AN INTRODUCTION TO PARALLEL COMPUTING

CHAPTER 3 PARALLEL MODELS AND COMPLEXITY

BY FRED ANNEXSTEIN

3
PARALLEL MODELS AND COMPLEXITY

CHAPTER OBJECTIVES

The goal of this chapter is to review several parallel programming models which can provide tools for the analysis of computational and communication complexity of parallel algorithms. We will apply the work and step model to a collection of very common parallel communication patterns including, map, scatter, gather, and stencil. We will then extend the analysis to some classic parallel algorithms, including reduction, prefix-scan, quicksort, and histogram. We will gain facility with expressing and analyzing algorithms using parallel asynchronous recursion.

For parallel algorithms it is natural to initially compare designs to well known sequential programs solving the same problem. In order to address issues of scalability, we want to target our designs to be able to make use of machines that are

parameterized by the number of cores available and the amount of local memory available for each process. Hence, we will often express results using asymptotic notations involving the big-Oh notation, which is appropriate for studying problems of scalability using large machines and data sets. Using this asymptotic analysis we explore the concept of work-efficiency, and apply the concept to different methods for implementing parallel prefix-scan operations. Finally we end with an exercise to implement a stencil for filtering in image processing.

SIMD MODELS FOR DATA PARALLELISM

Data-parallel programs require simplified hardware and therefore are more likely to successfully scale. Some hardware models use the acronym SIMD which stands for 'single instruction, multiple data', and refers to large-scale processors in which an array of processing units all execute the same sequence of instructions on different data values.

Historically, processors in SIMD computers consisted of a dense, high-speed network of Arithmetic Logic Units (ALUs). In the 'pure-SIMD' model every instruction is synchronously executed at the same time by

all the processors - with the exception that some processors may be disabled for particular instructions.

The SIMD programming paradigm is used in many scientific problems. Today the SIMD paradigm is a model implemented at a coarse grain and mapped to GPU processors in large cluster computing systems incorporating efficient graphics hardware.

PRAM COMPLEXITY MODELS

The computational complexity of data parallel algorithms refers to theoretical and asymptotic comparisons of algorithms. The models build on the analysis of sequential algorithms which typically use the well-studied von Neumann or RAM (random access memory) model to express runtimes. We typically express the runtime of a sequential program as an asymptotic function of the input size using big-Oh notation. We can do a similar analysis with memory usage as a function of input size. The asymptotic runtime of a sequential program is identical on any serial platform.

In contrast, when we express the asymptotic parallel runtime of a program, we do so in terms of the size of the input, but we

also must account for the number of processors and the communication parameters of the target machine model.

The simplest model for parallel complexity analysis is called the PRAM model, which is inspired by the sequential von Neumann RAM model, but with an unbounded number of processors and uniform unit communication costs.

WORK AND STEP COMPLEXITY

The Work complexity of an algorithm is defined as the total number of operations executed by a computation as function of input size. With work complexity there is no difference between one processor or many, since we are simply counting total number of operations.

The Step complexity is defined as the longest chain of sequential dependencies in the computation. When expressing results of step complexity we often assume the PRAM model with an unbounded number of parallel processors, all of which communicate in uniform unit time. We consider several basic parallel algorithms and express their asymptotic work and step complexity using the PRAM model.

THE MAP OPERATION

The map operation is a basic higher-order function on arrays that takes another function as an argument - the map function applies the argument function f-arg operating on each array element. The map function is built natively into python and other functional languages.

```
>>> add3 = lambda x: x+3
>>> map(add3, [1, 2 ,3, 5])
[4, 5, 6, 8]
```

A parallel map is code that is designed to assigns a thread to execute the function f-arg to each item in an array. Each thread will typically use its own local memory and unique ID. We express the work complexity of the map function on n data items as order n times the work of the function f-arg. The step complexity of the map function on n data items is expressed as a constant times the step complexity of the function f-arg.

```
W(n) = O(n) * Work(f-arg)
S(n) = O(1) * Step(f-arg)
```

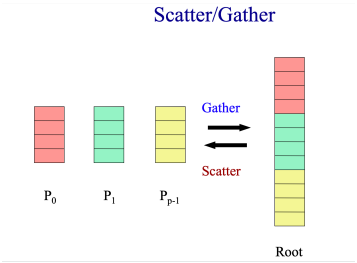
COLLECTIVE COMMUNICATIONS

A collective communication is network communication involving all the processors in a parallel computation. Collective communications imply a *synchronization point* among all the processors. This means that all processes must reach a point in their code after the communication is finished and before processors can all begin executing again.

BROADCAST, SCATTER, AND GATHER

A *broadcast* is a standard collective communication operation in which one processor sends the same data to all other processors. One of the main uses of broadcasting is to send out configuration parameters to all processors.

A *scatter* is another standard collective communication operation that involves a designated root processor sending data to all other processors. The difference between a broadcast and a scatter operation is that in a broadcast the root processor sends the *same* piece of data



to all processors, while in a scatter the root processor sends different *chunks of an array* to different processes.A *gather* is the inverse of a scatter in which chunks of an array are gathered together at a root processor.

We can express the work and step complexity of a p-processor collective communication as follows. For a broadcast of (constant size) data to p processors we have linear work O(p). The step complexity needed to implement a broadcast is generally considered as O(log p). The reason for this is that there is usually available some sort of tree-based communication network underlying any broadcast. Trees with p (leaf) processors can be built with log p levels, hence the result. Likewise, if we assume that the chunks of the array are of constant size we have that the work complexity of these operations is linear O(p), and the step complexity is logarithmic O(log p) using a tree-based communication network.

The Work complexity of Broadcast, Scatter and Gather is linear: W(p) = O(p)

The Step complexity of Broadcast, Scatter and Gather is logarithmic: S(p) = O(log p)

MANY TO MANY COMMUNICATION

The collective communication of broadcast, scatter, and gather are *many-to-one* or *one-to-many* communication patterns, which simply means that many processors send/receive to one processor. Oftentimes it is useful to be able to send many elements to many processes (i.e., a *many-to-many* communication pattern). The *all-gather* operation will gather all of the elements to all the processes. And in the most basic sense, all-gather is simply a gather followed by a broadcast. The *all-to-all* operation is the sending of data from all to all processors, and in the basic sense is simply a broadcast from every processor. There are a number of ways to optimize these many-to-many patterns depending on the type of communication network deployed.

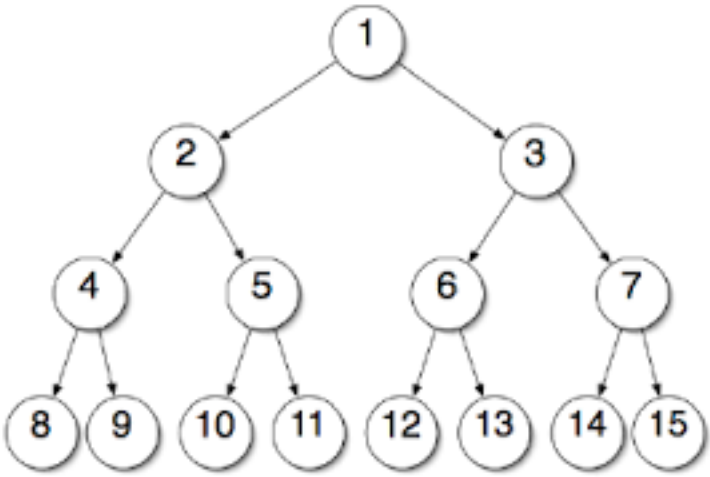
STENCILS

Stencils are a special type of many-to-many operation in which a fixed sized window is replicated to cover the data array, centered on each and every array element.

Here processors are communicating data exchanges within local neighborhoods, usually across sub-meshes or sub-cubes. Many image applications often use small 2D stencil patterns applied to large 2D image array. Stencils are used as a common class of applications related to convolution signal filtering. Another common use case is in finding localized statistics, averages, and variances across the stencil pattern.

SUMMING AN ARRAY

Consider the problem of summing 16 numbers in parallel, say using a balanced binary tree. The work required by this computation is 15 operations all additions. The step complexity required is 4 operations, since the longest chain of dependencies is the depth of the summation tree--the sums



need to be calculated starting with pairs of numbers at the leaves and going up one level at a time step. We see that summing n numbers (where n is a power of 2) using a balanced binary tree pattern requires linear work and logarithmic steps.

```
Work complexity of summing n numbers is
linear:
W(n) = O(n)

Step complexity of summing n numbers is
logarithmic: S(n) = O(log n)
```

RECURSIVE REDUCTIONS

Summing is a reduction operation since we take as input n numbers and the result is a single number. The idea of reduction can apply to any binary, associative operation, such as max, min, and product. Consider the reduction operation with a function argument f-arg for reducing an array x of values reduce(f-arg, x). We can described this operation as a simple recursive function. Note that the recursion tree implied is closely related to the binary tree of the sum reduction seen above. Here is a recursive python code for a generic reduction.

```
def reduce(f_arg, x):
   if len(x) == 1:
     return x[0]
   if len(x) == 2:
     return f_arg(x[0],x[1])
   else:
     left = reduce (f_arg, x[0: len(x)//2] )
     right = reduce(f_arg, x[len(x)//2 :])
     return f_arg(left,right)

>>> reduce(lambda x,y: x+y, [1,2,3,4,5])
15
```

The work and step complexity of the recursive reduction remains unchanged from the previous sum reduction so long as the function argument f_arg has constant work and step complexity. This can be seen from the fact that the recursion tree has linear size and logarithmic depth.

```
The Work complexity of recursive reduction
is linear:  W(n) = O(n)

The Step complexity of recursive reduction
logarithmic: S(n) = O(log n)
```

THE SCAN OPERATION

The prefix-scan operation generalizes the reduction operation and is one of most important data parallel primitives. The prefix-

scan is used in many parallel applications, as we will see. The prefix-scan is simply the running reduction of the elements in an array. There are two related versions of the scan, one called *inclusive-scan* and the other is the *exclusive-scan*. In an exclusive scan, all elements of the result are reductions upto but not including the *jth* element. In an inclusive scan, all the elements *including the jth* are included in the reduction. An exclusive scan can be generated from an inclusive scan by shifting the resulting array right by one element and inserting the identity element of the binary function f-arg at the front of the result.

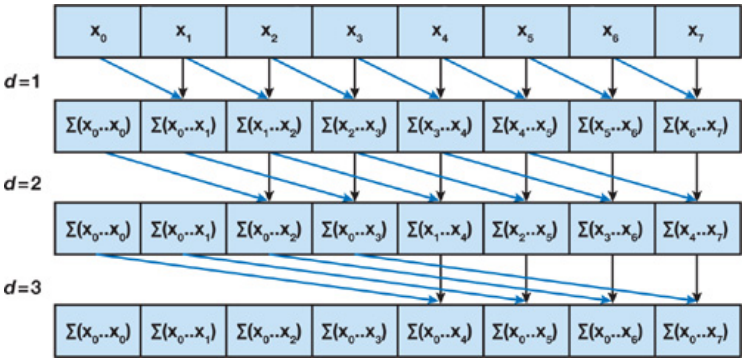
HILLIS -STEELE ALGORITHM

A first, simple attempt at implementing a prefix-scan in parallel is known as the Hillis-Steele algorithm. We consider the application to a sum prefix-scan for simplicity. The algorithm has log n steps over an array of size n. In each step d of the computation where d ranges over 1 to log n, we check whether an array index is greater than 2^d. If that is so, then we add the value of x[k – 2**(d – 1)] to the value x[k]. When

depth d reaches the log n barrier, then the calculation terminates and the result is the prefix-scan sum of the array. In each step all the individual additional array operations run in parallel. Here is pseudo-code for the Hillis-Steele algorithm.

```
#First input array x of size n

for d in range (1 to log(n)):
   for k in range(1,n+1): (in parallel)
       if k >= 2**d:
           x[k] = x[k – 2**(d–1)] + x[k]
```

We can visualize the first three steps of the Hillis-Steele algorithmic process as follows.



The problem that becomes apparent when we examine the work complexity of the Hillis-Steele algorithm is that work that is performed is $O(n \log n)$ addition operations. However, recall that a sequential scan

performs only $O(n)$ additions. Therefore, this naive implementation of Hillis-Steele is not considered work-efficient. The additional factor of log n operations can have a large impact on performance.

A NAIVE RECURSIVE PARALLEL SCAN

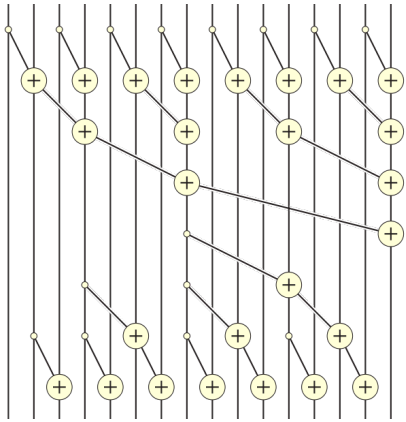
We can apply the tree-recursion idea from a modification of the recursive reduction algorithm above to the problem of computing a scan. Consider the following code for prefix-scan.

```
def scan(f_arg, x):
   if len(x) == 1:
     return [x[0]]
   if len(x) == 2:
     return [x[0], f_arg(x[0],x[1])]
   else:
     left = scan (f_arg, x[0: len(x)//2] )
     right = scan (f_arg, x[len(x)//2 :])
     r = map (lambda y: f_arg(left[-1],y), right)
     result = left + list( r )
   return result

>>> scan(lambda x,y: x+y, [1,2,3,4,5])
[1, 3, 6, 10, 15]
```

The map function that is called after the two recursive calls will significantly impact the work complexity of the algorithm so that it will have a work and step complexity worse than the previous Hillis-Steele example. We

leave it as an exercise to calculate the asymptotic complexity of this recursive scan.

BLELLOCH'S RECURSIVE METHOD



There is a more sophisticated recursive method which can out-perform both the Hillis-Steele method and the naive recursive method for computing parallel scans. The method is called the Blelloch algorithm and it is designed to compute a prefix-scan with linear work and logarithmic step complexity; hence the parallel algorithm is considered *work-efficient*. The Blelloch algorithm for computing a parallel prefix-sum can be accomplished with two recursive calls one for an up-sweep and one for a down-sweep of a tree-based structure. Details are to be found in the following reference [GE Blelloch, Vector Models for Data-Parallel Computing. The MIT Press, 1990].

COMPARING SCAN ALGORITHMS

Each of the three preceding algorithms have step complexity that is logarithmic. As noted parallel algorithms for prefix scans can be generalized to work for any efficient associative, binary operations. Their simple data parallel logic enables them to be computed efficiently on parallel hardware such as a GPUs. Here is a summary of the results on parallel scan algorithms.

```
Hillis-Steele
Work = O(n log n)     Steps = log n

Naive Recursive Solution
Work = O(n log n)     Steps = log^2 n

Blelloch
Work = O(n)           Steps = 2 log n
```

In practice, we usually do not have a processor for each array element. Instead, there will likely be many more array elements than processors. For example, if we have 32 processors and an array of 32000 numbers, then each processor should store a contiguous block of 1000 array elements. Efficient parallel implementations follow a two pass procedure where prefix sums are calculated in the first pass for each block on

each processing unit; the partial block sums are used in the parallel prefix calculation, which is then communicated back to the next processor for a second pass using the now known prefix as the initial value. Asymptotically this method takes approximately two read operations and one write operation per item.

In general, suppose we have n elements and p processors, and let us define k = n/p. Then the procedure to compute the scan is as follows:

1. Each processor does a local scan on its local data subarray.

2. The parallel prefix scan computes the scan of the block sums. Communicate this scan so that in the next block the value can be added to each of the block partial sums.

PARALLEL QUICKSORT

Let us now review the following code for the popular sorting routine known as *quicksort*. Recall that quicksort is a recursive function which partitions the unsorted list into a collection of *lesser* values each less than a given pivot element, and a collection of *greater* values each greater than the pivot. Quicksort will make a recursive call on each

of these lists, until an empty list is called, which is the base case. The following is valid python code for quicksort.

```
def qsort(list):
    if list == []:
        return []
    else:
        pivot = list[0]
        lesser = [x for x in list[1:]
                        if x < pivot])
        greater = [x for x in list[1:]
                        if x >= pivot])
    return qsort(lesser) +
              [pivot] + qsort(greater)
```

The Quicksort algorithm is not hard to parallelize, since the two recursive calls may be executed concurrently without race conditions. We can simply spawn two new processes to execute the two recursive calls in parallel. Note that all the pivot comparisons are independent and can therefore be done concurrently in parallel. We can complete the analysis by considering the structure of the recursion tree using recurrence relations. In the expected case the depth of the recursion tree is logarithmic. We have the following result showing that parallel version of quicksort is work-efficient.

Summary of Complexity of Quicksort

```
(Expected Case)
Work = O(n log n)
Steps = O(log n)
```

HISTOGRAMS IN PARALLEL

Recall the problem of computing a histogram. The basic computational problem is that for a given data set and a set of defined bins, we wish to calculate the size of each of the bins, that is the number of date elements that belong to each bin. The basic idea is to loop through all the data values and update the counters associated with the bin for that data element.

```
def count_elements(seq) -> dict:
    hist = {}
    for i in seq:
        if i in hist:
            hist[i] += 1
        else:
            hist[i] = 1
    return hist

a = (0, 1, 1, 1, 2, 3, 7, 7, 1)
print(count_elements(a))
b = "We sit in silence"
print(count_elements(b))


Output:
{0: 1, 1: 4, 2: 1, 3: 1, 7: 2}
{'W': 1, 'e': 3, ' ': 3, 's': 2, 'i': 3,
't': 1, 'n': 2, 'l': 1, 'c': 1}
```

Consider computing one histogram for each color channel on a large image. We have 256 bins, one for each intensity value in an 8-bit representation, and each pixel increments one bin for each color channel.

The main problem in parallelizing this type of operation is that we do not know the distribution of the data and therefore there are likely to be data races when two bins are being incremented at the same time. This data race problem is fairly easily solved. We may resort to locking when a counter is updated to insure thread safety. However this may have significant impacts on performance as locking can potentially serialize the code. Another solution is divide the dataset among the processors and direct them to compute local histograms sequentially. A final step is needed to bring all the local per-process histograms together. This can be handled by a final parallel reduction on the vectors of local histograms which will result in a global table of bin counts.

ASSIGNMENT #3 BLUR FILTERS

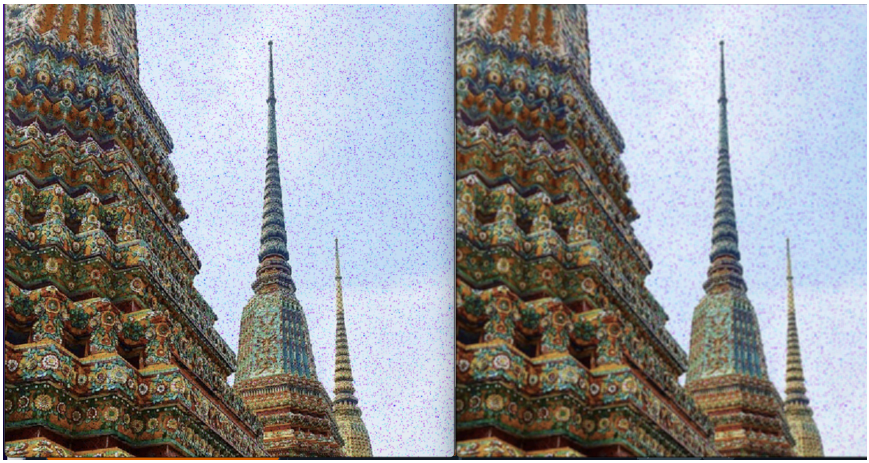
Basic blurring of images can be obtained by choosing a square stencil and calculating the median intensity values for each color

channel over the stencil for each pixel. Here is code to read in an image and produce such a blurring.

```
import matplotlib.pyplot as plt

import numpy as np
from PIL import Image

def blurfilter(in_img, out_img):
    """ For each pixel in in_img calc the mean intensity values
            using square 7x7 stencil"""
    for c in range(3):
        for  x in range(in_img.shape[1]):
           for y in range(in_img.shape[0]):
              val = 0
              for i in range(-3,4):
                 for j in range(-3,4):
                   if (x+i < img.shape[1]) and (x+i >= 0) and \
                      (y+j < img.shape[0] ) and (y+j >=0 ):
                        val += (int(img[y+j,x+i,c] ))

              out_img[y,x,c] = val // 49

img = np.array(Image.open('noisy1.jpg'))
print(img.shape)
imgblur= img.copy()
blurfilter(img, imgblur)

# Display and save blurred image
fig = plt.figure()
ax = fig.add_subplot(1, 2, 1)
imgplot = plt.imshow(img)
ax.set_title('Before')
ax = fig.add_subplot(1, 2, 2)
imgplot = plt.imshow(imgblur)
ax.set_title('After')
img2= Image.fromarray(imgblur)
img2.save(‘blurred.jpg')
```

If we run this code on an image with many noisy artifacts the results are less than satisfying-see the pair of images of the Wat Pho temple in Bankok.

Other types of blurring can be obtained by taking a weighted average over the stencil values. For example, Gaussian blurring is defined as exponentially weighted averaging, using square stencil at every pixel.



For Assignment #3 you will write a parallel program that applies a blurring filter that provides a weighted averaging of each color channel to obtain better results than the unweighted version above. You should experiment with different filter sizes, and compare speedup results you achieve for various filter sizes. Here is a link to the image on the left noisy1.jpg