Part B - Languages

OpenMP - Stencil and Fork-Join

Describe two more parallel patterns
Implement stencil and fork-join patterns using OpenMP directives

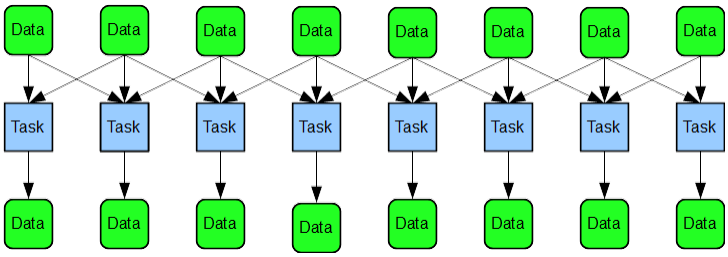
Stencil | Fork-Join | Exercises

The stencil and fork-join patterns are two other common patterns in parallel programming. The stencil pattern represents a variation on the map pattern that accesses data from neighbouring elements. A specialization of this pattern is convolution, which is central to machine learning technologies. The fork-join pattern represents a template for solutions to problems that involve divide and conquer strategies. This pattern covers problems in which tasks may differ from one another and problems in which completion of tasks in deferred until the data set is small enough for rapid solution.

This chapter describes the stencil and fork-join patterns in more detail. In each case, the serial code for an algorithm is presented first, followed by a parallel implementation using OpenMP directives.

THE STENCIL PATTERN

The stencil pattern represents regular data access to neighbouring elements. The inputs for any particular element are based on a regular access to a data at fixed offsets from the element. Each output data element is an identical function of its input and those of its neighborhoods.



The Stencil Pattern

The following serial function identifies the neighborhood for each element of **`out`** using offsets (specified by **`offset`**) and assembles contributions from the neighbouring input elements using the **`combine`** operation

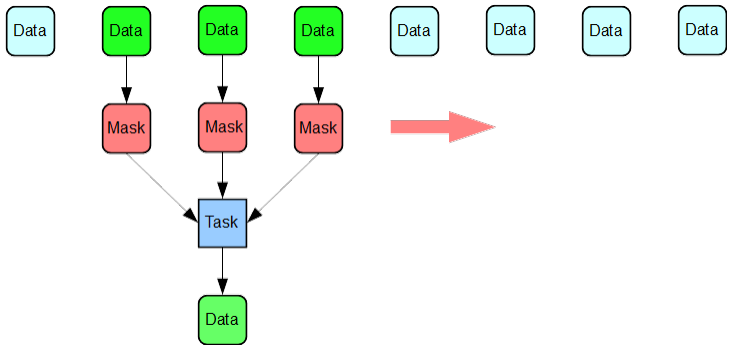
```
// stencil.h
// after McCool et al. 2012

template <typename T, typename U, typename C, int noffsets>
void stencil(
    const T* in,
    U* out,
    const int size,
    const int* offset,
    const T bound,
    C combine
    )
{
    T* neighborhood = new T[N];
    for (int i = 0; i < size; i++) {
        for (int j = 0; j < noffsets; j++) {
            int k = i + offset[j];
            if (k >= 0 && k < size)
```

```
                neighborhood[j] = in[k];
            else
                neighbourhood[j] = bound;
        }
        out[y] = combine(neighborhood);
    }
    delete [] neighbourhood;
}
```

Convolution

A convolution is a linear stencil that applies fixed weights to the elements of a neighborhood in the combination operation.

The following diagram illustrates the migration of a mask across the input data elements in combining their contributions to the generation of an output element. The highlighted data elements constitute the neighborhood and the **`mask`** elements specify the weights that apply to those input elements in the combination operation.



The Mask

Serial Example

The serial and parallel versions of the sample application consist of 5 files along with support from the Corona 1.0.2 API to access and save the data stored in a **`png`** file:

- **`main.cpp`** - main program
- Convolution Module
  - **`convolution.h`**
  - **`convolution.cpp`**
- Kernel Module
  - **`kernel.h`**
  - **`kernel.cpp`**

The main function performs the convolution on the file named on the command line using a serial algorithm, as well as its multi-threading equivalent.

```
// Convolution Example
// main.cpp
// 2020.10.22
// Chris Szalwinski

#include <iostream>
#include <sstream>
#include <cstdlib>
#include <string>
#include <chrono>
#include "corona.h"
#include "convolution.h"
using namespace std::chrono;
```

```
constexpr int MAX_NUM_THREADS = 8;

// report system time
//
void reportTime(const std::string& msg, steady_clock::duration span) {
    auto ms = duration_cast<microseconds>(span);
    std::cout << msg << ms.count() << " microseconds" << std::endl;
}

int main(int argc, char* argv[]) {
    if (argc > 2) {
        std::cerr << argv[0] << ": invalid number of arguments\n";
        std::cerr << "Usage: " << argv[0] << "  name of the image file\n";
        return 1;
    }

    // Open the file named on the command line
    corona::Image* image = corona::OpenImage(argv[1],
        corona::FF_AUTODETECT, corona::PF_R8G8B8A8);
    if (!image) {
        std::cerr << "Failed to open file named: " << argv[1] << "\n";
        return 2;
    }
    // extract the file descriptors
    int width = image->getWidth();
    int height = image->getHeight();
    void* pixels = image->getPixels();

    int np = 0;
    std::stringstream msg;
    steady_clock::time_point ts, te;

    // Create the Reference Solution
    ts = steady_clock::now();
    corona::Image* resultImage = convolute(image);
    te = steady_clock::now();
    reportTime("Serial processing time = ", te - ts);
    // Draw the reference convoluted image
    corona::SaveImage("Serial.png", corona::FF_AUTODETECT,
        resultImage);

    // Generate the Multi-Threading Cases
    for (int npr = 0; npr < MAX_NUM_THREADS; npr++) {
        ts = steady_clock::now();
        resultImage = convolute(image, npr + 1, np);
        te = steady_clock::now();
        msg << "Processing time (" << npr + 1
            << "," << np << " processor(s)) = ";
        reportTime(msg.str(), te - ts);
        msg.flush();
        msg.seekp(0);
    }
    // Draw the last convoluted image
    corona::SaveImage("MultiThreaded.png", corona::FF_AUTODETECT,
        resultImage);
}
```

The header file for the convolution module declares the prototypes for the serial and parallel convolution algorithms.

```
#pragma once
// Convolution Example - The Image
// convolute.h
```

```
// 2020.10.22
// Chris Szalwinski

#include "corona.h"

corona::Image* convolute(corona::Image*);
corona::Image* convolute(corona::Image*, const int, int&);
```

```
// Convolution Application - The Kernel
// convolute.cpp
// 2020.10.22
// Chris Szalwinski

#include <omp.h>
#include "convolution.h"
#include "kernel.h"

corona::Image* convolute(corona::Image* image) {

    const int width = image->getWidth();
    const int height = image->getHeight();
    int px, py;
    unsigned char* pixels = (unsigned char*)(image->getPixels());
    corona::Image* result = corona::CloneImage(image);
    unsigned char* resultPixels = (unsigned char*)(result->getPixels());
    Kernel kernel;

    for (py = 0; py < height; py++)
        for (px = 0; px < width; px++)
            (Pixel&)resultPixels[(py * width + px) * 4] =
            kernel.combine(pixels, px, py, width, height);

    return result;
}
```

The kernel module specifies the nature of the convolution. The header file includes the kernel mask for sharpening an image:

```
#pragma once
// Convolution Example - The Kernel
// kernel.h
// 2020.10.22
// Chris Szalwinski

struct Pixel {
    unsigned char accuR;
    unsigned char accuG;
    unsigned char accuB;
    unsigned char accuA;
};

class Kernel {
    double kernel[9] {
     0, -1,  0,
    -1,  5, -1,
     0, -1,  0 }; // sharpen
    int width{ 3 };
    int height{ 3 };
public:
    Pixel combine(const unsigned char*, const int, const int,
        const int, const int);
```

```
};
```

You can find different kernels here.

The implementation file for the kernel module transforms the original image to the convoluted one:

```
// Convolution Application - The Kernel
// kernel.cpp
// 2020.10.22
// Chris Szalwinski

#include "kernel.h"

unsigned char bracket(int x) {
    if (x > 255) x = 255;
    else if (x < 0) x = 0;
    return (unsigned char)x;
}

int bracket(int x, int max) {
    if (x > max) x = max;
    else if (x < 0) x = 0;
    return x;
}

Pixel Kernel::combine(const unsigned char* pixels, const int px, const int py,
    const int tWidth, const int tHeight) {

    int accuR = 0;
    int accuG = 0;
    int accuB = 0;
    int accuA = 255;

    for (int kernelY = 0; kernelY < height; kernelY++) {

        for (int kernelX = 0; kernelX < width; kernelX++) {

            int targetPixelX = bracket(px - width / 2 + kernelX, tWidth - 1);
            int targetPixelY = bracket(py - height / 2 + kernelY, tHeight - 1);

            int pixelIndex = (targetPixelY * tWidth + targetPixelX) * 4;
            accuR += int(pixels[pixelIndex++] *
             kernel[kernelY * width + kernelX]);
            accuG += int(pixels[pixelIndex++] *
             kernel[kernelY * width + kernelX]);
            accuB += int(pixels[pixelIndex++] *
             kernel[kernelY * width + kernelX]);
        }
    }

    return Pixel{ bracket(accuR), bracket(accuG), bracket(accuB),
     bracket(accuA) };
}
```

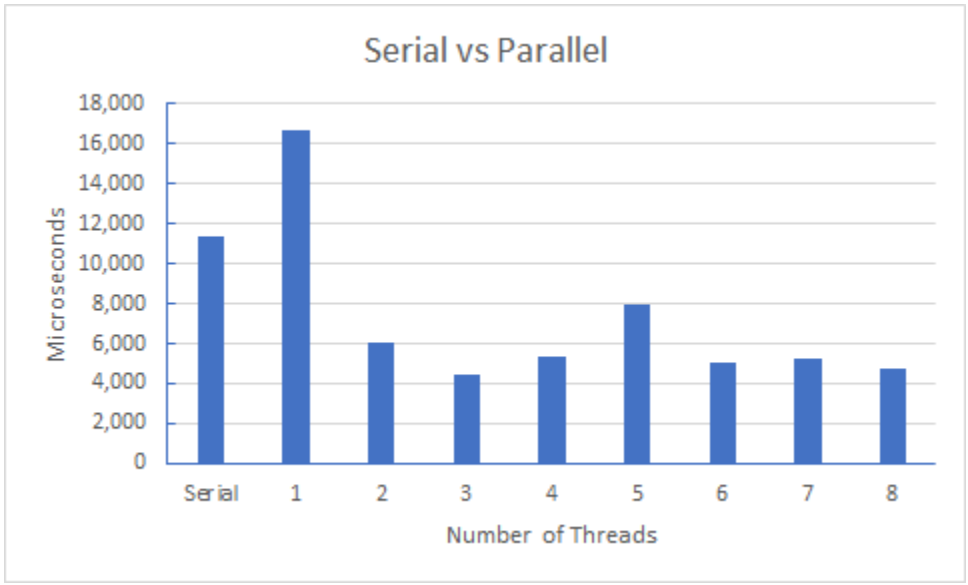
The figures below show the original and sharpened images:


Parallel Solution

The parallel solution requires adding two directive to the three-parameter convolute function. The **`private`** clause ensures that the **`x`** and **`y`** iteration control variables are not shared across the threads, but are local to each thread. The **`firstprivate`** clause ensures that the **`pixels`**, **`width`** and **`height`** variables are local to each thread and initialized to the same values.

```
corona::Image* convolute(corona::Image* image,
    const int threadsRequested, int& threadsSpawned) {

    const int width = image->getWidth();
    const int height = image->getHeight();
    int px, py;
    unsigned char* pixels = (unsigned char*)(image->getPixels());
    corona::Image* result = corona::CloneImage(image);
    unsigned char* resultPixels = (unsigned char*)(result->getPixels());
    Kernel kernel;

    #pragma omp parallel num_threads(threadsRequested)
    {
        threadsSpawned = omp_get_num_threads();
        #pragma omp for simd private(py, px), firstprivate(pixels, width, height)
        for (py = 0; py < height; py++)
            for (px = 0; px < width; px++)
                (Pixel&)resultPixels[(py * width + px) * 4] =
                kernel.combine(pixels, px, py, width, height);
    }

    return result;
}
```

The figure below compares the performance of this application for different numbers of threads.



Convolution using a 3 * 3 Kernel

THE FORK-JOIN PATTERN

The fork-join pattern is a pattern for parallelizing tasks rather than multiple data subject to the same task. A single flow of control forks into two or more flows that subsequently join to reconstitute a single flow of control. Each flow of control is independent of every other flow in the pattern and the tasks in the different flows may differ.

`task` and `taskwait`

OpenMP implements fork-join using the **`task`** construct to identify a fork. The **`taskwait`** construct identifies a join. The join applies to all child tasks of the current task. The implementation requires a parallel region.

```
#pragma omp parallel
{
    #pragma omp task
    foo(); // this statement is spawned as a child task
    goo(); // executes in parallel
    #pragma omp taskwait // waits for the child task to complete

}
```

OpenMP tasks capture local variables by value (**`firstprivate`**) and non-local variables by reference. Use **`shared`** to capture local variables by reference

```
double x;
#pragma omp parallel
{
    #pragma omp single
    {
        double a, b;
        #pragma omp task shared(a)
        a = foo(); // this statement is spawned as a child task
        b = goo(); // executes in parallel
        #pragma omp taskwait // waits for the child task to complete
        x = a + b;
    }
}
```

Task Dependencies

The **`task`** construct takes one or more optional **`depend`** clauses, which identify its dependencies. Each **`depend`** clause takes as its argument the type of dependency (**`in`**, **`out`**, or **`inout`**) followed by a variable or comma-separated list of variables. A colon (**`:`**) separates the type from the variable(s):

```
#pragma omp parallel
{
    #pragma omp single
    {
        double a, b, c = 40.0;
        #pragma omp task depend(out: a)
        a = foo(); // this statement is spawned as a child task
        #pragma omp task depend(in: a) depend(out: b)
        b = goo(); // this statement is spawned as a child task
        #pragma omp task depend(in: a, b) depend(inout: c)
```

```
        c += hoo(a, b); // this statement is spawned as a child task
    } // taskwait is implicit
}
```

EXERCISES

- Barlas, G. (2015). Multicore and GPU Programming - An Integrated Approach. Morgan-Kaufmann. 978-0-12-417137-4. pp.179-192.
- Jordan, H. F., Alaghband, G. (2003). Fundamentals of Parallel Processing. Prentice-Hall. 978-0-13-901158-7. pp. 269-273.
- Mattson, T. (2013). Tutorial Overview | Introduction to OpenMP
- Dan Grossman's 2010 Lecture on Parallel Prefix and Parallel Sorting