siguiente forma:

- Rendimiento: 20 %.
- Energía: 20 %.
- Modificaciones al diseño y paralelización: 10 %.
- Evaluación del rendimiento y energía: 20 %
- Contribuciones individuales: 20 %
- Conclusiones: 10 %.