UNIVERSITAT POLITÈCNICA DE VALÈNCIA

DSIC DEPARTAMENTO DE SISTEMAS INFORMÁTICOS Y COMPUTACIÓN

Máster Universitario en Computación en la Nube y de Altas Prestciones

Tecnología de la Programación Paralela (TPP).

Actividades para el aprendizaje de Threading Building Blocks (TBB)

Instrucciones

Este boletín está formado por una serie de actividades. Para cada actividad hay que crear una carpeta con nombre "Actividad_X" donde X indica el número de actividad. Dentro de la carpeta figurarán los códigos implementados y los documentos solicitados. Después se creará un fichero comprimido que incluya todas las carpetas y se subirá a la tarea de PoliformaT correspondiente. Borrad todos los ejecutables antes de crear el archivo comprimido y todos aquellos archivos que no sean el fichero fuente o el documento con resultados solicitado. Solo se solicitan los códigos fuente desarrollados.

Introducción

Esta parte contiene una serie de ejercicios propuestos de Threading Building Blocks. Adjunto a este enunciado y para la mayoría de actividades existe un código fuente base de la versión secuencial que se ha de paralelizar.

La librería TBB con la que vamos a trabajar es la que viene con la distribución de Intel. Por ello, es posible que antes de comenzar haya que leer las variables de entorno para trabajar con TBB:

```
source /opt/intel/tbb/bin/tbbvars.sh intel64
```

aunque también es posible que no sea necesario porque ya están inicializadas dichas variables.

A partir de aquí se puede trabajar con TBB, tanto con el compilador de Intel como con el de GNU.

Actividad 1

Copiad el siguiente código en un fichero con la extensión `.cpp`, por ejemplo, `actividad1.cpp`.

```
#include <iostream>
#include <tbb/task_scheduler_init.h>
using namespace std;
using namespace tbb;

int main() {
 task_scheduler_init init;
 cout << "Numero de threads = " << init.default_num_threads() << endl;
 return 0;
}
```

El programa se puede compilar con el compilador de GNU así:

```
g++ -o actividad1 actividad1.cpp -ltbb
```

o con el propio compilador de Intel:

```
icpc -o actividad1 actividad1.cpp -tbb
```

Obsérvese que para Intel es una opción del compilador mientras que para GNU es una librería. En principio, nosotros trabajaremos con el compilador de GNU, `g++`.

El programa se ejecuta normalmente:

```
./actividad1
```

Lanzad este programa para obtener el número de hilos disponible en un nodo.

Ahora vamos a copiar el siguiente código en sustitución del anterior:

```
#include <iostream>
#include "tbb/task_scheduler_init.h"
#include "tbb/tick_count.h"

int main() {
  int n = tbb::task_scheduler_init::default_num_threads();
  for (int p = 1; p <= n; ++p) {
     // Construccions de un planificador de tareas con p hilos
     tbb::task_scheduler_init init(p);
     tbb::tick_count t0 = tbb::tick_count::now();
     // En este punto se debería introducir el código a medir
     tbb::tick_count t1 = tbb::tick_count::now();
     double t = (t1 - t0).seconds();
     std::cout << "tiempo = " << t << " con " << p << " hilos" << std::endl;
     // El planificador se destruye de manera implícita
  }
  return 0;
}
```

Puede ser necesario incluir la librería `rt` en el enlazado del programa.

Este código servirá para tomar tiempos y variar el número de hilos a utilizar (si es que se desea variar el número de hilos).

Entrega: De esta actividad no hay que entregar nada. (**0.0 puntos**)

Actividad 2

Paralelizar con TBB el siguiente código:

```
void SerialApplyFoo( double a[], size_t n ) {
  for( size_t i=0; i<n; ++i )
    Foo(a[i]);
}
```

El código para la función `Foo` puede ser:

```
void Foo( double& f ) {
  unsigned int u = (unsigned int) f;
  f = (double) rand_r(&u) / RAND_MAX;
  f = f*f;
}
```

Utilizad el fichero base facilitado en la carpeta material y las transparencias de clase.

Se pide dos versiones de este programa, la primera sin expresiones lambda y la segunda con expresión lambda.

Entrega: De esta actividad solo hay que entregar el código del programa. (**0.20 puntos**)

Actividad 3

Paralelizar con TBB el programa desarrollado en la Actividad 1 del Boletín de actividades de OpenMP. Se recomienda utilizar el código `actividad3.cpp` de la carpeta `material`.

También se recomienda seguir estos dos pasos:

1. El código paralelo consistirá en una llamada al algoritmo `parallel_for`, tal como se hizo en el ejercicio anterior, donde el primer argumento es de tipo `blocked_range` y el segundo es un objeto función. Este segundo argumento puede ser la llamada al constructor de una clase que implementaremos en el siguiente punto.

2. Construir una clase con el operador de acceso a función `operator()` implementado. Este operador recibe como argumento el `blocked_range` sobre el que va a trabajar. La implementación consiste en el bucle para la `v` que se quiere paralelizar. Cada uno de los objetos (`tam`, `M`, `media` y `desvt`) tendrán que ser miembros de la clase y ser inicializados mediante un constructor.

Esta actividad, al igual que la anterior, se compone de dos tareas. La primera consiste en la implementación siguiendo los pasos anteriores, es decir, sin la utilización de expresiones lambda. Para la segunda tarea se utilizarán expresiones lambda.

Entrega: De esta actividad solo hay que entregar el código del programa. (**0.20 puntos**)

Actividad 4

Dado el algoritmo de Cholesky facilitado en la carpeta `material`, compiladlo mediante:

```
g++ -o cholesky cholesky.cpp ctimer.c -lblas -llapack
```

y probadlo.

Tarea 1: Copiar el algoritmo `cholesky.cpp` en uno nuevo, `cholesky_tarea1.cpp`, y paralelizarlo de la siguiente manera:

1. Paralelizar los dos bucles intermedios (para `i`) mediante el algoritmo `parallel_for` y creando las clases *body* pertinentes.
2. Sobre la paralelización anterior, paralelizar el bucle interno al segundo bucle para `i` para que se ejecuten las multiplicaciones en paralelo.

Tarea 2: Copiad de nuevo el algoritmo `cholesky.cpp` en un fichero llamado `cholesky_tarea2.cpp`, y paralelizadlo tal como se ha hecho en la tarea anterior pero utilizando en este caso expresiones Lambda.

Tarea 3: Copiar el código original `cholesky.cpp` en uno nuevo, `cholesky_tarea3.cpp`, y paralelizarlo utilizando la clase `blocked_range2d` para el segundo bucle intermedio. En este caso se deben utilizar directamente las expresiones Lambda para paralelizarlo.

Entrega: De esta actividad se entregan los códigos y una tabla con tiempos de ejecución para varios tamaños de problema comparando las versiones desarrolladas en las tareas 2 y 3 con al menos una de las versiones paralelizadas mediante OpenMP que se hicieron de este algoritmo. Es suficiente con aportar alguna tabla de tiempos en un fichero de texto aunque, si se desea, es mejor aportar una gráfica comparativa. No hay que variar el número de hilos, se utilizarán los que vienen por defecto. (**0.50 puntos**)

Actividad 5

Paralelizar con TBB el siguiente código:

```
double SerialSumFoo( double a[], size_t n ) {
  double sum = 0;
  for( size_t i=0; i!=n; ++i )
    sum += Foo(a[i]);
  return sum;
}
```

donde la función `Foo` puede ser la siguiente:

```
double Foo( double f ) {
  unsigned int u = (unsigned int) f;
  double a = (double) rand_r(&u) / RAND_MAX;
  a = a*a;
  return a;
}
```

Existe un código, `actividad5.cpp`, que se puede utilizar para implementar esta actividad.
Entrega: De esta actividad solo hay que entregar el código del programa. (**0.15 puntos**)

Actividad 6

Paralelizar con TBB el siguiente código:

```
long SerialMinIndexFoo( const double a[], size_t n ) {
  double value_of_min = DBL_MAX;
  long index_of_min = -1;
  for( size_t i=0; i<n; ++i ) {
    double value = Foo(a[i]);
    if( value < value_of_min ) {
      value_of_min = value;
      index_of_min = i;
    }
  }
  return index_of_min;
}
```

Existe un código, `actividad6.cpp`, que se puede utilizar para implementar esta actividad.
Entrega: De esta actividad solo hay que entregar el código del programa. (**0.20 puntos**)

Actividad 7

Paralelizar con TBB el siguiente código:

```
#include <iostream>
using namespace std;

void Foo( double& f ) {
  f = f*f;
}

struct s_item {
  double data;
  struct s_item *next;
};

typedef struct s_item Item;

void SerialApplyFooToList( Item *root ) {
  for( Item* ptr=root; ptr!=NULL; ptr=ptr->next )
    Foo(ptr->data);
}

int main( )  {

  long n = 10;
```

```
  Item *root = NULL;
  root = new( Item );
  root->data = 0.0;
  Item *p;
  size_t i;
  for( i=1, p = root; i<n; i++, p = p->next ) {
    p->next = new( Item );
    p->next->data = (double) i;
    p->next->next = NULL;
  }
  SerialApplyFooToList( root );

}
```

Hay que observar que el bucle a paralelizar en la función `SerialApplyFooList` posee un número indeterminado de iteraciones, tantas como elementos en la lista, y el número de elementos no se conoce a priori. Por lo tanto, no sirven los algoritmos vistos hasta ahora. El algoritmo a utilizar aquí es `parallel_do`. Para utilizar este algoritmo es necesario que exista un *iterador* (`Iterator`) sobre la lista de elementos. Así pues, en la nueva función a construir (`ParallelApplyFooList`) tendremos que crear un contenedor para poder disponer de un *iterador* que se pueda utilizar en la función `parallel_do`. Se puede utilizar, por ejemplo, el contenedor `list` de C++.

Existe un código, `actividad7.cpp`, que se puede utilizar para implementar esta actividad.

Entrega: De esta actividad solo hay que entregar el código del programa. Se pueden utilizar o no expresiones *lambda* con total libertad. (**0.15 puntos**)

Actividad 8

Paralelizar con TBB el código facilitado en el fichero `actividad8.cpp` teniendo en cuenta que se trata de un *pipeline*. Para ello, debéis copiar el código que resuelve este problema y que tenéis en las transparencias de clase y en el libro de TBB de la bibliografía. La parte del código marcada como "Resolucion secuencial" es la que debería ser sustituida por el código paralelo.

En la carpeta existe un fichero (`fichero_entrada`) con un texto con el que trabajar.

Entrega: De esta actividad solo hay que entregar el código del programa. (**0.10 puntos**)

Actividad 9

El algoritmo de TBB `parallel_invoke` permite ejecutar en paralelo una serie de funciones. Recibe como argumentos objetos función o punteros a funciones y se pueden aceptar entre 2 y 10 argumentos. Una forma de utilizarlo consiste en crear una clase que implemente el operador de llamada a función (`operator()`), es decir, lo que llamamos un *functor*. Dicho operador implementará lo mismo que la función que se pretende ejecutar en paralelo y que será sustituida por esta nueva clase.

En el caso de funciones recursivas, las llamadas recursivas serán sustituidas por una llamada a la función `parallel_invoke` a la que se le pasarán como argumentos estas funciones u objetos función. Estos dos argumentos consistirán en la instanciación de dos objetos de la clase creada. Los argumentos del constructor de esta clase coincidirán con los argumentos de la antigua llamada a la función a la que sustituyen.

Consultad y probad la paralelización realizada de la función de Fibonacci mediante `parallel_invoke`, (`fibonacci_parallel_invoke.cpp`) y mediante `task_group` (`fibonacci_task_group.cpp`).

Tarea a realizar: Implementad una versión equivalente a cada una de las dos versiones anteriores pero con expresiones *lambda*. Para ello, utilizad el fichero proporcionado `fibonacci.cpp` y sustituid las funciones `fib1` y `fib2` por una versión con `parallel_invoke` y con `task_group`, respectivamente. NOTA: La idea de las expresiones *lambda* es la de evitar la necesidad de modificar la signatura de la función recursiva secuencial de base `fib`.

Entrega: De esta actividad solo hay que entregar el código del programa. (**0.15 puntos**)

Actividad 10

Paralelizar con TBB el algoritmo de ordenación *quicksort* adjunto en el fichero `quicksort.cpp` siguiendo las siguientes tareas:

Tarea 1: Realizar la paralelización mediante el algoritmo `parallel_invoke`.

Tarea 2: Realizar la paralelización mediante la clase `task_group`.

Este ejercicio hay que realizarlo directamente con expresiones lambda.

Entrega: De esta actividad solo hay que entregar el código del programa. (**0.15 puntos**)