Class Notes : Programming Parallel Algorithms

CS 15-840B (Fall 1992)

Guy E. Blelloch Jonathan C. Hardwick

February 1993

CMU-CS-93-115

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Abstract

These are the lecture notes for CS 15-840B, a hands-on class in programming parallel algorithms. The class was taught in the fall of 1992 by Guy Blelloch, using the programming language NESL. It stressed the clean and concise expression of a variety of parallel algorithms. About 35 graduate students attended the class, of whom 28 took it for credit. These notes were written by students in the class, and were then reviewed and organized by Guy Blelloch and Jonathan Hardwick. The sample NESL code has been converted from the older LISP-style syntax into the new ML-style syntax. These notes are not in a polished form, and probably contain several errors and omissions, particularly with respect to references in the literature. Corrections are welcome.

This research was sponsored in part by the Avionics Laboratory, Wright Research and Development Center, Aeronautical Systems Division (AFSC), U.S. Air Force, Wright-Patterson AFB, Ohio 45433-6543 under Contract F33615-90-C-1465, ARPA Order No. 7597 and in part by the Pittsburgh Supercomputing Center (Grant ASC890018P), who provided Connection Machine CM-2 and Cray Y-MP time.

Keywords: Parallel algorithms, parallel machine models, NESL

Class Notes: Programming Parallel Algorithms
CS 15-840B (Fall 1992)

CMU-CS-93-115

Guy E. Blelloch, Jonathan C. Hardwick

February 1993

These are the lecture notes for CS 15-840B, a hands-on class in programming parallel algorithms. The class was taught in the fall of 1992 by Guy Blelloch, using the programming language NESL. It stressed the clean and concise expression of a variety of parallel algorithms. About 35 graduate students attended the class, of whom 28 took it for credit. These notes were written by students in the class, and were then reviewed and organized by Guy Blelloch and Jonathan Hardwick. The sample NESL code has been converted from the older LISP-style syntax into the new ML-style syntax. These notes are not in a polished form, and probably contain several errors and omissions, particularly with respect to references in the literature. Corrections are welcome.

Keywords: PARALLEL ALGORITHMS, PARALLEL MACHINE MODELS, NESL

(141 pages)

List Of Lectures

15-840: Programming Parallel Algorithms
Scribe: Erik Seligman

Lecture #1
Tuesday, 15 Sep 92

1 The Class

1.1 Goals

- Algorithm design—thinking in parallel
  - Sorting & searching
  - Graph algorithms
  - Text searching
  - Geometry and graphics
- Implementation (learn by doing)

1.2 Class requirements

- Scribe one lecture.
- A few assignments (OK to discuss with other students).
- Final project.
- Discussion outside class (helpful but not required).

2 Why Parallel Algorithms?

2.1 Speed and size

- Speed—sorting 10 million 64-bit integers takes 200 seconds on a DEC 5000/200 serial workstation, versus only 0.2 seconds on a CM-5 with 1024 processors.
- Size—the CM-5 has 32 gigabytes of memory (32 megabytes per processor). This makes it possible to solve much larger problems.

2.2 Example problem areas

- Graphics (rendering, ray tracing)
- Simulation (weather forecasting, fluid flow, molecular dynamics)
- AI (neural nets, searching)
- Biology (gene searching)
- Hardware design (chip verification)

3 Some Definitions

3.1 Scalable parallelism

- Intuitively: if the size of the problem is increased, we can increase the number of processors effectively used, i.e., there is no limit on the parallelism.
- Formally: $T_p(n) = o(T_s(n))$, i.e.,

$$\lim_{n \to \infty} \frac{T_p(n)}{T_s(n)} = 0$$

  where $T_p(n)$ is the time taken by the parallel algorithm, and $T_s(n)$ is the time taken by the best serial algorithm. We assume that there are an unlimited number of processors available.
- Example: consider a vision system with 4 phases:

  Line-recognition $\longrightarrow$ feature-detection $\longrightarrow$ grouping $\longrightarrow$ etc.

  - If each phase must be done serially, then doing the phases in parallel only gives us at most a factor of 4 speedup.
  - Suppose that all except the "grouping" phase can be done individually with scalable parallel algorithms. Then the algorithm as a whole is still not scalable (although performance may be improved) due to the bottleneck in one phase.

3.2 Work = Processors × Time

- Intuitively: how much time a parallel algorithm would take to simulate on a serial machine.
- Formally: $W_p = PT_p$
- Example: an algorithm that requires n processors and $\log n$ time requires $n \log n$ work.

3.3 Work efficient

- Intuitively: the parallel algorithm does no more work than the best serial algorithm.
- Formally: $T_p(n) = O(T_s(n)/P)$, or $W_p(n) = O(T_s(n))$.
- Example: a parallel sorting algorithm called bitonic sort requires n processors and $O(\log^2 n)$ time, therefore requiring $O(n \log^2 n)$ work. It is not work efficient since we know that we can sort serially in $O(n \log n)$ time.
- With a work-efficient parallel algorithm, there is (in theory) no need for a serial one!

4 An Example: Primes

The problem: find all primes less than n. This example shows that naive parallel algorithms are typically not work efficient.

4.1 Serial sieve algorithm

The standard "sieve" algorithm repeatedly finds the smallest remaining prime and knocks out multiples of that prime from an array of size n:

```
for $i = 0$ to $n$ do $primes[i] = 1$
for $i = 2$ to $\sqrt{n}$ do
  if $primes[i] = 1$ then
    for $j = 2i$ to $n$ by $i$ do
      $primes[j] = 0$
```

Note that we only need to go up to $\sqrt{n}$ since every compound (non-prime) integer $\leq n$ must have at least one multiple $\leq \sqrt{n}$. The time required for this algorithm is

$$\sum_{p \in \text{primes}, p < \sqrt{n}} \left(\frac{n}{p} \right) \approx O(n \log \log n)$$

and the constant is small (≈ 0.5).

4.2 Parallel stream sieve

Feed the stream of numbers $n \ldots 2$ into a line of processors:

$$\ldots 6, 5, 4, 3, 2 \rightarrow P1 \rightarrow P2 \rightarrow P3 \ldots$$

Every processor starts inactive, and activates on the first number that it receives. It then filters out all numbers which are multiples of its number.

For example, P1 would get the value 2. Then 3 would enter P1 and pass to P2 (which will get 3, since it is inactive), while 4 enters P1 and is filtered out because it is a multiple of 2.

We need a processor for each prime less than $\sqrt{n}$; since the density of primes is about $n/\log n$, we need no more than $\sqrt{n}/\log n$ processors. The time required is $O(n)$ (since we must introduce the numbers into the pipeline one by one), and the work done is therefore $P \times T = O(n^{3/2}/\log n)$, so the algorithm is not work efficient. The constant is also bad, since we are replacing the adds of the serial sieve algorithm with mods. (It is possible to rewrite this algorithm to use adds instead.)

4.3 Parallel dropout sieve

Each number from 2 to n is assigned to a processor. The algorithm repeatedly finds the first processor that hasn't been eliminated yet; this processor then broadcasts its value to all the others, and each processor eliminates itself if it is a multiple of the broadcast number.

We use n processors and $\sqrt{n}$ time so the work done is $O(n^{3/2})$; again, this algorithm is not work efficient, and the mods here are harder to eliminate.

This algorithm is significantly faster that the previous one if we assume that we have enough processors. However, if for example we want to find all the primes less than a million, it is unlikely that we will have a million processors.

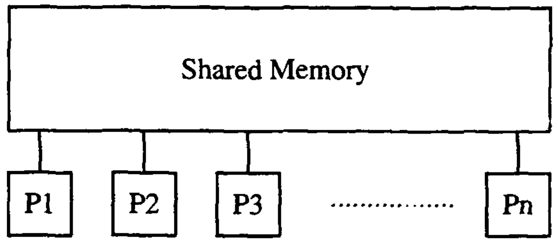


Figure 1: CRCW PRAM

5 Assumptions for Assignment 1

Use a CRCW PRAM (Concurrent Read Concurrent Write Parallel Random Access Machine). Assume that the shared memory can be accessed in unit time by any processor(s), and that synchronization is free. If more than one processor tries to write to the same location, an arbitrary one will succeed.

6 For More Information

See the NESL manual [7] for a sample primes algorithm. The different theoretical models of parallel machine are defined in Chapter 30 of Cormen, Leiserson and Rivest [12].

15-840: Programming Parallel Algorithms — Lecture #2
Scribe: Matt Zekauskas — Thursday, 17 Sep 92

Overview

- Review concepts from last lecture:
  - *Scalable.* When is an algorithm scalable?
  - *Work.* How do we define how much work an algorithm does?
  - *Work efficient.* What does it mean for an algorithm to be work efficient?
  - *Efficiently scalable.* This is our goal. An algorithm is efficiently scalable when it is both work efficient and scalable; as we increase the problem size the performance should grow linearly with the number of processors.
- Introduce the *scan* operation.
- Show some *parallel prefix* algorithms, which use the scan operation.

1 Scalability

Informally, an algorithm is *scalable* if we get a speedup using more processors for larger problems; the "performance" of the algorithm increases with problem size and the number of processors. ("Performance" can be thought of as MFLOPS or MIPS; performance = $1/\text{time}$.) An algorithm is *efficiently scalable* if, as we increase the problem size, the performance grows linearly with the number of processors. A problem is not scalable if the performance reaches a plateau. For example, if after 1000 processors we cannot get any more performance by adding more processors for larger problems, then the algorithm is not scalable. This would indicate a bottleneck, or a phase that is serial.

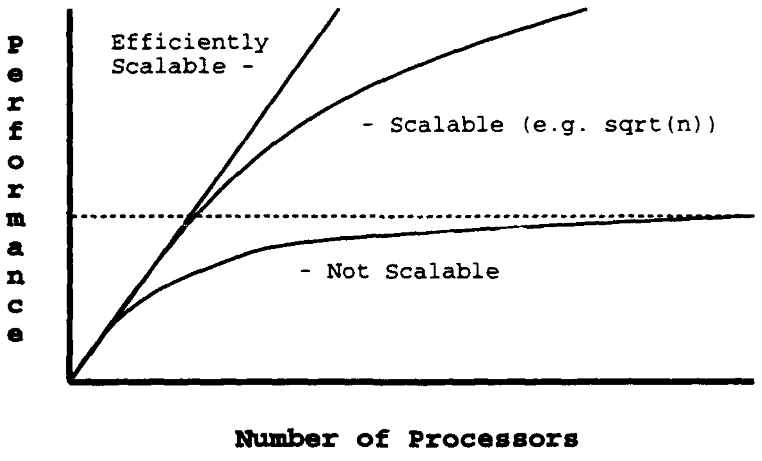


Figure 1: Not scalable vs. scalable vs. efficiently scalable. The size of the problem is assumed to grow with the number of processors.

Why should we care about scalability? For example, if we have a 1024 processor machine, and our algorithm does not reach its performance plateau with 1024 processors, but at 2048, isn't it a good algorithm? Perhaps for no . B t /hat about in five or ten years? By then, we should have at least 32K processor mach..es. Problem size seems to grow naturally—we will always have a bigger problem to solve tomorrow. This algorithm would be useless with the bigger machine and problem size.

2 Work Efficiency

Work efficiency is a stronger condition than scalability. We want performance to be linear with the numb^r of processors.

2.1 Why study work efficiency?

The basic reason why we want a work-efficient algorithm is cost. If we run our algorithm on a real supercomputer, we will be charged based on the product of the total running time and the number of processors. Wasted cycles burn money. On the other hand, if we have to buy a supercomputer to solve our problem, we would like to buy the smallest supercomputer we need.

It is also very easy to write a work inefficient algorithm, so it is important to always keep work efficiency in mind.

2.2 Causes of work inefficiency

An algorithm is *work inefficient* if it does not use all the processors all the time. If processors are doing useless work, or are idle, they are not contributing to the problem solution.

There are two reasons for not using a processor:

- The algorithm cannot use it. We will want to design a different algorithm.
- The processor is waiting for communication. Different communication patterns are handled better by different multiprocessors. We always want an algorithm with minimal communication.

Take a look at the dropout sieve algorithm from the last lecture. There were $n - 2$ processors; numbers were broadcast to all processors, who would then do a mod with their processor number, and knock themselves out if they were a multiple of that number. Most of the processors are wasting time, especially toward the end of the algorithm. At the end most of the processors have already been knocked out, and are therefore doing nothing. In addition, only a very small percentage of the active processors will actually knock themselves out on a given step. This is also true for the stream sieve algorithm. As we get toward the end of the stream, it is very rare that a processor will be able to capture a new prime number. Most of the numbers will be dropped by the first few processors. Neither algorithm can use all the processors. By contrast, the serial algorithm only does the work necessary to knock out a given prime; to knock out multiples of 997 less than 1,000,000 the algorithm does only about 1003 operations instead of 1,000,000.

2.3 Example: digital logic simulator

Given some digital logic (on the order of millions of gates), we want to change some inputs, propagate the signals to the outputs, and then verify the outputs. If we are simulating synchronous logic, there will be a clock tick, a propagation phase (which will change some inputs), and then another clock tick starting a new propagation phase.

Naive algorithm

Assign one gate to each processor. Since there will usually be many more gates than real processors, the gates will be assigned to *virtual* processors, which are then multiplexed onto the real processors. To keep communication costs down, we might multiplex gates which are close together physically onto the same physical processor. We iterate until no lines change and the logic is "settled".

Guy wrote an implementation like this, and it ran twice as slowly on an 8K processor CM-2 as on a workstation. The CM-2 has one bit per processor, so it is roughly equivalent to a 256 processor machine with 32-bit processors.

Problems

There are two possible problems with the naive parallel algorithm:

- Communication costs. If the gates are not well placed this could be a problem.
- Only a small percentage of the processors are active. At the start of the algorithm only the processors near the inputs are active. At the end only the processors near the outputs are active. At any given point only a few processors are active.

Furthermore, when a simulation is run, usually only a few inputs are changed. Therefore even fewer processors near the inputs are active (although the signals usually propagate out, and hence more processors near the outputs might be active).

The serial algorithm is efficient because it is demand-driven. Only gates that have inputs that change are evaluated—the algorithm follows the signals through the logic. The serial algorithm uses a priority queue of lines with changed levels, and so will not evaluate any gate before all its inputs are ready. An efficient parallel algorithm would use parallel priority queues.

Note that there are lots of gates that do change, but they may only comprise 25 percent of the million gates. Therefore, there is parallelism available to be exploited; if only two gates changed, we could never get an efficient parallel algorithm.

2.4 Example: graphics image rendering

Given a list of high-level object descriptions, place pixels in the appropriate spots on a raster display. The depths of the objects are known, so pixels of one object may obscure pixels from another object.

Naive algorithm

Assign one (probably virtual) processor to each pixel. Broadcast objects one at a time. The processor determines if its pixel is in the object, and if the object's pixel is on top of a previous object's pixel. If the pixel is in the foreground at that location, it is displayed, and the z (depth) of the pixel is remembered by that processor.[1]

Problems

Again, there are a small number of processors active at any given time. If adjacent virtual processors in a block are assigned to a real processor, then for a given object only a few processors will be active. Someone suggested distributing the pixels randomly amongst the real processors instead of in a local block. This would improve utilization somewhat, since more real processors would be active, but it would still be a small number when visualizing small objects.

Solutions

There are several possible improvements to the naive parallel algorithm:

- We could break the picture into quadrants, and then broadcast the quadrants to the appropriate processors. Having multiple objects broadcast by (other) multiple processors to the correct quadrants would allow all the quadrants to be busy at the same time.
- We could assign one processor per object. This would allow the objects to all be processed in parallel. However, a giant object (e.g., a wall in the background) would take the most time, and would become a bottleneck.
- Best (so far): assign m processors to each object. m could be the number of pixels in an object, or proportional to the number of pixels in an object. Note that we now have to worry about communication costs, synchronization, etc.

Summary

It is very important to pay attention to work efficiency. If we lose 50% of our speedup in efficiency, and another 10% due to communication among processors, all of a sudden we have a parallel algorithm that is slower than its serial counterpart.

3 Scan Operation

The scan operation gives all *prefix sums* of an array.
Definition: *Given an array*

$$[a_0, a_1, a_2, \ldots, a_{n-1}]$$

[1]There is real hardware that works this way, namely the pixel plane machine from UNC.

and a binary associative operator $\oplus$ *(for example: add, multiply, matrix multiply, max, min, ...), return all partial sums*

$$[a_0, (a_0 \oplus a_1), (a_0 \oplus a_1 \oplus a_2), \ldots, (a_0 \oplus a_1 \oplus \cdots \oplus a_{n-1})].$$

This is obviously done in $O(n)$ time by a serial algorithm. Simply apply the operation repeatedly to each element, keeping a running sum, and return the partial results. Thus, the algorithm has a "serial flavor"—but it can be done efficiently in parallel.

The scan operation is often used in parallel algorithm design. If the algorithm seems serial, but we can utilize a scan operation, then it can be done efficiently in parallel.

3.1 Parallel implementation of a simpler operation

To see how a scan is done in parallel, we will first look at a simpler operation: finding the last value in a scan (i.e., the sum of all the values in an array). It can be done in $O(\log n)$ time by arranging all the processors into a tree of $\log n$ depth (see Figure 2).

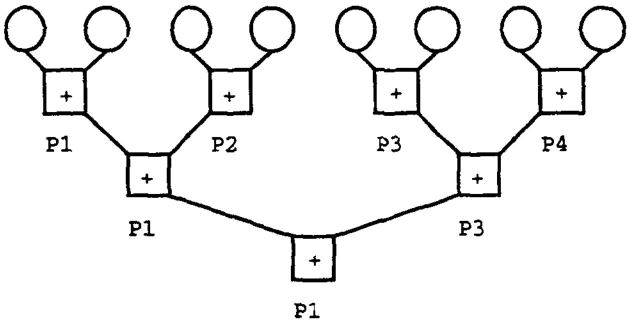


Figure 2: The sum of all values in an array

This is not work efficient, however, because the number of processors needed is $n/2$ (one for each pair of leaves). This gives $O(n \log n)$ total work. The serial algorithm takes $O(n)$ time, so $W_p(n) \neq O(T_s(n))$.

Can it be made work efficient? Yes! Use $n/\log n$ processors instead of $n/2$ processors. Have each processor do $\log n$ serial operations, then combine those values in a tree. The first step takes $\log n$ time since each processor adds $\log n$ numbers. The combining step takes as long as the depth of the tree, or $\log P$ time, where P is the number of processors. Thus, $T_p(n) = \log n + \log P = O(\log n)$, since $\log n$ grows faster than $\log(n/\log n)$. The total work, $P \times T_p(n) = (n/\log n) \times O(\log n) = O(n)$, the same as the serial case, so this is a work-efficient algorithm.

In summary, if an algorithm does not have enough parallelism to use one processor per element, using less processors can lead to a work efficient algorithm.

3.2 Parallel scan operation

Can we use the trick of having each processor do some local work to make the scan operation be work efficient in parallel?

Multiple trees—a naive approach

How about n trees, each one smaller than the last? They could be arranged in a triangle, with the partial sums broadcast to all other trees lower in the triangle (see Figure 3).

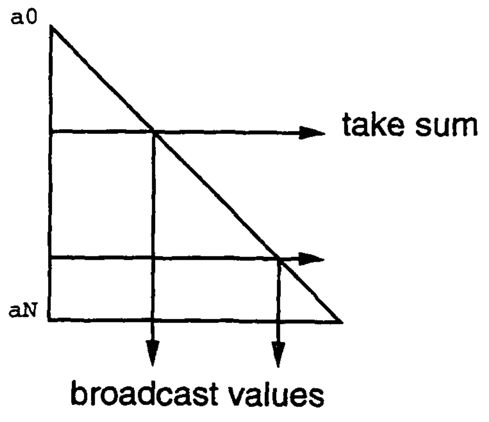


Figure 3: Multiple trees

To do this, we need about $n^2/2$ processors. Each line going across is a separately computed sum. Using the local work trick, each sum can be done with $n/\log n$ processors in $O(\log n)$ time. Therefore, we need a total of $O(n^2/\log n)$ processors. The scan can be computed in $O(\log n)$ time, but the total work is $O(n^2)$, so this is not work efficient.

Two-phase local work algorithm

Use $\sqrt{n}$ processors:

1. Each processor sums $\sqrt{n}$ values locally ($O(\sqrt{n})$ time).
2. Scan across by having each processor tell the neighbor to its right its sum added to the value from the neighbor to its left ($O(\sqrt{n})$ time).
3. Each processor does a local scan, starting with the offset it received from the neighbor to its left ($O(\sqrt{n})$ time).

The work here is $p \times T_p(n) = \sqrt{n} \times 3\sqrt{n} = O(n)$. However, we are doing a constant factor more work than the serial algorithm, because we are doing more than twice as many add operations: n during the first local phase, $\sqrt{n}$ for the scan across processors, and n during the second local phase. (It can be proven that we can't do better than $3n/2$ operations.) Since we are doing about twice the number of operations, the best we can expect is a speedup of 500 times if 1000 processors are used.

Can we do better? That is, can we solve the problem with the same amount of work, but with more processors, and therefore in less time? How about reducing the amount of local work to $n^{1/4}$, thereby increasing the number of processors to $n^{3/4}$. Then the two local work phases would each take $O(n^{1/4})$, the scan phase would take $O(n^{3/4})$, and the work would be $O(n\sqrt{n})$. Note that the original algorithm was balanced, and the new algorithm will not be work efficient if the times for the various phases are altered.

However, the scan step in the middle can be done in the same manner, i.e., have some processors do local work, and then do a linear combination of values. This can be extended recursively into a full tree, using a technique called *tree summation.*

Tree summation

A scan can be performed using processors arranged in a tree. The scan is done in two steps; first go up the tree collecting partial sums, and then go down the tree distributing the partial sums. The tree summation will be illustrated for a *prescan.* A prescan is similar to a scan operation, except that the resulting vector of length n is

$$[I, a_0, (a_0 \oplus a_1), \ldots, (a_0 \oplus \ldots a_{n-2})]$$

where I is the identity element for the operation being performed. It is the scan operation "shifted right by one". The scan can easily be computed in constant time from the prescan by dropping the first element (the identity), and adding an element at the end which is the sum of the last element of the prescan and a_{n-1}. Figure 4 illustrates the computation of a prescan using tree summation.

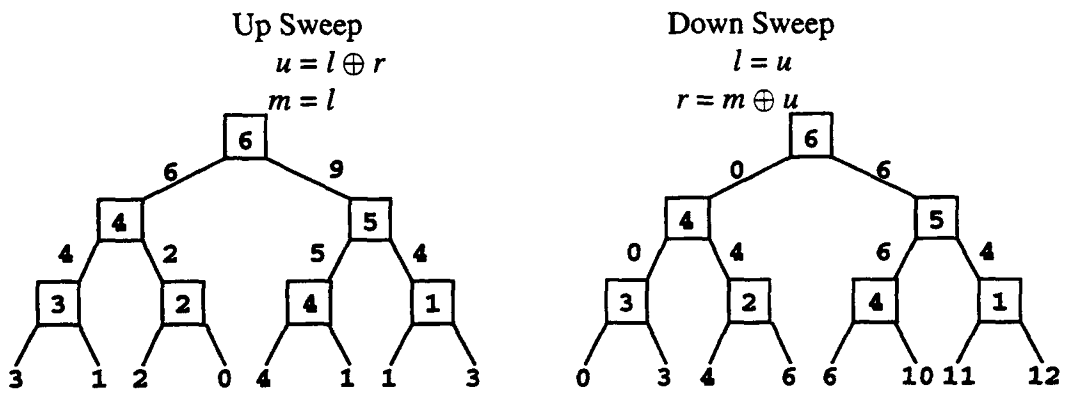


Figure 4: Prescan using tree summation

Combined with the local work trick using $n/\log n$ processors, a scan can now be done in $O(\log n)$ time, which gives $O(n)$ work. This algorithm is therefore work efficient, and takes less time than the two-phase local work algorithm alone.

4 Uses Of The Scan Operation

- Array Compression.

  Given an array of values

$$[a_0, a_1, a_2, a_3, a_4, a_5, a_6, a_7, a_8, a_9, \ldots, a_n]$$

and an array of flags

$$[1, 0, 1, 1, 0, 0, 0, 1, 1, 1, 0, \ldots]$$

compress the flagged elements to the front of the array:

$$[a_0, a_2, a_3, a_7, a_8, a_9, \ldots].$$

This could be used in e.g. quicksort, where the elements which we want to pivot are marked with a 1 in the flag array. If we prescan the flags, we are left with array indices in the positions corresponding to 1's in the flag array. In this example, a prescan gives:

$$[0, 1, 1, 2, 3, 3, 3, 3, 4, 5, 6, \ldots]$$

So a_0 goes in position 0, a_2 goes in position 1, a_7 goes in position 3, and so on.

Since compression (packing of an array) uses the prescan operation, it can be done using $n/\log n$ processors in $O(\log n)$ time.

- Load Balancing.

  Array compression can in turn be used to load balance jobs on a bunch of processors. Use a flag array with 1's for jobs that are still active. Compress this array to find all those jobs, and then redistribute it among the processors. There can be multiple jobs per processor, both before and after load balancing.

- Others. There are lots, but time is short...

5 Segmented Scan

Another very useful operation is the *segmented scan*. In addition to the array which the scan is performed on, there is a companion flag array. Whenever there is a 1 in the flag array, reset the sum to the identity element. For example,

$$\begin{array}{lccccccll}
[& a_0 & a_1 & a_2 & a_3 & a_4 & a_5 & \ldots &] \\
[& 0 & 0 & 1 & 0 & 0 & 1 & \ldots &] \\
\hline
[& a_0 & (a_0 \oplus a_1) & a_2 & (a_2 \oplus a_3) & (a_2 \oplus a_3 \oplus a_4) & a_5 & \ldots &]
\end{array}$$

This too can be done in parallel using $n/\log n$ processors in $O(\log n)$ time.

6 For More Information

Complete details of the scan operation, its implementation and uses can be found in Blelloch [5].

This lecture was a demonstration of NESL projected onto a screen. These notes are just the transcripts from the demonstration, updated to reflect the new ML-style syntax of NESL.

1 Loading NESL

```
Allegro CL 3.1.12.2 [DECstation] (11/19/90)
Copyright (C) 1985-1990, Franz Inc., Berkeley, CA, USA
<cl> (load "/afs/cs/project/scandal/nesl/current/load")

; Loading /afs/cs/project/scandal/nesl/current/load.lisp.

; Loading NESL version 2.6.2, Feb. 20, 1993

; -> You can switch to Lisp by typing lisp();
; -> and back to Nesl by typing (nesl).
; -> Use (nesl) when an error aborts you out of the interpreter.
; -> Type help(); for a list of the top level commands.

.....
```

2 Scalar Operations

```
<Nesl> 2 * (3 + 4);

Compiling VCODE....
Writing VCODE file....
Starting VCODE process locally...
Start of interpretation...
End of interpretation...
Reading VCODE output....
14 : INT
<Nesl> verbose();

Verbose: Off
<Nesl> (2.2 + 1.1) / 5.0;

0.66 : FLOAT
<Nesl> t or f;

T : BOOL
<Nesl> `a < `d;

T : BOOL
<Nesl> 1.6 + 7;
```

```
Error at top level.
For function + in expression
  1.6 + 7
inferred argument types don't match function specification.
  Argument types: FLOAT,INT
  Function types: A,A :: A IN NUMBER

<Nesl> 1.6 + float(7);

8.6 : FLOAT
<Nesl> round(1.6) + 7;

9 : INT
<Nesl> sin(.6);

0.564642473395035 : FLOAT
<Nesl> a = 4;

4 : INT
<Nesl> a + 5;

9 : INT
<Nesl> if (4 < 5) then 11 else 12;

11 : INT
<Nesl> let a = 3 * 4
       in a + (a * 1);

24 : INT
<Nesl> let a = 3 * 4;
           b = a + 5
       in a + b;

29 : INT
<Nesl> function fact(i) =
         if (i == 1)
         then 1
         else i * (fact(i-1));

FACT(I) : INT -> INT
<Nesl> fact(5);

120 : INT
<Nesl> pi = 3.14159;
```

```
3.14159 : FLOAT
<Nesl> function circarea(r) = pi * r * r;

CIRCAREA(R) : FLOAT -> FLOAT
<Nesl> circarea(3.0);

28.27431 : FLOAT
<Nesl> (2, `a);

(2, `a) : INT,CHAR
<Nesl> function div_rem(a, b) = (a / b, rem(a, b));

DIV_REM(A,B) : INT,INT -> INT,INT
<Nesl> div_rem (20, 6);

(3, 2) : INT,INT
```

3 Vector Operations

```
<Nesl> [2, 5, 1, 3];

[2, 5, 1, 3] : [INT]
<Nesl> "this is a vector";

"this is a vector" : [CHAR]
<Nesl> [(2, 3.4), (5, 8.9)];

[(2, 3.4), (5, 8.9)] : [(INT,FLOAT)]
<Nesl> ["this", "is", "a", "nested", "vector"];

["this", "is", "a", "nested", "vector"] : [[CHAR]]
<Nesl> [2, 3.0, 4];

Error at top level.
For function SEQ_SNOC in expression
  [2,3.0]
inferred argument types don't match function specification.
  Argument types: [INT],FLOAT
  Function types: [A],A :: A IN ANY

<Nesl> {a + 1: a in [2, 3, 4]};

[3, 4, 5] : [INT]
<Nesl> let a = [2, 3, 4] in {a + 1: a};
```

```
[3, 4, 5] : [INT]
<Nesl> {a + b: a in [2, 3, 4]; b in [4, 5, 6]};

[6, 8, 10] : [INT]
<Nesl> let a = [2, 3, 4]; b = [4, 5, 6] in {a + b: a; b};

[6, 8, 10] : [INT]
<Nesl> {a == b: a in "this"; b in "that"};

[T, T, F, F] : [BOOL]
<Nesl> {fact(a): a in [1, 2, 3, 4, 5]};

[1, 2, 6, 24, 120] : [INT]
<Nesl> {div_rem(100, a): a in [5, 6, 7, 8]};

[(20, 0), (16, 4), (14, 2), (12, 4)] : [(INT,INT)]
<Nesl> sum([2, 3, 4]);

9 : INT
<Nesl> dist(5, 10);

[5, 5, 5, 5, 5, 5, 5, 5, 5, 5] : [INT]
<Nesl> [2:50:3];

[2, 5, 8, 11, 14, 17, 20, 23, 26, 29, 32, 35, 38, 41, 44, 47] : [INT]
<Nesl> "big" ++ " boy";

"big boy" : [CHAR]
<Nesl> {x in "wombat" | x <= `m};

"mba" : [CHAR]
<Nesl> {sum(a): a in [[2, 3, 4], [1], [7, 8, 9]]};

[9, 1, 24] : [INT]
<Nesl> bottop("testing");

["test", "ing"] : [[CHAR]]
<Nesl> partition("break into words", [5, 5, 6]);

["break", " into", " words"] : [[CHAR]]
<Nesl> function my_sum(a) =
         if (#a == 1) then a[0]
         else
           let res = {my_sum(x): x in bottop(a)}
           in res[0] + res[1];
```

```
MY_SUM(A) : [A] -> A :: A IN NUMBER
<Nesl> my_sum([7, 2, 6]);

15 : INT
```

4 Loading Program Files And Getting Help

```
<Nesl> load("/afs/cs/project/scandal/nesl/current/examples/median.cnesl");

; CNesl loading /afs/cs/project/scandal/nesl/current/examples/median.cnesl.
<Nesl> median([8, 1, 2, 9, 5, 3, 4]);

4 : INT
<Nesl> median("this is a test");

`i : CHAR
<Nesl> describe(++);

INTERFACE:

v1 ++ v2

TYPE:
 [A],[A] -> [A] :: A IN ANY

DOCUMENTATION:

 Given two sequences, \fun{++} appends them.

<Nesl> help();

NESL toplevel forms:
  function name(arg1,..,argn) [: typespec] = exp;
                                                 -- Function Definition
  datatype name(t1,..tn) [:: typebind];          -- Record Definition
  name = exp;                                    -- Toplevel Assignment
  exp;                                           -- Any NESL expression

Toplevel Commands:
  DESCRIBE(funname);     -- Describe a NESL function with funname.
  LOAD(filename);        -- Load a file.
  VERBOSE();             -- Toggle the verbose switch.
  SET_PRINT_LENGTH(n);   -- Set the maximum number of elements that
                            are printed in a sequence.
  LISP(); or EXIT();     -- Exit NESL and go to the Lisp interpreter.
  HELP();                 -- Print this message.
  BUGS();                 -- Lists the known bugs.

Commands for running VCODE on remote machines:
  CONFIGURATION();        -- List the properties of the current
                             configuration.
  USE_MACHINE(config);    -- Use the machine with configuration CONFIG.
  LIST_MACHINES();        -- List the available configurations.
  SET_MEMORY_SIZE(n);     -- Set the memory size of the current
                             configuration.
   name &= exp [,MEM := exp] [,MAXTIME := exp];
                            -- Executes exp in the background
```

5 An Example: String Searching

```
<Nesl> teststr = "string small strap asop string";

"string small strap asop string" : [CHAR]
<Nesl> candidates = [0:#teststr-5];

[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18,
 19, 20, 21, 22, 23, 24] : [INT]
<Nesl> {a == `s: a in teststr -> candidates};

[T, F, F, F, F, F, F, T, F, F, F, F, F, T, F, F, F, F, F, F, T, F,
 F, F, T] : [BOOL]
<Nesl> candidates = {c in candidates;
                     a in teststr -> candidates | a == `s};

[0, 7, 13, 20, 24] : [INT]
<Nesl> candidates = {c in candidates;
                     a in teststr -> {candidates+1:candidates}
                     | a == `t};

[0, 13, 24] : [INT]
<Nesl> candidates = {c in candidates;
                     a in teststr -> {candidates+2:candidates}
                     | a == `r};

[0, 13, 24] : [INT]
<Nesl> candidates = {c in candidates;
                     a in teststr -> {candidates+3:candidates}
                     | a == `i};

[0, 24] : [INT]
```

```
<Nesl> candidates = {c in candidates;
                     a in teststr -> {candidates+4:candidates}
                     | a == `n};

[0, 24] : [INT]

<Nesl> function next_cands(cands, w, s, i) =
         if (i == #w) then cands
         else
           let letter = w[i];
               next_chars = s -> {cands + i: cands};
               new_cands = {c in cands; l in next_chars | l == letter}
           in next_cands(new_cands, w, s, i + 1);

NEXT_CANDS(CANDS,W,S,I) : [INT],[A],[A],INT -> [INT] :: A IN ORDINAL
<Nesl> function string_search(w, s) =
         next_cands([0:#s - (#w - 1)], w, s, 0);

STRING_SEARCH(W,S) : [A],[A] -> [INT] :: A IN ORDINAL
<Nesl> longstr =
"This course will be a hands on class on programming parallel
algorithms.  It will introduce several parallel data structures and a
variety of parallel algorithms, and then look at how they can be
programmed.  The class will stress the clean and concise expression of
parallel algorithms and will present the opportunity to program
non-trivial parallel algorithms and run them on a few different
parallel machines.  The course should be appropriate for graduate
students in all areas and for advanced undergraduates.  The
prerequisite is an algorithms class.  Undergraduates also require
permission of the instructor.";

"This course will be a hands on class on programming parallel
algorithms.  It will introduce several parallel data structures and a
variety of parallel algorithms, and then look at how they can be
programmed.  The class will stress the clean and concise expression of
parallel algorithms and will present the opportunity to program
non-trivial parallel algorithms and run them on a few different
parallel machines.  The course should be appropriate for graduate
students in all areas and for advanced undergraduates.  The
prerequisite is an algorithms class.  Undergraduates also require
permission of the instructor." : [CHAR]
<Nesl> string_search("will", longstr);

[12, 77, 219, 291] : [INT]
<Nesl> string_search("student", longstr);
```

```
[461] : [INT]
```

6 Exiting

```
<Nesl> exit();

NIL
<cl> (exit)
; Exiting Lisp
```

15-840: Programming Parallel Algorithms — Lecture #4
Scribe: Andrzej Filinski — Tuesday, 29 Sep 92

Overview

Today's topics:

- NESL notes.
- Models of parallelism and mappings between them.

1 NESL Notes

Some useful hints for running NESL:

1. Do not generate giant results (e.g., `primes(200000)`). The NESL top-level loop works as follows: the compiler, running in LISP, reads a NESL expression and generates a VCODE file. This file is executed by another process (possibly even running on a different machine), and the result is written to an output file. Finally, the output file is read back into NESL and the result is displayed. It could easily happen that reading a large result takes longer than actually computing it.

2. If you do need to generate giant results, you may find the following built-in operations useful:

```
write-object-to-file(<object>, "filename")
read-object-from-file(<type>, "filename")
```

   See the NESL manual [7] for further details.

2 Models Of Parallelism

We will consider a series of increasingly abstract levels of parallelism, each building on top of the previous one (see Figure 1). There exist simulations or mappings with guaranteed running times from every level to the one below.

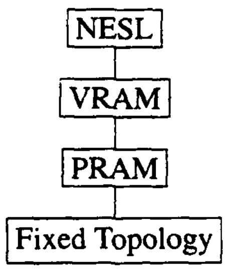


Figure 1: A hierarchy of models of parallelism

Why not program for the lowest level directly? Our goal is a machine-independent notion of parallelism. In particular, since we don't know what architectures and network topologies will be popular in the future, we want to express parallel algorithms in such a way that they can be easily mapped onto a wide variety of machines.

3 Mapping PRAM Onto A Fixed Topology

A general fixed-topology network consists of p processors, connected by some set of wires (usually arranged in a regular pattern). One could easily teach an entire course about this subject alone. See Leighton [19] for further information.

Common examples of topologies include the bus, ring, grid, toroidal mesh, and hypercube networks. Perhaps less well-known are cube-connected cycles, fat trees, tree-of-meshes, butterfly, omega, Banyan, De Bruijn, and shuffle-exchange networks. Many of these are actually equivalent, modulo node numbering. And many of the different classes of networks can still be efficiently mapped onto each other.

Real (i.e., commercial) implementations exist for most of these topologies. Sometimes a network topology was motivated by or derived from a particular algorithm, sometimes the other way round.

3.1 The butterfly network

We will consider a particular fixed topology, the *butterfly network*, but the principles apply to most other topologies as well.

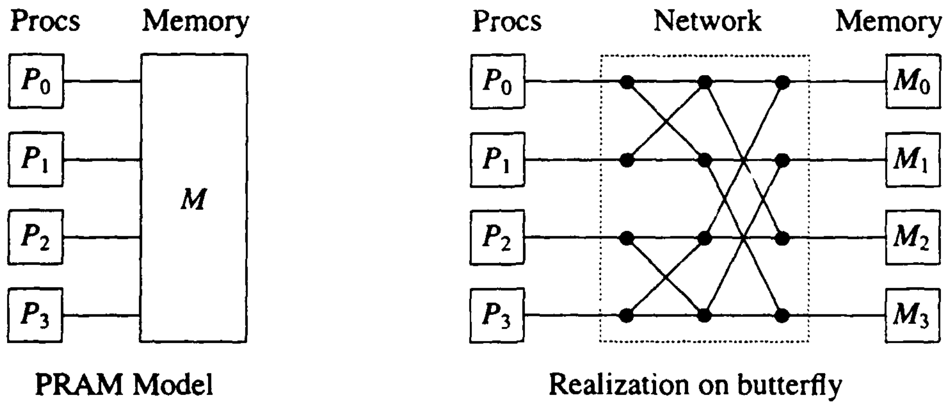


Figure 2: Implementation of a PRAM on a butterfly network

The butterfly network (so called because of the repeated ⧖-patterns) connects a set of processors to a set of memory banks through a series of simple 2×2 switching nodes. At level 0, the nodes are a distance of 2^0 apart; at level 1, the distance is 2^1, and so on. Thus, at each level we can either invert the corresponding address bit or leave it alone, so there is a $\log n$ route from every processor to every memory bank.

For simplicity, the example in Figure 2 has the same number of memory banks as processors. We could easily have more banks if we added some kind of fanout network at the end.

Although every processor can communicate with every memory bank, they may not be able to do so at the same time. In other words, we cannot set up an arbitrary permutation without either adding queues to the switches or possibly requiring two different communications to go over the same wire. If we add an *inverted* butterfly network following the first one, this becomes possible; such a configuration is called a Beneš network. It can also be done with 3 butterflies oriented in the same direction. It is still an open problem whether 2 unidirectional butterflies suffice.

3.2 Mapping PRAM onto butterfly

We will assume that sending a message across a wire takes unit time. Thus, with n processors, every memory access takes at least $O(\log n)$ time. However, we can overcome this latency by allocating several *virtual* PRAM processors to every butterfly processor, so that we can still do useful work while waiting for the memory. Specifically, we use the following strategy:

1. Assign $\log n$ virtual processors to a physical processor, executing one instruction of each every $\log n$ cycles.

2. Pipeline memory accesses, so that a new request can be issued on every cycle and satisfied $\log n$ cycles later.

We still have a problem, however, if several physical processors need to access the same memory bank at the same time (even assuming an EREW model, different addresses may well reside in the same bank). To prevent this, we add:

3. Randomly hash memory locations, so that there is no direct correlation between a memory address and the bank that it is assigned to.

It can be shown that this reduces conflicts for memory banks to a very low level. Hashing also serves to eliminate "hot spots" *within* the network, i.e., bottleneck switches or wires. For a full discussion, see Leighton [19].

3.3 Cost of the simulation

Summarizing, we can simulate a PRAM program using P processors and running in time T,

$$P_{PRAM} = n \log n, \qquad T_{PRAM} = t$$

on a butterfly network with

$$P_{BF} = n, \qquad T_{BF} = t \log n.$$

with high probability. In both cases the work is $nt \log n$, so the simulation is work efficient (i.e., no physical processor is wasting time). However, for n processors we need $n \log n$ switches in the network, so the number of switches grows superlinearly with the number of processors. Fortunately, switches are typically much simpler than processors, so this is not a problem in practice. A bigger problem is the large number of wires required to build a butterfly network.

More generally, we have the following equation:

$$T_{BF}(T_{PRAM}, P_{PRAM}, P_{BF}) = kT_{PRAM} \times \left(\frac{P_{PRAM}}{P_{BF}} + \log P_{BF}\right)$$

which gives us the running time on a butterfly network given T_{PRAM}, P_{PRAM}, and P_{BF}. Thus, as long as the first summand is dominant (i.e., if we simulate $\log P_{BF}$ or more PRAM processors on every BF processor), the mapping is work efficient.

Examples of butterfly-based machines include the BBN Butterfly (no longer made) and the Cray. The newest model of the latter, the C-90, has 16 processors and 1024 memory banks. Each processor has vector registers with 128 elements in each, so it can generate 128 independent memory addresses at a time. The latency of the communication network and the memory access itself takes up to 25 cycles, but once the actual transfer begins, throughput is 1 value per cycle. Thus, as long as the processors can keep their vector registers filled, they never have to sit idle waiting for memory.

The Crays do not currently do random hashing. In some cases this is indeed a problem, so programmers try hard to avoid bank conflicts. However, a hashing mechanism could probably be added with relatively simple hardware, and Cray may well be looking into this.

4 Mapping VRAM Onto PRAM

4.1 The VRAM model

A VRAM machine consists of the following components:

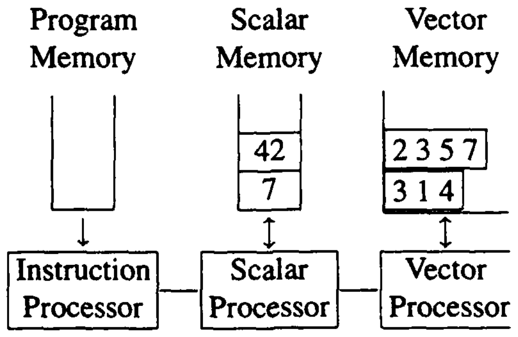


Figure 3: A VRAM machine

If we remove the vector memory and processor, this is just an ordinary RAM machine. Note that vectors need not all be the same length. The VRAM differs from the PRAM in that the allocation of parallel computations to processors is no longer part of the program. Conceptually, there is only a single vector processor with a set of *vector instructions*:

- Elementwise operations (e.g., add two vectors of equal length).
- Data movement (indirect addressing, especially permutations).
- Sums, scans, etc.

Simulating a SIMD PRAM on a VRAM is trivial: use one giant vector M for the shared memory, and a vector P for per-processor data.

4.2 Complexity on a VRAM

For a problem of size n, we introduce two measures:

1. Step complexity, written $S(n)$, is defined as the total number of instructions (scalar or vector) executed while running the program.

Step complexity gives a useful indication of the "degree of parallelism" exhibited by an algorithm, but does not by itself give enough information for a useful model. In fact, it can be shown that in a complexity model that doesn't take account of how much work each vector instruction does, $\mathcal{P} = \mathcal{NP}$ (see Pratt [25]). This is because we can essentially simulate nondeterminism by parallelism. We need to add:

2. Work complexity, $W(n) = \sum_{i=1}^{S} l_i$, where l_i is the length of the vector(s) involved at step i. This is a natural measure of how much work would be required to simulate the execution using only a single processor, just as $P \times T$ is in the PRAM model.

4.3 An example

Consider adding up all elements of a vector of length n. Let us assume that vector summation is not a primitive of our VRAM machine, but that we do have the following instructions available:

- Elementwise add.
- Operations for extracting all the odd- and even-numbered elements of a vector, giving two vectors of half the length.

Let us also assume that n is a power of 2. Then there is a very simple algorithm for computing vector sums: add all pairwise adjacent elements of the vector to obtain a vector of size $n/2$; repeat $\log n$ times to get the final result (a vector of size 1). The complexities are:

$$\begin{aligned} S(n) &= O(\log n) \\ W(n) &= O\left(\sum_{i=1}^{\log n} \frac{n}{2^i}\right) = O(n) \end{aligned}$$

In fact, the algorithm uses exactly the same number of additions as the obvious serial one.

If we want to map this algorithm onto a PRAM with p processors ($p \leq n$), each pair-summation step can be directly executed in parallel. To simulate one step of the algorithm on a vector of length m will require $O(\lceil \frac{m}{p} \rceil)$ time, since we can divide the vector evenly among the processors, and each processor can loop over its elements. This gives a total running time for the algorithm of

$$T_{PRAM} = O\left(\sum_{i=1}^{\log n} \left\lceil \frac{n}{2^i p} \right\rceil\right) = O\left(\sum_{i=1}^{\log n} \left(\frac{n}{2^i p} + 1\right)\right) = O(n/p + \log n)$$

which is work efficient if $p \leq n/\log n$. Note that the VRAM summation algorithm did not have to worry about explicitly assigning work to processors—the mapping from one model to the other takes care of this in a uniform way.

4.4 Cost of the simulation

In general, we can compute the running time of a VRAM algorithm on a PRAM machine using *Brent's Theorem*, also known as Brent's Lemma, since its proof is so simple [9]:

$$\begin{aligned} T_{PRAM}(W,S,p) &= O\left(\sum_{i=1}^{S} \left\lceil \frac{l_i}{p} \right\rceil\right) \\ &= O\left(\sum_{i=1}^{S} \left(\frac{l_i}{p} + 1\right)\right) \\ &= O\left(\frac{1}{p}\left(\sum_{i=1}^{S} l_i\right) + S\right) \\ &= O\left(W/p + S\right) \end{aligned}$$

This assumes that we can execute each vector instruction in time $\lceil l_i/p \rceil$, but a scan or a sum actually needs $\lceil l_i/p \rceil + \log p$ time. Thus, if we include scans and sums as VRAM primitives, the complexity increases to

$$\begin{aligned} T_{PRAM}(W,S,p) &= O\left(\sum_{i=1}^{S} \left(\left\lceil \frac{l_i}{p} \right\rceil + \log p\right)\right) \\ &= O(W/p + S\log p). \end{aligned}$$

As usual, we only need to make first the term dominate (i.e., use "few enough" processors) to obtain a work-efficient algorithm.

5 Mapping NESL Onto VRAM

The final abstraction step goes from VRAM programs to NESL code. In the VRAM model, vectors can contain only scalar values. In NESL, however, we also allow *nested parallelism*: vector components can themselves be vectors, possibly even of different size. And we can now execute *vector* instructions in parallel, as if we had an unlimited number of vector processors in the VRAM model. For example, we can compute the sums of all components in a vector of vectors,

```
{sum(x): x in [[2, 3, 4], [5, 6], [7, 8, 9]]} ⇒ [9, 11, 24]
```

in a *single* step.

The step complexity of such a nested vector operation is defined as the *maximum* of the individual step complexities; the work complexity is the *sum* of the work complexities. Again, this reflects our intuitive interpretation of these measures.

The actual mapping of NESL code onto a VRAM is somewhat involved, but relies on representing all nested vectors in flattened form, i.e., as a vector of scalars, in accordance with the VRAM model. Nested parallelism is then realized in terms of *segmented scans*, which (to within a constant factor) are as cheap as ordinary ones. Thus, the complexity of the composite mapping of NESL code—even if that code does not explicitly contain any scan operations—onto a PRAM is still

$$T_{PRAM}(W,S,p) = O(W/p + S\log p).$$

But if we have scans as primitive constant-time operations in our PRAM model (which may be reasonable if the mapping from PRAM onto a fixed topology allows us to do scans at "no extra cost"), the complexity falls back to $O(W/p + S)$.

6 For More Information

For a general discussion of models of parallelism, and the specifics of the scan-based model, see Blelloch [6].

15-840: Programming Parallel Algorithms — Lecture #5
Scribe: Jonathan Hardwick — Thursday, 1 Oct 92

Overview

- Clarifications of Assignment 2 and last lecture.
- Different approaches to the primes problem (Assignment 1).
- Thinking in parallel:
  - Recursively solving smaller problems.
  - Nested parallelism.
- Calculating work and step complexity in NESL.
- A problem to think about.

1 Clarifications

- `odd_elts`, `even_elts` and `interleave` all have $S(n) = O(1)$ and $W(n) = O(n)$. For example, this NESL implementation of `even_elts` has the correct complexity:

```
function even_elts(a) = a -> [0:#a:2]
```

- In the last lecture, the method to map an algorithm from a PRAM onto a butterfly only gives us a probabilistic upper bound on the time.

2 The Primes Problem Revisited

The classic serial algorithm is:

```
for $i = 0$ to $n$ do $primes[i] = 1$
for $i = 2$ to $\sqrt{n}$ do
  if $primes[i] = 1$ then
    for $j = 2i$ to $n$ by $i$ do
      $primes[i] = 0$
```

Note that this has both inner and outer loops. The optimal parallel solution must parallelize both of these loops.

2.1 Approach 1 : Parallelize the inner loop

A naive solution is to parallelize only the inner loop, dividing the vector evenly amongst the processors. A master processor executes the outer loop, broadcasting the variable i. Each processor then knocks out the multiples of i that fall within its section of the vector. The time taken by this algorithm is composed of the time for the outer serial loop (which is $O(\sqrt{n})$), and the time taken to do the inner sieve for each prime in parallel on P processors (which is $O(\sum_{i<\sqrt{n}, i\in pr}\lceil\frac{n}{Pi}\rceil)$). So we have:

$$\begin{aligned}
T_p(n) &= O\left(\sqrt{n} + \sum_{i<\sqrt{n}, i\in pr} \left\lceil \frac{n}{Pi} \right\rceil\right) \\
&= O\left(\sqrt{n} + \sum_{i<\sqrt{n}, i\in pr} \frac{n}{Pi} + \sum_{i<\sqrt{n}, i\in pr} 1\right) \\
&= O\left(\sqrt{n} + \frac{n \log\log n}{P} + \frac{\sqrt{n}}{\log n}\right)
\end{aligned}$$

The last term represents the "wasted work" of the algorithm; it is smaller than either of the other two terms. To minimize T, pick P so that the other two terms are equal, i.e., $P = \sqrt{n}\log\log n$, giving $T = O(\sqrt{n})$.

2.2 Approach 2 : Recurse on a subset

Note that we can rewrite the outer loop in terms of a recursion on a smaller subset of the problem:

```
sprimes = primes($\sqrt{(n)}$)
for i = 1 to #sprimes do
  for j = 2× primes[i] to n by sprimes[i] do
    primes[j] = 0
```

This changes the running time to

$$T(n) = O\left(\frac{\sqrt{n}}{\log n} + \frac{n \log\log n}{P} + \frac{\sqrt{n}}{\log n}\right)$$

so we can reduce T to $O(\sqrt{n}/\log n)$ by increasing P to $\sqrt{n}\log n \log\log n$.

2.3 Approach 3 : The $O(\log n)$ time version

We know that the primes algorithm will generate sieves of $n/2$ elements, $n/3$ elements, $n/5$ elements, etc. Ideally, we want to be able to generate and apply the sieves simultaneously, i.e., parallelize both loops. There are a total of $n \log\log n$ sieve elements, and we want to make each processor responsible for $\log n$ of them. The tricky part is allocating processors—we solve this problem using segmented scans:

1. Recurse to find the primes less than $\sqrt{n}$.
2. Divide the primes into n to generate a vector of sieve lengths.

$$\begin{array}{|c|c|c|c|c|}\hline \frac{n}{2} & \frac{n}{3} & \frac{n}{5} & \frac{n}{7} & \cdots \\ \hline\end{array}$$

3. Do a plus-prescan of this vector, giving pointers to the block start positions.

$$\begin{array}{|c|c|c|c|c|}\hline 0 & \frac{n}{2} & \frac{n}{2}+\frac{n}{3} & \frac{n}{2}+\frac{n}{3}+\frac{n}{5} & \cdots \\ \hline\end{array}$$

4. Send the vector

$$\begin{array}{|c|c|c|c|c|}\hline 2 & 3 & 5 & 7 & \cdots \\ \hline\end{array}$$

to the pointed-to positions in a "sieve" vector, and mark these positions as segment starts as well.

5. Do a segmented copy-scan on the sieve vector.

$$\begin{array}{|c|c|c|c|c|c|c|c|c|c|c|c|}\hline 2 & 2 & 2 & \cdots & 3 & 3 & 3 & \cdots & 5 & 5 & 5 & \cdots \\ \hline\end{array}$$

6. Do a segmented plus-scan on the sieve vector, generating the multiples corresponding to our sieves.

$$\begin{array}{|c|c|c|c|c|c|c|c|c|c|c|c|}\hline 2 & 4 & 6 & \cdots & 3 & 6 & 9 & \cdots & 5 & 10 & 15 & \cdots \\ \hline\end{array}$$

7. Finally, use the elements of the sieve vector as addresses to send 0's to in the flag vector, and then pack using this flag vector into the destination vector.

The cost of scans on vectors of length $n \log\log n$ dominate this algorithm. The time taken is

$$\begin{aligned} T(n) &= O\left(\frac{n\log\log n}{P} + \log(n\log\log n) + T(\sqrt{n})\right) \\ &= O\left(\frac{n\log\log n}{P} + \log n\right) \end{aligned}$$

since the recursive calls take half as much time for each successive recursion, and hence do not affect the overall complexity. So we can pick $P = (n\log\log n)/\log n$, giving $T = O(\log n)$.

3 Important Tricks

There are two "tricks for thinking in parallel" to learn from this example:

- Recursively solving smaller problems. This is not necessarily divide-and-conquer; we may just be solving a single smaller problem (e.g., to solve the outer parallelism of primes).
- Nested parallelism. In NESL, this works even if the subproblems have different sizes, since we can represent the subproblems in terms of a segmented vector. Think back to the rendering example in the second lecture for another example of nested parallelism, where we had parallelism both within the objects (multiple pixels) and across the objects (multiple objects).

4 Calculating Work And Step Complexity In NESL

4.1 Primitives

By definition, all NESL primitives have a step complexity of $O(1)$, and a work complexity of $O(l)$, where l is the length of the vector. Note that the step complexity for primitives therefore becomes $O(W/P + S\log P)$ on a simple PRAM, or $O(W/P + S)$ on a PRAM with scans implemented in hardware. Also note that where a NESL primitive operates on two or more vectors of different lengths, the step complexity may not always be obvious. For example, what is the step complexity for a `get` operation getting from a longer vector to a shorter one? The rule in this case is that l is the length of the shorter vector, by analogy to the work required for a serial implementation of the primitive. However, there is an exception for the `put` operation; if it is putting from a short vector to a longer one, and the longer vector is later re-used, NESL has to physically build a copy of the longer vector (since it is a functional language). In this case, the primitive has work complexity $O(L)$, where L is the length of the longer vector. If the longer vector is never re-used, the NESL compiler can operate in-place on it, and hence the primitive has $O(l)$ complexity.

4.2 Combining complexities

The rules for combining work and step complexities when operations are done in parallel should be intuitively obvious:

- Total work complexity is the sum of the work of the subcalls:

$$work = \sum_{i}^{m} w_i$$

- Total step complexity is the maximum work done by a subcall:

$$step = max(s_i).$$

An example where we need to analyze the step complexity of subcalls is a parallel quicksort.

4.3 Quicksort

Quicksort is a highly-parallelizable algorithm. By analogy to the performance of the serial implementation, we would hope to get $P(n) = n/\log n$ and $T(n) = O(\log^2 n)$ from a work-efficient parallel implementation. The algorithm is very simple:

1. Split the vector into two subvectors, containing those elements less than or equal to the pivot, and those elements greater than the pivot.
2. Recurse on the two subvectors.
3. Append the results from the two recursive calls, and return the result of the append.

This is indeed work efficient when implemented in NESL. The splitting operation is just a double pack, i.e., two primitive operations. Also, note the nested parallelism we can get on recursion—Figure 1 shows the outer parallelism gained by operating on multiple subvectors on each time step (vertically), and the inner parallelism within each subvector (horizontally):

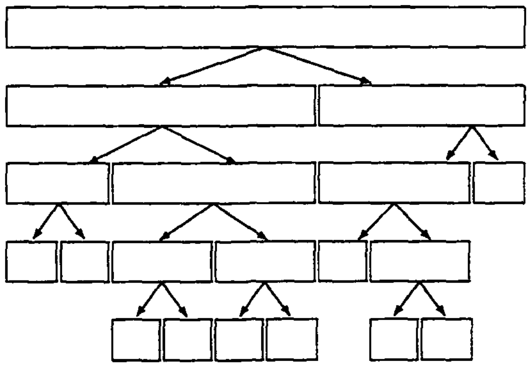


Figure 1: Behaviour of parallel quicksort over time

Best case analysis

We pick a perfect pivot (and hence divide the vector exactly in half) on each recursive call. The work of doing the split is $O(n)$, and the step complexity is $O(1)$. Thus we have:

$$\begin{aligned} W(n) &= 2W(n/2) + O(n) = O(n \log n) \\ S(n) &= S(n/2) + O(1) = O(\log n) \end{aligned}$$

which gives us the expected $T(n) = O(\log^2 n)$ when mapped to a PRAM with $n/\log n$ processors.

Worst case analysis

We pick the pessimal pivot, and divide the vector into 1 and $n-1$ elements, on each recursive call:

$$\begin{aligned} W(n) &= W(n-1) + O(n) \\ &= O(n^2) \\ S(n) &= S(n-1) + O(1) \\ &= O(n). \end{aligned}$$

Expected case analysis

Jájá [15, pages 466–469] uses randomized analysis to prove that, in the expected case, we get $W(n) = O(n \log n)$ and $S(n) = O(\log n)$ with very high probability.

5 A Problem

Given a vector of values such as $[7.3,3,5,9,8,2,3]$, remove the duplicates to form a set. This is probabilistically linear on a serial implementation using, e.g., hash tables; how can it be made work efficient on a parallel machine?

6 For More Information

Jájá [15] gives a detailed analysis of the quicksort algorithm.

Overview

- Operations on sets and sequences:
  - Removing duplicates
  - Naming problem
  - List ranking

1 Removing Duplicates

Given a sequence of items, the remove-duplicates algorithm removes all duplicates from the list, returning the resulting sequence. The order of the resulting sequence does not matter.

1.1 Approach 1 : Packing the range array

If the range of numbers is small, we can use an array equal in size to the range. We set a flag in the array for each number that appears in the input list, and then pack the range based on this array. This can be easily expressed in NESL:

```
function rem_duplicates(v) =
  let range = 1 + max_val(v)
  in {x in [0:range]; y in dist(f, range) <- {(i, t) : i in v} | y}
```

The obvious disadvantage of this approach is that it explodes when given a large range of numbers, both in memory and in the time taken to pack a large vector.

1.2 Approach 2 : Hashing

Using a hash table is a natural choice for the serial algorithm. A simple serial algorithm, with a good hash function and a hash table of approximate size $2n$ (of course, a prime hash size is best), can guarantee that with very high probability the bucket size will be less than $\log n$.

From here we can design a simple recursive parallel algorithm for removing duplicates. Assign a processor to each element in the input vector. We use buckets of size one, and hash all the items in parallel. For example, Figure 1 describes a particular hash function applied to the vector [69, 23, 91, 18, 42, 23, 18].

We assume a simultaneous-write architecture, which ensures that after two writes into the same memory location, one of the values will be correctly written. As a consequence, all but one of the values that map into the same bucket will be removed (42, 23 and 18 in our example). After all writes have completed, every processor reads back the value from the location in the hash table where it wrote its own value. If a different value is found, the element is kept for the next level of recursion. In our example, the only element left for the next level of recursion is 42. All other elements are added to the "no duplicates" list. This can be done by packing the hash table, i.e., ignoring the unassigned hash entries. Note that a different hash function has to be used for each

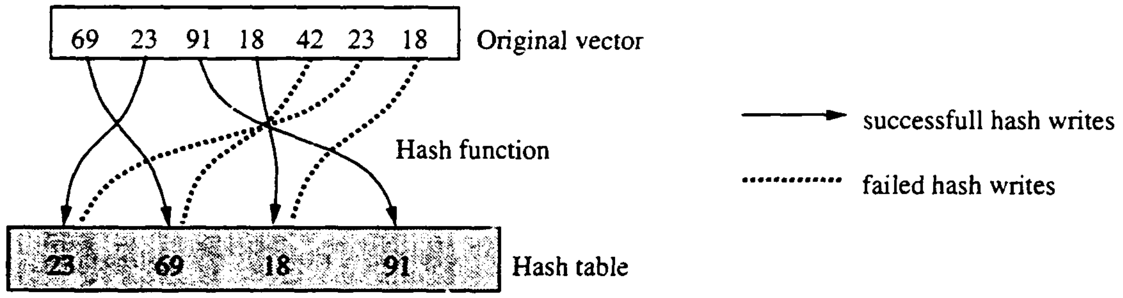


Figure 1: Parallel hashing

level of recursion to avoid repetitive hash collisions on each level of recursion. A good choice of hash function is $h(v) = v \bmod size_{hash}$.

With high probability, each level of recursion reduces the size of active elements by some constant fraction (see Figure 2), so we can conclude that $S(n) = O(\log n)$ and $W(n) = O(n)$.

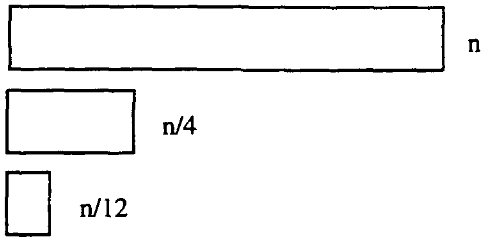


Figure 2: The reduction in problem size after each level of hash recursion

The remove-duplicates algorithm is frequently used for set operations—for instance, there is a trivial implementation of the set union problem given the remove-duplicates code. The set intersection problem, however, does not naturally fit into the remove-duplicates algorithm since in some sense we want to do the inverse of removing duplicates, i.e., we are looking for duplicates.

2 The Naming Problem

Removing duplicates can be trivially solved using the algorithm for the more general *Naming Problem.* Given a sequence of n values, we want to assign a unique name in the range $[0, n]$ for each value in the input vector. All elements with the same value get assigned the same name.

2.1 Approach 1 : Sorting

The naive approach is to sort the elements, but this is not normally work efficient since the work complexity of a comparison-based sorting algorithm is $W(n) = O(n \log n)$. However, if the range of input numbers is small we can use a radix sort, which has work complexity $W(n) = O(n + m)$ where m is the number of different elements. So for small m, the algorithm is close to linear in work.

2.2 Approach 2 : Hashing

When we have many different elements, we can use an adaptation of the remove-duplicates algorithm. The steps of the algorithm are:

1. Hash the keys.
2. Write to hash locations.
3. Read back the values, detecting any hash collisions.
4. Get the indices of collided keys.
5. Assign new labels to keys without any collisions.
6. Call recursively on collided keys.
7. Merge the collided keys with the "no collision" keys.

3 List Ranking

Given a linked list of elements, the list ranking algorithm returns a linked list whose elements have been assigned a number corresponding to their position in the sequence (see Figure 3).

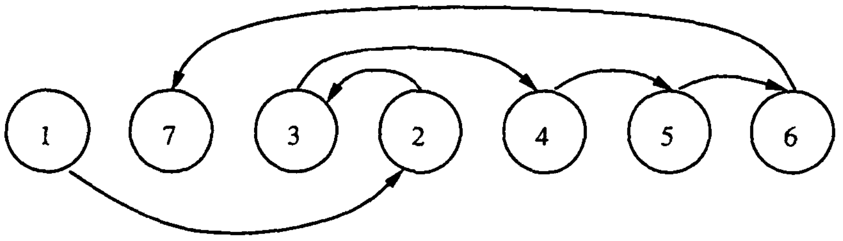


Figure 3: The linked list with assigned indices

3.1 Approach 1 : Recursive doubling

Assigning a processor for each element in the linked list we apply the following algorithm:

```
reverse pointers
$v(0) \leftarrow 0$
**repeat** $\log_2 n$ **times do**
  $v \leftarrow v + v(p)$
  $p \leftarrow p(p)$
```

$v(p)$ and $p(p)$ denote the counter and pointer at element p. For $\log_2 n$ steps each processor increases its counter v by the pointed item's value, and then sets its pointer to the pointed item's pointer. At iteration k of the loop, element i points to the item 2^k away from it, and contains the count of all elements between i and $i - 2^k$. After $\log_2 n$ steps, every element points at element 0, and contains the count of elements between i and $i - 2^{\log_2 i}$, which is its position in the list. Since every processor needs $\log_2 n$ steps, the work is $W(n) = O(n \log n)$. This is obviously not work efficient.

3.2 Approach 2 : Random mate

If the list is broken into two lists on each iteration, then $\log n$ iterations are sufficient for a list with n nodes, given n processors. If we could figure out how to subdivide the list into smaller lists, we could operate on the tree of subdivided lists and achieve $W(n) = O(n)$. Unfortunately, due to the inherently serial nature of linked lists, there does not seem to be an easy way to partition a linked list.

Instead of breaking the list in half, we can contract neighboring pairs to form a shorter list (i.e., create the tree bottom-up). One parallel algorithm based on list contraction consists of two phases. In the first phase (the computation phase) we create a sum-up tree by contracting the list. Pairs of elements are flagged for contraction using the following randomized algorithm:

```
$flag \leftarrow CoinFlip()$
**if** $flag = T$ **then**
    **if** $flag(p) = T$ **then**
        $flag \leftarrow F$
```

This flags a certain subset of the original sequence. Furthermore, it guarantees that neighbors are never flagged. Initially, the value d_i assigned as a label to the edges leaving the nodes is 1. When node i is taken out of the list, node $i+1$ is connected to node $i-1$ with an edge labeled $d_i + d_{i-1}$. In other words, we collect the partial sums upon element removal. Then we recursively subdivide the list at the flagged elements. For instance, let the original list be

$$[a, b, c, d, e, f, g, h]$$

and suppose that coin flipping throws out elements b and f at step 1 and elements c and g at step 2. At step 3, we are left with the vector

$$[a, d, e, h]$$

and after six such steps, we are left with the tree shown in Figure 4.

In the second phase (the expansion phase) the tree is traversed downwards in the usual prescan manner, calculating the following at each step:

$$rank\ (L(v)) = rank\ (v)$$
$$rank\ (R(v)) = sum\ (L(v)) + rank\ (v)$$

We now analyze the complexity of this algorithm by determining the tree depth. Since the probability of an element dropping out is $\frac{1}{2}\frac{1}{2} = \frac{1}{4}$, after each iteration of the subdivision we are left with around $\frac{3}{4}$ of the previous length. The depth of the tree is thus $\log_{3/4} n$ in the expected case. (There are deterministic selection methods that guarantee the choice of $\log n$ elements, and can therefore be used to implement a work-efficient algorithm.)

The step complexity of this algorithm is $S(n) = O(\log n)$, and the work complexity is $W(n) = O(n)$. However, the algorithm requires a considerable amount of book-keeping, which significantly increases the complexity of the code. In addition, since we are far from being able to subdivide in a binary fashion, the constant is even larger.

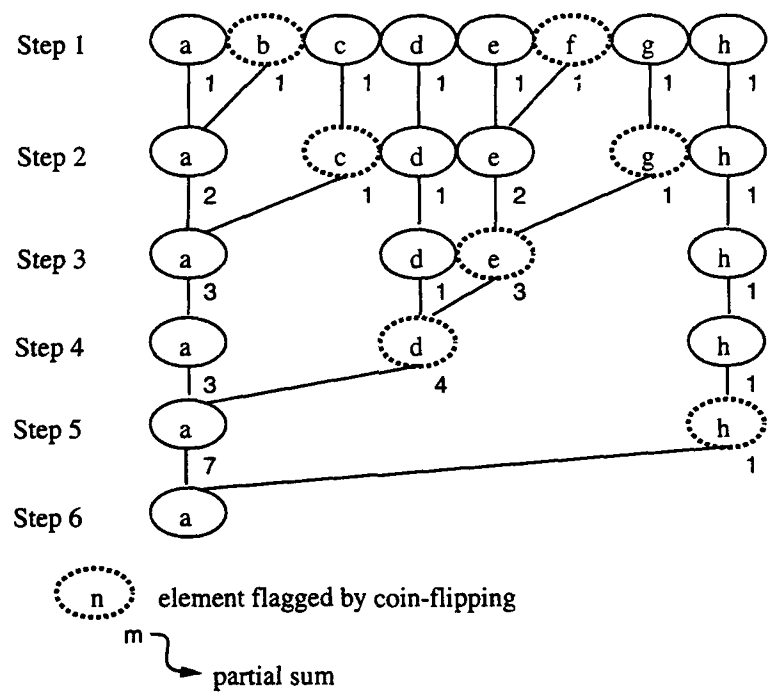


Figure 4: Subdivision tree created by coin flipping

4 For More Information

List ranking is covered in Chapter 30 of Cormen, Leiserson and Rivest [12]. Randomized algorithms are discussed in Chapter 9 of Jájá [15]. Miller and Reif [20] give more applications of the parallel tree contraction algorithm.

15-840: Programming Parallel Algorithms — Lecture #7
Scribe: John Pane — Thursday, 8 Oct 92

Overview

- The truth about complexity in NESL.
- Sorting, merging and order statistics:
  - Parallelizing serial algorithms
  - Merging in parallel
  - Batcher's bitonic sort

1 The Truth About Complexity In NESL

1.1 Work complexity

A more accurate formula for work complexity is

$$W(n) = \sum_{i=1}^{m} w_i + \sum_{c \in \text{constants}} (size(c) \times m)$$

where m is the number of parallel calls. When a constant is used inside of an apply-to-each form, NESL makes copies of the constant. A ›onstant is defined as any variable that appears in the apply-to-each form but is not bound in the apply-to-each (i.e., it is a constant with respect to the apply-to-each, not to the whole program). This is only a problem in primitives where the amount of work is less than the size of the largest vector.

For example, the work complexity for {a + #a: a} is $(length(a))^2$. This would be trivial to fix by assigning *#a* to a variable outside of the apply-to-each form.

1.2 Step complexity

Composition of step complexity only really works if the code inside the apply-to-each form is *contained*. The definition of contained code is code where only one branch of a conditional has $S(n) > O(1)$. For example, the following code is not contained because both branches of the inner `if` have $S(n) > O(1)$:

```
function power(a, n) =
  if (n == 0)
  then 1
  else
    if evenp(n)
    then square(power(a, n/2))
    else a * square(power(a, n/2))
```

This can be fixed by calculating `power(a, n/2)` outside the conditional:

```
function power(a, n) =
  if (n == 0)
  then 1
  else
    let pow = power(a, n/2)
    in if evenp(n)
       then square(pow)
       else a * square(pow)
```

Ideally, NESL would not have this restriction. It exists because of the way the compiler is currently implemented.

2 A Comparison Of Sorts

In the following table, all times are big-O. For quicksort, the times are expected case.

Sort	Serial	Parallel	
	Time	Step	Work
Insertion	n^2	n	n^2
Selection	n^2	n	n^2
Bubble	n^2	n	n^2
Merge	$n \log n$	-	-
Quicksort	$n \log n$	$\log n$	$n \log n$
Radix	n	-	-
Heap	$n \log n$	-	-

3 Odd-Even Transposition Sort

This is the parallel equivalent of bubble sort. The following steps are repeated $n/2$ times:

1. Compare each element at an even position with the element after it, swapping if necessary.
2. Do the same with elements at odd positions.

4 Trying To Parallelize Heapsort

Heapsort consists of first creating a heap (an implementation of a priority queue), and then removing the elements one by one from the top of the heap. Although it is easy to create the heap in parallel, we cannot remove elements from a standard serial heap in parallel. We consider both phases:

1. Heapify. If we view the heap as a tree, we can heapify the tree by recursively heapifying the left and right children and then inserting self. The insertion requires $O(\log n)$ time and the two recursive calls can be made in parallel. The work and step complexities are therefore:

$$\begin{aligned} W(n) &= 2W(n/2) + O(\log n) = O(n) \\ S(n) &= S(n/2) + O(\log n) = O(\log^2 n) \end{aligned}$$

2. Remove elements. Each removal consists of taking the minimum element (which is at the top of the heap), replacing it with the last element in the heap, and sifting the new top of the heap down. The sifting operation takes on average $\log n$ steps, and we cannot make multiple removals at the same time. The heap property only guarantees that the top element is the minimum—we don't know anything special about the other elements near the top of the heap.

The serial heapsort therefore does not lead to a good parallel sorting algorithm.

5 Mergesort

Mergesort consists of recursively sorting two halves of a sequence and then merging the results. To get a good parallel mergesort, we must first have an efficient parallel algorithm for merging. The standard serial merging algorithm repeatedly selects the lesser of the first elements of the two sorted sequences. This cannot be directly converted into a parallel algorithm. Instead, we now consider three different parallel algorithms for merging.

5.1 Merging by divide-and-conquer

We can use the following divide-and-conquer approach:

1. Select the middle element of one of the sequences tc form a *cut*. To ensure a reasonable division of labor, always cut the larger of the two sequences.
2. Use a binary search to find the corresponding cut in the other list.
3. Recursively merge the two pairs of cut sequences.
4. Append the two merged sequences to form the full list.

To determine the complexity of this algorithm, we need to consider what data structure we should use to represent the sequences:

- **An array:** using an array, the cut and binary search takes $O(\log n)$ work and steps, and the append takes $O(n)$ work and $O(1)$ steps. Thus the complexities are:

$$\begin{aligned} W(n) &= 2W(n/2) + O(\log n + n) = O(n \log n) \\ S(n) &= S(n/2) + O(\log n + 1) = O(\log^2 n) \end{aligned}$$

- **A linked list:** using a linked list, the cut and binary search takes $O(n)$ work and steps, but if we keep a pointer to the end of each list the append can be done in $O(1)$ work and steps. The complexities are therefore:

$$\begin{aligned} W(n) &= 2W(n/2) + O(n+1) = O(n \log n) \\ S(n) &= S(n/2) + O(n+1) = O(n) \end{aligned}$$

- **Array and linked list:** if we use an array on the way down the recursion, and a linked list on the way up, then we get the best of both worlds and get the following complexities:

$$\begin{aligned} W(n) &= 2W(n/2) + O(\log n + 1) = O(n) \\ S(n) &= S(n/2) + O(\log n + 1) = O(\log^2 n) \end{aligned}$$

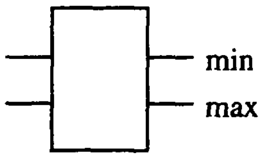
Thus we can get a work-efficient parallel merging algorithm if we use two different data structures. The only problem is that the result is returned as a linked list. To convert this list back into an array (so that it can be used as an input for another merge) requires a list-rank. This can be done with $O(\log n)$ steps and $O(n)$ work.

A mergesort based on this merging algorithm would therefore require:

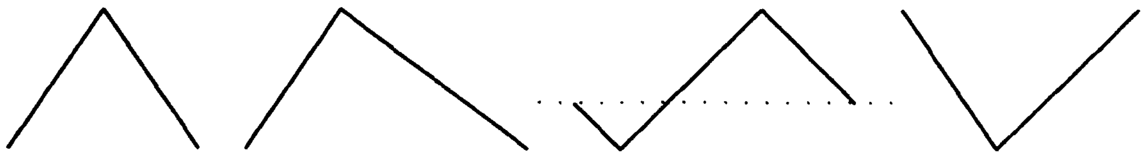
$$\begin{aligned} W(n) &= 2W(n/2) + O(n) = O(n \log n) \\ S(n) &= S(n/2) + O(\log^2 n) = O(\log^3 n) \end{aligned}$$

5.2 Batcher's bitonic merge

This algorithm was developed for a switching network composed of comparator switches. Comparator switches are 2×2 gates which place the minimum input value on one output and the maximum input value on the other output:



Bitonic merging consists of taking a *bitonic sequence* and converting it into a monotonic sequence. A bitonic sequence is composed of two back-to-back monotonic sequences, one increasing and one decreasing, rotated by any amount. In general, we don't know where the inflection point is in the bitonic sequence. Some examples:



To convert a bitonic sequence into a monotonic sequence, we can use the following algorithm:

1. Split the sequence down the middle (this is not necessarily the inflection point).
2. Compare the two halves element by element, sending the minimum values to the left and the maximum values to the right. The two resulting subsequences are bitonic, and all of the values in the left subsequence are less than all of the values in the right subsequence (see Figure 2).

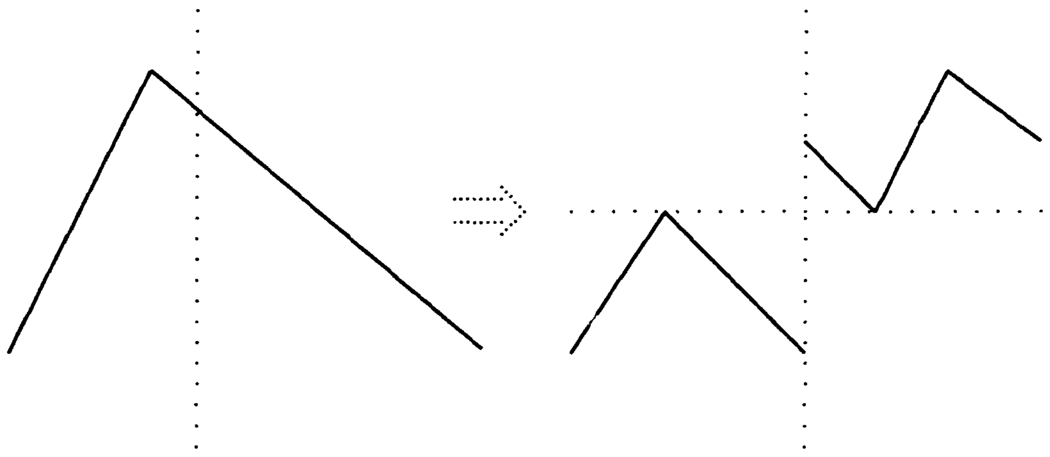


Figure 1: Splitting a bitonic sequence into two subsequences

3. Recursively sort the subsequences, and append the results.

To create a parallel mergesort out of this algorithm, we need to reverse one of the two input sequences and then append them to form a bitonic sequence. The step complexity of the resulting sort is $S(n) = O(\log n)$, and the work complexity is $W(n) = 2W(n/2) + O(n) = O(n \log n)$. This is not work efficient, but it has the advantage that it can be implemented directly in hardware as a bitonic merging network (see Figure 2).

The input wires receive the values in the bitonic sequence, and feed into a backwards butterfly. Note that each pair of switches shown can be replaced by a single switch, since they are both computing the same maximum and minimum. The depth of the network is $\log n$, and the communication pattern is independent of the data. A sorting circuit can be built out of these sorting elements as shown in Figure 3; the depth is $\log n(\log n + 1)/2 = (log^2 n)/2$, and the number of switches required is $depth \times n/2$.

6 For More Information

Batcher [4] gives more details of the bitonic merge and its application in sorting. For a general discussion of sorting networks, see Chapter 28 of Cormen, Leiserson and Rivest [12].

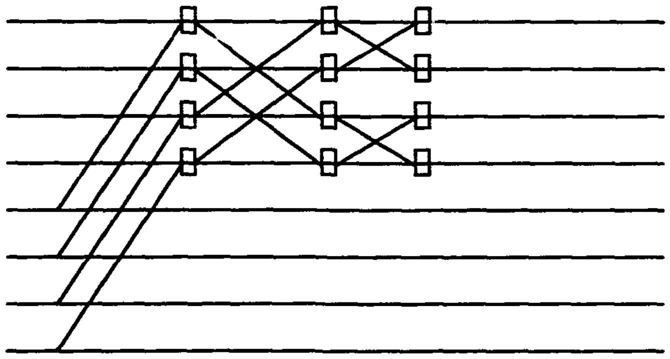


Figure 2: Structure of a bitonic merging network

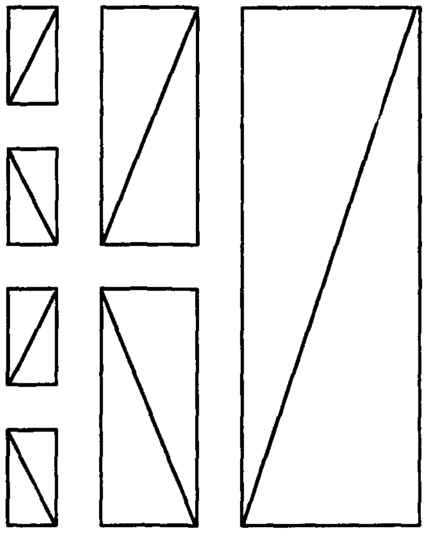


Figure 3: Using bitonic merge to build a sorting network

Overview

- More about merge—a simpler alternative to the merge introduced last lecture.
- Sample sort—multiple pivots with local or recursive sorting of buckets.
- Radix sort—a parallel version of counting sort.

1 Halving Merge

The halving merge algorithm is yet another example where solving a smaller problem provides a framework for a parallel solution of the larger problem. The first steps of the halving merge for lists L_1 and L_2 are:

1. Choose even-indexed elements from L_1 and L_2.
2. Recursively merge them.

So how does this help us to merge L_1 and L_2? Consider for example:

$$\begin{aligned} L_1 &= [3,5,7,9] \\ L_2 &= [1,2,8,11] \end{aligned}$$

Merging the even-indexed elements results in

$$[1,3,7,8].$$

Since L_1 is sorted, an even-indexed element from L_1 is less than or equal to the next (odd-indexed) element from L_1 and similarly for L_2. Hence it seems reasonable to insert these odd-indexed elements after the even elements in the merged list. Insertion of the odd-indexed elements into the merged result above gives

$$[1,2,3,5,7,9,8,11].$$

Clearly this particular result is not sorted, and it's not sorted for two lists in general. However, it is *near-sorted* (i.e., sorted except for non-overlapping cycles).[1] If an efficient way existed to undo the cycles in the result, the merge would be complete. In fact, there is a way to transform a near-sorted sequence into a sorted one with $O(n)$ work and $O(1)$ steps. In NESL:

```
function fix_near_merge(nm) =
  let x = reverse(min_scan(reverse(nm)));
      y = {max(n, nm): n in max_scan(nm); nm}
  in {min(x, y): x; y}
```

[1] Proving the near-sorted property of this list is left as an exercise for the reader.

A mergesort constructed from the halvir,; merge plus this final fixup step has

$$\begin{aligned} S(n) &= S(n/2) + \log n = O(\log^2 n) \\ W(n) &= 2W(n/2) + n = O(n \log n), \end{aligned}$$

or on a P-RAM with p processors

$$T_{PRAM} = O\left(\frac{n \log n}{p} + (\log^2 n) \log p\right).$$

2 Sample Sort

Sample sort is an example of a parallel algorithm that has no useful serial analog; it is a parallel generalization of quicksort. The steps involved are:

1. Pick a set of $p - 1$ splitters, which will be used to partition keys.
2. Based on the splitters, send the keys to one of p buckets.
3. Sort within the buckets.[2]
4. Append buckets to form the final result.

2.1 Selecting splitters

One way to select splitters is to simply pick $p - 1$ keys at random and sort them. This results in an average bucket size of n/p. However, this naive approach has an expected worst case bucket size of $(n \log n)/p$. So, it is likely that one bucket is bigger than the average bucket by a factor of $\log n$.

One way to select splitters to make the buckets more uniform in size is to use *oversampling*. The idea is that instead of sampling $p - 1$ keys, we pick $s(p - 1)$ keys where s is the oversampling ratio. A reasonable choice for s is $\log n$. Then after sorting the $(\log n)(p - 1)$ keys, pick $p - 1$ keys with a stride of $\log n$. See Blelloch et al. [8] for a comparison of the theoretical and empirical ratio of maximum bucket size to average bucket size over a range of oversampling ratios. A tradeoff exists between obtaining uniform bucket sizes and the time takes by this phase of the sort.

2.2 Distributing keys to buckets

After the splitters have been selected, they are broadcast to each processor. Each processor then performs a binary search to partition its keys over the buckets, and routes its keys to the appropriate buckets. This part of the algorithm takes

$$\begin{aligned} S_{buckets} &= S_{broadcast} + S_{binsearch} + S_{send} \\ &= O(p + (n/p) \log p + (n/p)). \end{aligned}$$

[2]This can be done serially, although we could also do it with a recursive call to sample sort in order to decrease broadcast overhead.

2.3 Sorting the buckets

The time taken by this phase is equal to the sort time of the processor with the largest bucket. If we assume the largest bucket is only a constant factor larger than the average bucket (which is almost certainly true if we pick a good oversampling ratio), then

$$S_{localsort} = O((n/p)\log(n/p)).$$

2.4 Summary

Thus total step complexity is

$$S_{sampsort} = O(p + (n/p)\log p + (n/p) + (n/p)\log(n/p)).$$

We pick p to minimize $S_{sampsort}$, which essentially minimizes the second term, and so

$$\begin{aligned} p &= (n/p)\log(p) \\ p^2 &= n\log(p) \\ p &= \sqrt{n\log(n)}. \end{aligned}$$

This algorithm is practical because the initial send of the keys to buckets is the major communication overhead. The rest of the algorithm proceeds locally to each processor.

3 Radix Sort

3.1 Serially speaking

Recall that serial radix sort requires being able to extract subdigits of numbers, and sorts each subdigit starting at the least significant digit. For instance, for radix-10 (i.e., decimal) the following list of numbers

$$\begin{array}{ccc} 3 & 9 & \boxed{5} \\ 5 & 8 & \boxed{3} \\ 9 & 6 & \boxed{2} \\ 5 & 8 & \boxed{2} \end{array}$$

is sorted by comparing the least significant digits and exchanging the keys accordingly (as shown). Recall that this intermediate step must use a stable sort (i.e., equal keys remain in the same order as in the unsorted list). Hence on the next iteration we get:

$$\begin{array}{ccc} 9 & \boxed{6} & 2 \\ 5 & \boxed{8} & 2 \\ 5 & \boxed{8} & 3 \\ 3 & \boxed{9} & 5 \end{array}$$

The keys now happen to be in order according to the middle digit, and thus the last step produces:

3	9	5
5	8	2
5	8	3
9	6	2

A common way to implement each step of the serial radix sort is to use the rank of each digit directly in a counting sort. If there are r possible values for a digit, each counting sort requires $O(n+r)$ work and hence, for keys in the range $0 \ldots m$, would require $O((\log m/\log r)(n+r))$ total work.

3.2 Parallel ways

If there was a stable way to find the ranks of the keys in parallel, this rank vector could then be used to permute the keys. In order to do this, we would like to have a multi-prefix (Ranade [28]) or fetch-and-op (Ultracomputer [13]) operation which would allow the system to count the number of keys in buckets $k < j$, and use that result as the start of a "sub-scan" through the keys in the bucket j in order to compute their ranks.

What if we don't have this type of operation? One solution is to partition the keys into sets of size $O(\sqrt{n})$. Then assign a bucket radix to each processor which results in the number of buckets being approximately equal to the number of keys. Each processor can then do a radix sort independently. A scan step is necessary afterwards to rank the keys of each bucket. A permutation can then be performed to form a new ordering of the original list.

The above steps of partition, sort, recombine and send would need to be done about twice as many times as in the case where we had the multi-prefix operation.

4 For More Information

See Blelloch et al. [8] for a comparison of sorting algorithms on the CM-2, including radix, sample, and bitonic sorts. Deterministic sorting is covered in Chapter 4 of Jájá [15].

15-840: Programming Parallel Algorithms — Lecture #9
Scribe: Chris Okasaki — Thursday, 15 Oct 92

Overview

- Review of radix sort.
- String comparison.
- Collecting duplicate elements.

1 Review of Radix Sort

Remember that radix sort is *not* comparison based. It may, however, easily be applied to integers, floats, strings, or tuples of these types. To sort floats that use the sign-exponent-mantissa IEEE standard requires some preprocessing which flips the exponent and mantissa based on the sign bit. Once the preprocessing is done, a standard integer sort will sort the numbers (some care has to be taken with invalid numbers, NaN's). For strings, it is relatively simple to arrange that work is proportional to the combined length of the strings. For a set of n strings each of length l_i, we can use radix sort to sort with complexity $W = O(\sum_{i=0}^{i<n} l_i)$ and $S = O(\max_{i=0}^{i<n} l_i)$.

The bottom line is that radix sort is not as limited as many people think. The knock against radix sort is that its *theory* may not be as interesting as other algorithms'. However, if you look at the library sort routines for machines such as the Connection Machine CM-2 or the Cray, you'll see that they all use radix sort.

In most sorting algorithms, we don't worry about the size of the objects being sorted. For integers, we usually assume that they are at least large enough to represent pointers ($\log n$ bits for a problem size of n); in fact, we usually assume that the size is some multiple of this value, i.e., $k \log n$. But for radix sort it is vitally important to know the size of the integers—it will determine how many passes of the algorithm are required. We want to balance the number of keys and the number of buckets, where the number of buckets is determined by the number of bits considered on each pass. More buckets means fewer passes of the algorithm, but more overhead in maintaining those buckets. For a serial algorithm, a good choice is to use n buckets ($\log n$ bits per pass and k passes). Each pass then requires $O(n)$ work and the total work is $O(kn) = O(n)$. For a typical problem that we might be solving on a serial supercomputer, we might have a million 64-bit keys. Since $\log n = 20$, this would require 4 passes.

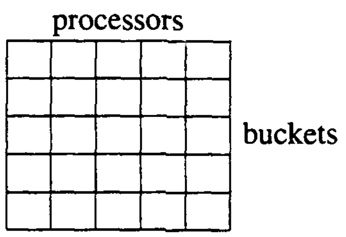
In a parallel radix sort we need to limit the number of buckets, so we may choose a smaller "chunk size", say $\log n/2$. This requires twice as many passes and double the work, but it's still $O(n)$. In general, we want to choose a chunk size such that the number of processors times the number of buckets per processor is $O(n)$. So more processors means a smaller chunk size (fewer buckets per processor) and more passes. For instance, to sort 2^{18} 64-bit keys using 2^{12} processors, we could use 2^6 buckets per processor and $\lceil 64/6 \rceil = 11$ passes (compared to the 4 used by the serial processor).

1.1 Counting sort

Each pass of the parallel radix sort is a single counting sort, which consists of 4 steps:

1. Clear buckets.
2. Count into buckets (easy with a *send-with-add* or similar, harder otherwise).
3. Scan buckets (a normal `plus_scan`).
4. Calculate new locations (the hardest part, requires a *multi-prefix* or similar).

Since this is difficult to parallelize, we break the keys into groups (1 group per processor) and count serially on each processor. To combine the results we perform a scan across the matrix formed by the buckets on each processor. If this matrix is represented as



the scan operation should add the values in first row (the 0 buckets), then "wrap around" to the second row (the 1 buckets), and so on. This can be accomplished in NESL by performing a vector scan on the transposed and flattened matrix.

2 Rank *vs.* Sort

Often when we might consider sorting, what we really want is the *rank* of each element (the index it would have in the sorted list) rather than the sorted list itself. NESL provides a rank operation as a primitive. For example:

```
rank([3, 4, 7, 2, 1, 5]) ⟹ [2, 3, 5, 1, 0, 4]
```

Given `rank`, it is trivial to implement `sort` as

```
function sort(a) = permute(a, rank(a))
```

3 Comparing Strings

A comparison-based sorting algorithm for strings requires (of course) a comparison routine for strings. This is easy to implement in serial, but how might we write a parallel version?

It's actually just a `reduce` operation, but over a somewhat more complicated operation than + or `max`. Consider first how to compare strings of length two:

$$ab < cd \equiv (a < c) \vee ((a = c) \wedge (b < d))$$

The same relationship holds true if we consider a, b, c, d to be substrings rather than characters.

So we need to maintain two pieces of information (= and <) on each substring. This is easy to establish (in parallel) for the individual characters of each string (up to the length of the shorter string). Then, we can repeatedly combine this information on larger and larger substrings by reducing over the following function:

```
function lex(a, b) =
  let (a=, a<) = a
  in if a=
     then b
     else a
```

This would be used as `lex_reduce({(a == b, a < b): a; b})`. Note that `lex` is associative (this is a requirement for `reduce` to work properly) but, unlike + and `max`, it is *not* commutative.

4 Collecting Duplicates

Suppose we wanted to collect together all duplicate elements in a vector, e.g.

`[3, 7, 4, 4, 3, 3, 7, 4, 3]` $\Longrightarrow$ `[[3, 3, 3, 3], [4, 4, 4], [7, 7]]`

In serial, we might use hashing and linked lists, but linked lists are bad for parallel implementations (although efficient list ranking algorithms exist, they only work if the position of every element in the list is known at the start, rather than only the position of the head).

Alternatively, we could just sort the items and then group the neighboring duplicates (using the algorithm for `partition` for example). But there is a useful refinement to this approach. By first applying the naming problem, so that each unique key is in the range $0 \ldots n-1$, radix sort may be used very efficiently—just two passes on $\log n/2$ bits each.

Often, we may be interested in grouping data associated with each key rather than the duplicate keys themselves. NESL provides a function `collect` which does just this. For example:

```
collect([(2, "this"), (3, "a"), (2, "is"), (5, "test")])
  ⟹ [(3, ["a"]), (2, ["this", "is"]), (5, ["test"])]
```

Overview

- Medians and order statistics.
- String operations:
  - Searching
  - Word breaking
  - Word wrapping
  - Parsing

1 Medians And Order Statistics

Given a set A of size n and an index i, suppose that we wish to select the i^{th} smallest element from A. Of course, this can be done simply by sorting the set A and yanking out the appropriate element—however, sorting requires $n \log n$ work, and thus this approach is not work efficient. In the serial world there are two well-known algorithms used to select the i^{th} smallest element. The first is a randomized algorithm, and has an expected complexity of $O(n)$. The second is a deterministic algorithm with worst-case complexity of $O(n)$. However, the second algorithm is more complex to implement, and has a high constant; in practice, the first algorithm is used most of the time.[1] We find that the first algorithm is also the easier of the two to parallelize.

1.1 Selection in expected linear time

A summary of the algorithm:

1. Randomly pick a pivot from set A.
2. Pack the elements of the set A into 2 vectors: elements less than the pivot into the first, those greater than or equal to the pivot into the second.
3. Based on the size of each vector, decide which vector contains the i^{th} smallest element of the original vector. Recurse on the appropriate vector, subtracting from i the correct amount so that the recursive call selects the i^{th} smallest element relative to the original vector.

Although this algorithm is similar to quicksort, it does less work, since a recursive call is made on only one of the two partitions. Of course, in the worst case, the vector passed to each recursive call is only one element smaller than the original vector, and the recursion will reach depth n. However in the expected case, the algorithm has depth $\log n$ and does $O(n)$ work.

This expected linear time algorithm is easy to parallelize: all of the comparisons and packs can be done in parallel.

[1]Both algorithms are described in Chapter 10 of Cormen, Leiserson and Rivest [12]

Selection in worst-case linear time

A summary of the algorithm:

1. Divide the set A into groups of five elements.
2. Sort each group of 5 elements. Any sort algorithm can be used to to do this, since the net effect is to provide a constant time operation over each group of 5.
3. Find the median of each group of 5.
4. Recursively find the median of all the medians found in step 3, and use this value as the pivot.
5. Continue as with the previous algorithm.

This algorithm has worst-case linear time because it is *guaranteed* that a certain fraction of the elements will be less than the pivot value, and thus at least that many elements will be eliminated on each recursive call. Consider the following diagram, showing the set A after step 4:

$$\begin{array}{ccccccccccccc}
\longleftarrow & & & & & \multicolumn{3}{c}{n/5 \text{ columns}} & & & & & \longrightarrow \\
\mathtt{L} & \mathtt{L} & \mathtt{L} & \mathtt{L} & \ldots & \mathtt{L} & \mathtt{L} & \mathtt{X} & \ldots & \mathtt{X} & \mathtt{X} & \mathtt{X} & \mathtt{X} \\
\mathtt{L} & \mathtt{L} & \mathtt{L} & \mathtt{L} & \ldots & \mathtt{L} & \mathtt{L} & \mathtt{X} & \ldots & \mathtt{X} & \mathtt{X} & \mathtt{X} & \mathtt{X} \\
\mathtt{M} & \mathtt{M} & \mathtt{M} & \mathtt{M} & \ldots & \mathtt{M} & \mathtt{P} & \mathtt{X} & \ldots & \mathtt{X} & \mathtt{X} & \mathtt{X} & \mathtt{X} \\
\mathtt{X} & \mathtt{X} & \mathtt{X} & \mathtt{X} & \ldots & \mathtt{X} & \mathtt{X} & \mathtt{X} & \ldots & \mathtt{X} & \mathtt{X} & \mathtt{X} & \mathtt{X} \\
\mathtt{X} & \mathtt{X} & \mathtt{X} & \mathtt{X} & \ldots & \mathtt{X} & \mathtt{X} & \mathtt{X} & \ldots & \mathtt{X} & \mathtt{X} & \mathtt{X} & \mathtt{X}
\end{array}$$

Each column is a sorted group of 5 elements, increasing from top to bottom. The third row contains the median of each column, and the columns are arranged in order of increasing medians from left to right. The element `P` is the median of the medians, and is used as the pivot. All of the medians marked `M` are less than `P`, and all elements marked `L` are less than their corresponding medians, and therefore also less than `P`. Thus, it is guaranteed that in $n/10$ columns, 3 of the elements will be less than the pivot, and therefore at least $3n/10$ elements will be eliminated in each recursion.

This algorithm first performs a constant-time sort on $n/5$ groups of elements. It then calls itself recursively on a set of $n/5$ medians, and finally performs another recursive call on a set that is no larger than $7n/10$. Its work complexity is thus

$$W(n) = kn + W\left(\frac{n}{5}\right) + W\left(\frac{7n}{10}\right) = O(n).$$

Note that although the work complexity of algorithm is $O(n)$ its constant is large due to the expensive subsorts. Its step complexity is

$$S(n) = 1 + S\left(\frac{n}{5}\right) + S\left(\frac{7n}{10}\right) = O(n^{\sim 0.82}).$$

Unfortunately, this step complexity of about $O(n^{0.82})$ is large compared to the $O(\log n)$ complexity of the first algorithm—thus, this algorithm will not show as significant a speedup when parallelized, since much of its work cannot be done in parallel. Vishkin [35] improves on the algorithm described above by breaking into groups of $\log n$ elements, giving a step complexity of $S(n) = O(\log n)$ and a work complexity of $W(n) = O(n)$.

Observe that randomization appears to play a fairly important role in parallel algorithms, as reflected by the two algorithms presented: the randomized algorithm parallelized well but the deterministic one didn't.

2 String Processing

There are many types of string processing problems that we might want to solve in parallel. Some examples are:

- Searching (e.g., `grep`). See next lecture.
- Breaking into words.
- Word counting (e.g., `wc`).
- Text justification (e.g., `fmt`).
- Parsing (e.g., of regular expressions, or context-free languages).

2.1 Breaking text into words

Given a set of delimiters (space, newline, etc) we wish to break up a string into its component words. A NESL implementation would take as input a string and return a vector of strings:

`"xxxxx yyy zzzzzzz"` $\Longrightarrow$ `("xxxxx" "yyy" "zzzzzzz")`

This is easy to implement with constant step complexity using NESL. Basically, each character position looks one character ahead of itself, setting an end-of-word flag if the following character is a delimiter. The input is then split based on these flags. Recall that as this takes constant time in NESL, it can be implemented in logarithmic time on a PRAM.

2.2 Word counting

This problem is similarly easy to parallelize, and can be implemented as an extension to the word breaking problem.

2.3 Word wrapping

The goal of word-wrapping is to lay some text out line by line, producing a line break before any word that would otherwise extend beyond a certain character position (i.e., line wrapping as `emacs` might do it). The input is a line length and a vector of integers, corresponding to the lengths of the words, and the output is a vector of integers representing the positions at which line breaks are inserted. It is assumed that no word is longer than the line length. In the serial world, this problem can easily be solved with $O(n)$ work.

Word wrapping is a difficult problem to solve in parallel, because there is a strong dependency in needing to know the $(i-1)^{th}$ line in order to generate the i^{th} line. This dependency of future lines on previous lines makes it difficult or impossible to use a scan-like operation to solve the problem in parallel. No solution was discussed in class.

3 Parsing

3.1 Example: parsing a context free language

This example outlines an approach one might take to do parallel parsing of expressions in a simple parenthesized LISP-like language. An expression to be parsed looks like:

```
(defun (foo a) (+ a 7))
```

The first step in parsing such an expression involves breaking the expression up into tokens (i.e., *tokenizing*). In a LISP-like language, this can be accomplished by doing a one-character lookahead at each character position. End-of-token flags are set at character positions preceding a space, at spaces, and at parentheses.

There are a couple of extensions to the basic algorithm:

- Integers are harder to recognize because the end of an integer is more dependent on context. However, they can still be handled with a limited lookahead at the appropriate character positions.
- Comments are of the form "`;` . . . `\n`". We can handle these by creating a vector of flags in which `1`'s are placed at locations containing a semicolon. Comments can then be flagged by doing a segmented `or_scan` over this vector, with the scan being reset at character positions containing a `\n`. Of course, this approach would also capture comments that are enclosed within text strings, so all strings must be flagged before to applying this procedure.

Once the expression has been tokenized, its parentheses need to be matched. *Parenthesis matching* cannot be done using a scan alone. This is basically due to the same reason that parenthesis matching cannot be done using a regular expression—a scan operation, like a regular expression, keeps a constant amount of state, and parentheses cannot be matched using only a constant amount of state. A context-free language, or something with a "stack", is required.

However, a scan can serve as the first step in parenthesis matching. A `plus_scan` over a vector containing a 1 for every "`(`" and a -1 for every "`)`" will number every opening and closing parenthesis with the appropriate nesting depth number. The parentheses are balanced if an equal number of ('s and) 's occur at each nest depth.

If all of the parentheses at a given depth are grouped together, then a string of the form "() () () () () () () . . ." results, i.e., matching becomes a trivial matter of looking at the neighboring parenthesis. Once the parentheses have been checked for balance, they can be grouped together simply by sorting them according to their depth numbers. A radix sort is particularly appropriate for this task as the maximum depth number is known in advance, and is less than the size of the input string. Unfortunately, however, a radix sort won't complete in $\log n$ steps. If this becomes a problem, a completely different approach is required, such as divide and conquer (the input is divided and parentheses are matched in each partition; unmatched parentheses must be dealt with in the merge process). The algorithm is rather tricky to code, but it has the virtue of being work efficient if implemented properly.

3.2 Parsing regular languages

The standard way of parsing a regular language is to convert the regular expression describing it into a finite state automaton (FSA), and to simulate the execution of this FSA when given the string to be parsed as input. Thus, the task of parsing a regular language in parallel is the same as that of simulating the execution of a FSA in parallel.

Recall that a FSA consists of an input tape, a head that reads the input tape, a finite set of states, and a transition table describing, for each state, what states may be reached from it if an appropriate symbol is read from the input tape. One way to represent the transition table is to construct a set of $m \times m$ boolean matrices T_{ij}, such that there is one boolean matrix for each letter of the input alphabet (m is the number of states in the automaton). For each input symbol σ, the boolean matrix T^{σ} is defined as:

$$T^{\sigma}_{ij} = \begin{cases} 1 & \text{if state } j \text{ may result if } \sigma \text{ is read in state } i \\ 0 & \text{otherwise.} \end{cases}$$

Observe that if the automaton being described happens to be a deterministic FSA, each of the matrices will simply be a permutation of the states, and could more easily be represented by a list of pairs.

The behavior of the FSA for a certain input string can be computed in parallel by doing a `matrix_multiply_reduce` over the string of boolean matrices obtained by replacing each character in the input with its corresponding boolean transition matrix.[2] The result of this reduce operation is a matrix specifying what state the FSA will terminate in, given the input string in question and a particular initial state. The terminating state corresponding to the appropriate initial state can then be looked up in the table. Observe that the problem can be solved in this manner only because matrix multiplication is *associative*, thereby allowing the reduction to be computed in a tree-like fashion.

Unfortunately, each of the boolean matrices has m^2 entries, where m is the number of states in the FSA. This means that n matrix multiplications must be performed, each with m^3 cost, yielding a work complexity of $O(nm^3)$. If the FSA in question is deterministic and permutations are used

[2]If all the intermediate states of the FSA need to be known, a `matrix_multiply_scan` can be used instead.

to represent the transition tables, the work complexity becomes $O(nm)$—recall that permutations can be composed in linear time, and that composition of permutations is associative. However, the analogous serial algorithm does $O(n)$ work, and thus neither of the parallel algorithms is work efficient, although in cases where m is small they remain practical. Some work has been done to show that $O(nm)$ is the best work complexity with which this problem can be solved in parallel. Note, however, that the parallel algorithm provides more information than is obtained from the serial algorithm—it provides the resultant state for *all* start states. If a serial algorithm were to compute this rather than provide the result for just one start state, it would have to do the same amount of work as the parallel algorithm.

In conclusion, some parsing problems can be solved efficiently in parallel (such as the LISP-like parsing in the earlier example); solutions to other parsing problems may not be completely work efficient but are nevertheless often practical.

4 For More Information

Selection is discussed further in Jájá [15, pages 181–184].

15-840: Programming Parallel Algorithms — Lecture #11
Scribe: Jürgen Dingel — Thursday, 22 Oct 92

Overview

The entire lecture was devoted to the problem of string matching. Formally:

> *Input*: A string S of length n where each element is from some finite set of characters, and a pattern P of length $m < n$.
> *Output*: Start locations of all occurrences of P in S.

The following concepts were introduced:

- Periodic/non-periodic patterns
- Witness arrays
- Duel functions

1 The Naive Serial Algorithm

Serially, the problem can easily be solved by going through the string while looping over the pattern. More precisely, for each element of the string S which is equal to the first element of the pattern P we compare the following elements of S with those of P. If the whole pattern can be matched the start index is part of the output. If not, we move to the next element of the string. For illustration, consider the following pseudocode:

```
r ← 0
for i = 0 to length (s) − 1 do
  k ← 1 ; l ← 0
  while s[k] = p[l] and l < length (p) do
    k ← k + 1 ; l ← l + 1
  if l = length (p) then
    result [r] ← i ; r ← r + 1
```

The worst case for this algorithm is when the string contains a lot of repeated characters, as for example in S = aaaaaaaaa and P = aaaa. In this case the complexity is $O(nm)$. But when the characters are distributed less "randomly" over the string (we will make this more precise later) then we have $O(n + m)$. So this algorithm is suitable for e.g., ordinary text, but not for binary numbers.

Several attempts have been made to come up with a more efficient algorithm for this fundamental problem. One is the Knuth-Morris-Pratt algorithm from 1977 which has a worst case complexity of $O(n + m)$. Another is the Boyer-Moore algorithm with the same worst case complexity but with $O(n/m)$ complexity for some inputs (when the characters are distributed irregularly).

2 The Parallel Algorithm

The aim is to design a work-efficient parallel algorithm (i.e., one with $O(n + m)$ work complexity), with a logarithmic step complexity.

2.1 A naive solution

We could simulate the serial algorithm as follows: first, concurrently get all the indices of elements of the string which are equal to the first element of the pattern (candidate array). Then we recursively compare the rest of the pattern with the substrings starting at these indices, and knock out the ones where a mismatch occurs. The work complexity of this algorithm is the same as for the serial version, i.e., $O(nm)$ for worst case and $O(n+m)$ for irregular strings. The step complexity is $O(m)$. Thus, the algorithm works efficiently for irregular strings and short patterns. But we want more...

2.2 A better solution

The algorithm described in the following can also be found in Jájá [15, pages 327–337].

The key to a more efficient (albeit more complicated) solution lies in the observation that it is mainly the regularities in the pattern which make the algorithm inefficient. We thus must find a better way to cope with these regularities. It seems to make sense to split the algorithm into two phases:

1. Pre-analysis of the pattern.
2. Searching for the pattern in the string.

where we want the pre-analysis phase to detect the regularities such that in the search phase they can be dealt with more easily.

First, we formalize what we mean by regularities. A pattern P is *periodic* if it is of the form U^kV where $k > 1$ and V is a prefix of U. U is called the period of P. For example, abababa is periodic whereas abababc is not.

Now we introduce the notion of a *witness array*. Rather than giving a formal definition we just provide the intuition. Suppose we had a non-periodic pattern P of length m. Imagine a copy of P is placed on top of P such that the first and the i^{th} ($1 \leq i \leq \lfloor \frac{m}{2} \rfloor$) positions are aligned. Then, the overlapping portions of the strings cannot be identical, because if they were P would contain a period of length i. For example,

$$\begin{array}{cccccccccccccc}
 & & & & & a & b & c & a & \underline{a} & b & c & a & b \\
 & & & & a & \underline{b} & c & a & a & b & c & a & b & \\
 & & & \underline{a} & b & c & a & a & b & c & a & b & & \\
 & & \underline{a} & b & c & a & a & b & c & a & b & & & \\
nP= & a & \underline{b} & \underline{c} & a & \underline{a} & b & c & a & \underline{b} & & & & \\
 & 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & & & &
\end{array}$$

The witness array $W[0 \ldots \lfloor \frac{m}{2} \rfloor]$ contains for each copy aligning at position i the index of an element which distinguishes the overlapping portions (there could be several of these elements). These indices are meant to point into the original pattern. So, for the above example the corresponding witness array is $W = [0, 1, 2, 4, 8]$. Note that the first position of the witness array, i.e., $W[0]$, is filled with a dummy value.

Also note that:

1. We only consider $\lfloor \frac{m}{2} \rfloor$ copies because by definition we want the period to occur at least twice.
2. If we split the string into portions of length m this means that the pattern can occur at most once in the second half of a portion.
3. The computation of the witness array can be used to determine whether a pattern is periodic or not; the pattern is periodic iff for one of the copies no difference can be found.
4. The witness array can be computed by simply taking $\lfloor \frac{m}{2} \rfloor$ copies of the pattern and concurrently comparing them appropriately with the pattern. This requires $O(m^2)$ work and a constant number of steps.

Using the concept of a witness array, dealing with both non-periodic and periodic patterns becomes much easier. Point 2 above suggests how a non-periodic pattern can be taken care of.

2.3 Non-periodic patterns

To handle non-periodic patterns, the string S is decomposed into blocks s_i consisting of $\lfloor \frac{m}{2} \rfloor$ consecutive characters. Each block is dealt with concurrently. We know that each s_i can contain at most one position at which a match can occur. Out of the $\lfloor \frac{m}{2} \rfloor$ candidates in each block we have to filter out this position. Suppose we had a way to eliminate all but one candidate in each s_i. We could then concurrently compare the rest of the pattern with the positions following the candidate. Since we only have $\frac{2n}{m}$ blocks this would require $O(n)$ work.

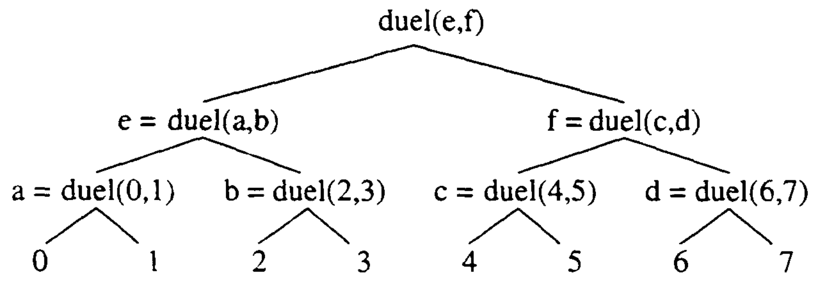
As a way to disqualify all candidates in s_i but one we now introduce the *duel function*. This function will be given the string s_i, the witness array W of the pattern P and two locations k and l of s_i. It will return as output one of the locations k and l such that P can not occur at the eliminated location. Since the witness array tells us where two copies at positions k and l differ we can look into the string at the corresponding position and see which one of the copies does not match. More precisely,

```
duel(k, l) =
    differ_at ← W[l − k]
    **if** s_i[ differ_at ] ≠ p[ differ_at ] **then return** l **else return** k
```

For illustration consider the example below. p_k and p_l denote the k^{th} and the l^{th} copies of the pattern.

$$\begin{array}{lcccccccccc}
p_l = & & & & a & \underline{b} & a & a & b & & \\
p_k = & & a & b & a & \underline{a} & b & & & & \text{duel}(k,l) = l \\
s_i = & \dots & a & b & a & \underline{b} & a & \dots & & &
\end{array}$$

Copy p_k is knocked out by the duel function because the pattern cannot start at position k. For each of the blocks s_i we can now disqualify all but one candidate by building a binary tree using the duel function. Suppose s_i had 8 elements:



Since only the root is a reasonable candidate we only keep this.

Because each of the s_i can be treated concurrently and the duel function is constant, the overall algorithm has step complexity $O(\log m)$. For each of the s_i the work is $O(m)$, and since there are $2n/m$ of these blocks, the work for the overall algorithm is $O(n)$.

Instead of building the tree we could also walk linearly through the $\lfloor \frac{m}{2} \rfloor$ candidates of a block s_i doing the same kind of "duel-reduce", which would give a step complexity of $O(m)$.

2.4 Periodic patterns

We now describe how to handle periodic patterns. Suppose the given pattern P is of the form U^kV where U is minimal, i.e., U is a prefix of every other period of p. Let m be the length of u. Given the witness array of P this case can be dealt with quite easily using the above algorithm. The idea is that we first find all the occurrences of U in S, and then we check if these can be extended to occurrences of P.

1. First, we split the string S into blocks s_i with m elements each.

2. Since U is not periodic we can now use the above algorithm for the non-periodic case to get the one candidate for an occurrence of U in each of the s_i. This gives us an array of flags C in which exactly the candidate positions are set. For example:

$$\begin{array}{rcllllllllllllllll} P & = & (\mathrm{a\,b})^2 \\ S & = & \mathrm{c} & \mathrm{a} & \mathrm{b} & \mathrm{a} & \mathrm{a} & \mathrm{b} & \mathrm{b} & \mathrm{a} & \mathrm{b} & \mathrm{a} & \mathrm{b} & \mathrm{a} & \mathrm{b} & \mathrm{a} & \mathrm{b} & \mathrm{c} \\ C & = & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 0 \end{array}$$

   As shown above this would require $O(\log n)$ steps and $O(n)$ work.

3. For each of the occurrences of P in S we then have to check if it is followed by at least k candidates (which must all be m elements apart). This can be done by turning the array of candidates into a $\frac{n}{m} \times m$ matrix and doing a segmented backward `plus_scan` on each row. For example:

$$C = \begin{array}{cccccccc} 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 1 & 1 & 1 & 0 \end{array} \qquad C' = \begin{array}{cccccccc} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 3 & 2 & 1 & 0 & 0 \end{array}$$

   The elements in C' with an entry greater than or equal to $k = 2$ correspond to occurrences of P in S. This also requires $O(\log m)$ steps and $O(n)$ time.

Thus, the overall complexity of this algorithm is $S = O(\log m)$ and $W = O(n)$.

3 For More Information

The string matchirg algorithm presented here is based on Breslauer and Galil [10]. String matching is covered at length in Chapter 7 of Jájá [15].

15-840: Programming Parallel Algorithms — Lecture #12
Scribe: Jonathan Hardwick — Tuesday, 29 Oct 92

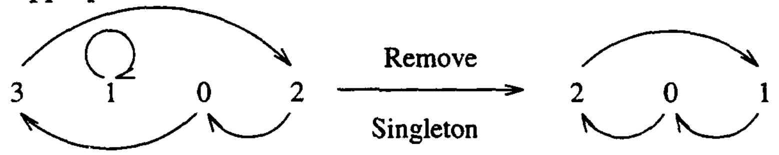
Overview

- Discussion of Problem 2 from Assignment 3.
- Review of string matching.
- Unbalanced trees.

1 Assignment 3, Problem 2

The task was to write a function `CountCycles` that took a permutation and returned the number of cycles in the permutation. The basic idea was to use a random mate algorithm; the tricky part of the problem was how to keep track of the permutations. There were two common approaches:

- Write a specialized pack function that "correctly" packs a permutation, renumbering the elements as appropriate:

  Suitable NESL code is:

```
function pack_permutation(array, flags) =
  {a in enumerate(flags) -> array; fl in flags | fl}
```

- Keep two auxiliary vectors around which point to the next and previous elements in a permutation for each element. These two vectors are side-effected by the algorithm, but are never actually compressed. The vector that does get compressed uses pointers into these vectors to access them.

2 String Matching

Given an irregular pattern of length m and a string of length n, we want to pick out ~ sample of the pattern that represents a "fingerprint". The idea is that the fingerprint captures the nonregularity of the pattern in a much shorter form, and thus matching the fingerprint against the string is faster than using the full pattern. Formally, a fingerprint is a set of up to $\log m$ positions within the pattern, with the property that any two matches to a fingerprint will be separated by at least $m/2$ positions.

The fingerprint is constructed from *witness locations*. Given two copies of a pattern which are out-of-sync by up to $m/2$ positions, a witness location is a position at which the two copies differ. Since the pattern is not periodic, there must be at least one such witness location for any pair of copies. See the diagram in Section 2.2 of the previous lecture for an example.

A suitable algorithm to find a fingerprint is

```
Arrange $m/2$ copies of the pattern on top of each other, shifting each by 1.
repeat
  Find the witness location of the leftmost and rightmost copies of the pattern.
  One of the two copies must match at most half of the other copies at the witness location.
  Choose the copy with the fewest matches, and discard everything else.
  Add the witness location to our fingerprint.
  Repeat with the new smaller set of copies.
end
```

Since this algorithm starts with $m/2$ copies of the pattern, and throws away at least half of the copies on each step, it has a worst case step complexity of $S(m) = O(\log m)$.

For example, the first iteration of this algorithm applied to the example from the previous lecture would identify a witness location at position 8 in the pattern:

$$\begin{array}{ccccccccccccc}
 & & & & a & b & c & a & \underline{a} & b & c & a & b \\
 & & & a & b & c & a & a & b & c & a & b & \\
 & & a & b & c & a & a & b & c & a & b & & \\
 & a & b & c & a & a & b & c & a & b & & & \\
a & b & c & a & a & b & c & a & \underline{b} & & & & \\
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & & & &
\end{array}$$

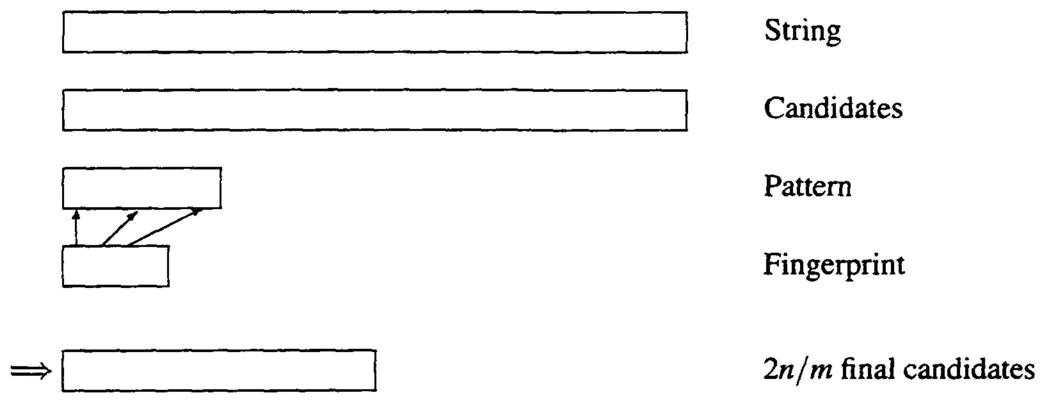
Throwing out all the copies which don't have an "a" at that location (we could just as well have chosen "b"), the second iteration would identify a second witness location at position 5:

$$\begin{array}{ccccccccccccc}
 & & & & a & \underline{b} & c & a & a & b & c & a & b \\
 & a & b & c & a & \underline{a} & b & c & a & b & & & \\
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & & & &
\end{array}$$

Thus, one possible fingerprint is `_ _ _ _ a _ _ _ b`. Note that this is a fingerprint for the original pattern, but we could also end up with a fingerprint for one of the shifted copies. For example, `_ _ _ _ b _ _ _ b` is an equally valid fingerprint for the pattern shifted over by 3.

Given the fingerprint, we then do an exhaustive match against every position of the string, taking $O(n \log m)$ steps. Due to the property of the fingerprint, we are guaranteed to find at most $2n/m$ matches. These matches are then checked against the full pattern, to eliminate any spurious matches; this takes $O(n)$ steps. To summarize, the full algorithm has 3 steps:

1. Extract a fingerprint from the pattern.
2. Use the fingerprint to match against the string.
3. Check all matches found against the full pattern.



The overall step complexity of the algorithm is $O(2n + n \log m)$.

3 Unbalanced Trees

Due to their regular nature, balanced trees are usually easy to deal with in parallel algorithms. However, there are many practical applications for which we would like to use unbalanced trees, rather than possibly paying the fixed-depth cost of a balanced tree:

- Expression evaluation
- Compiler technology
- Graph algorithms
- Text algorithms, e.g., word-wrapping

As an example, consider the "summing subtrees" problem, where we have an unbalanced binary tree whose nodes are labeled with integers, and we want to replace each label with the sum of all labels of its children in the tree.

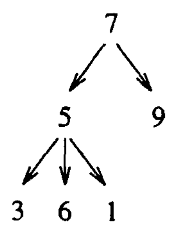


Figure 1: Example of tree summation

This can be though of as a *leaffix* operation, by analogy to the tree-based prefix operations on sequences. How can we do this work-efficiently, i.e., with step complexity $S(n) = O(\log n)$, where n is the number of nodes in the tree? It is easy to see how to do this for the two limiting cases (see Figure 2); level-order traversal for the balanced binary tree, and list ranking for the linear list.

In order to do tree summation on anything in between these two extremes, we need to introduce a new operation.

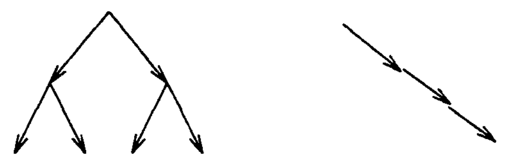


Figure 2: Limiting cases of tree summation

3.1 Euler tour

This is a recursive depth-first tour of the tree from a leaf node, associating two edges in the tour with each edge in the tree (see Figure 3).

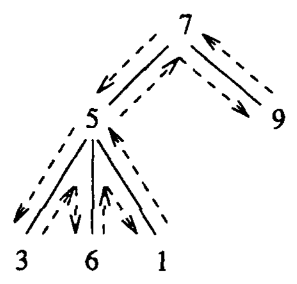


Figure 3: Euler tour of a tree

It is easy to generate a linked list that represents the tree using a Euler tour, and the list can then be converted into an ordered vector using list ranking. For example, we can view the above tree as a parenthesized list where the label was inserted on the way down the tree, resulting in the following vector:

```
( ( ( ) ( ) ( ) ) ( ) )
7 5 3 0 6 0 1 0 0 9 0 0
```

We can then do a `plus_scan` operation on this vector, and finally for each matching pair of parentheses subtract the value at the right parenthesis from that at the left parenthesis to give the sum of the subtree rooted at that node. For example, in NESL:

```
datatype ptree([int], [int])

function left_paren(a) = let ptree(l, r) = a in l

function right_paren(a) = let ptree(l, r) = a in r

function plus_leaffix(v, tr) =
  let l = dist(0, 2 * #v) <- {(v, i): v; i in left_paren(tr)};
      s = plus_scan(l)
  in {right - left: right in s -> right_paren(tr);
                    left in s -> left_paren(tr)}
```

Note that this only works because we have an inverse for the + operator—it wouldn't work for an associative operator which didn't have an inverse, e.g., max.

3.2 Rootfix operation

This is the inverse of the leaffix operation; it operates on everything "above" it (i.e., towards the root) in the tree. The `plus_rootfix` can be constructed from the Euler tour using the following technique; instead of inserting the label n at the position of a "(" and a 0 at the corresponding ")", insert n at the position of the "(" and $-n$ at the matching ")". This ensures that all the subtrees cancel out, and only values coming down from "above" have any effect. In NESL:

```
function plus_rootfix(v, tr) =
  let l = dist(0, 2 * #v) <- {(v, i): v; i in left_paren(tr)};
      r = l <- {(-v, i): v; i in right_paren(tr)};
      s = plus_scan(r)
  in s -> left_paren(tr)
```

3.3 Trees in parenthetical representation

An example application of a rootfix operation is to calculate the depth of every vertex in a tree. Using the tree in Figure 3 as an example, construct two vectors which point to the locations in the Euler tour with matching parentheses:

$$\begin{aligned} lp &= [0,1,2,4,6,9] \\ rp &= [11,8,3,6,7,10] \end{aligned}$$

Put 1's into a vector of twice the size using the indices in lp:

$$l = [1,1,1,0,1,0,1,0,0,1,0,0]$$

Put -1's into the vector using the indices in rp:

$$l = [1,1,1,-1,1,-1,1,-1,-1,1,-1,1]$$

`plus_scan` l, and get from the result of the scan using the indices in lp:

$$\begin{aligned} l &= [0,1,2,3,2,3,2,3,2,1,2,1] \\ &\Longrightarrow [0,1,2,2,2,1] \end{aligned}$$

4 For More Information

See Blelloch [6] for more on the leaffix and rootfix operations. The Euler tour technique is due to Tarjan and Vishkin [32], and the string matching algorithm is due to Vishkin [36].

15-840: Programming Parallel Algorithms — Lecture #13
Scribe: Dave Eckhardt — Thursday, 29 Oct 92

Overview

- Notes on associativity.
- More tree contraction:
  - Rake-compress
  - Left-rake/right-rake
- Evaluating arithmetic expressions.
- Evaluating variable-binding trees.

1 Associativity

Various members of the class have pointed out that finite-precision arithmetic operations are not always associative, even when their mathematical ideals are. The fact that they are not associative is a problem because parallel scans yield correct results only for associative operators.

For example, consider integer addition. If a machine dealt with additive overflow by returning a ceiling or floor value, that would not be associative. For example, let the "top" value be 1000; then

$$-900 + 900 + 900 = 100 \text{ } or \text{ } 900.$$

depending on the order in which we do the additions. Fortunately, most machines handle additive overflow by wrapping, and this *is* associative. Assuming words range from -1000 to 1000, then

$$900 + 900 = 1800 \rightarrow -200$$

$$-900 + -200 = -1100 \rightarrow 900.$$

Fixed-precision floating-point addition is even more interesting, since overflow isn't the only problem—it is easy to accumulate roundoff errors. Single-precision floating point numbers typically have a mantissa of only 23 bits. Since we must normalize numbers (express them as multiples of the same power of 10) before adding them, we cannot accurately add numbers which differ by more than a factor of about $10^7 \doteq 2^{23}$. For example, $10^7 + 1 \rightarrow 10^7$.

Now, imagine a sum of the vector $[10^7, 1, 1, 1, 1, 1, 1, 1, \ldots]$, containing a total of $10^7 + 1$ elements. The mathematically correct answer is 2×10^7; a serial loop would yield 10^7, and a parallel reduce implemented by a tree would yield around $2 \times 10^7 - 3$ (the 10^7 would "eat" one or two 1's, but by the second or third level of the tree the numbers would be large enough to normalize without loss). So in this case the parallel implementation would produce a more accurate result than the serial one. It's important to point out that the most accurate algorithm depends on the values and order of the input vector. As a practical matter, addition is most stable if the vector is first sorted by increasing *absolute* value.

2 Tree Contraction

In this section we examine parallel algorithms for contracting trees. If trees have n nodes and depth d, the challenge is to devise algorithms that run in $O(\log n)$ steps rather than $O(d)$, because trees are often unbalanced. The *Rake-Compress* algorithm is randomized; the *Left-rake/Right-rake* algorithm is deterministic but requires both an Euler tour and a list-ranking as preprocessing steps.

2.1 Rake-compress

The rake-compress algorithm reduces a binary tree to a single point by alternately applying two operators:

- **Rake**. Contract all leaves into parent nodes.
- **Compress**. Shorten long chains of one-child nodes by employing a standard coin-flipping algorithm. All nodes flip coins; if a node comes up "heads", and has a single child that comes up "tails", it contracts that child into itse'f.

The algorithm consists of alternating the two operators until the tree has contracted down to one node. Why does this work? For an intuitive explanation, consider the extreme cases the algorithm deals with: balanced trees (each rake step will decrease the tree height by one) and linked lists (each compression step will probabilistically remove a constant fraction of the nodes).

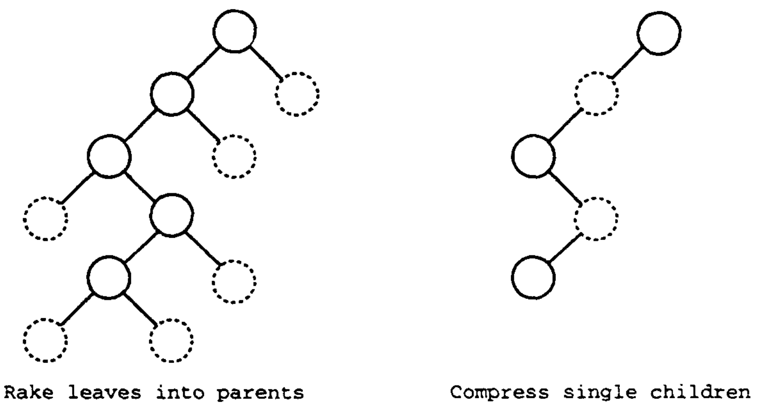


Figure 1: Rake-compress

The actual proof is complicated, but note that any tree which resists one operator will fall prey to the other: if there are few leaves relative to the height of the tree, there will be long chains; if there are few long chains, there must be many leaves.

2.2 Left-rake/right-rake

This algorithm works only on strictly binary trees (those where every internal node has two children). It requires that all leaves be labeled by an in-order numbering (which can be generated by doing an Euler tour from the root and then a list-ranking of all nodes which are leaves), with the exception of the leftmost and rightmost nodes. There is one basic operation, rake, which removes both a leaf and its parent, compressing both into the leaf's grandparent.. This operation is equivalent to the two operations of rake-compress: rake on a leaf and then compress on that leaf's sibling. Thus, one-child chains don't have a chance to form. The algorithm has three steps:

1. Rake odd left leaves.
2. Rake odd right leaves.
3. Divide all leaf numbers by 2.

After the first two steps, half of the leaves (the odd ones) have been removed, and roughly half of the new leaves are odd.

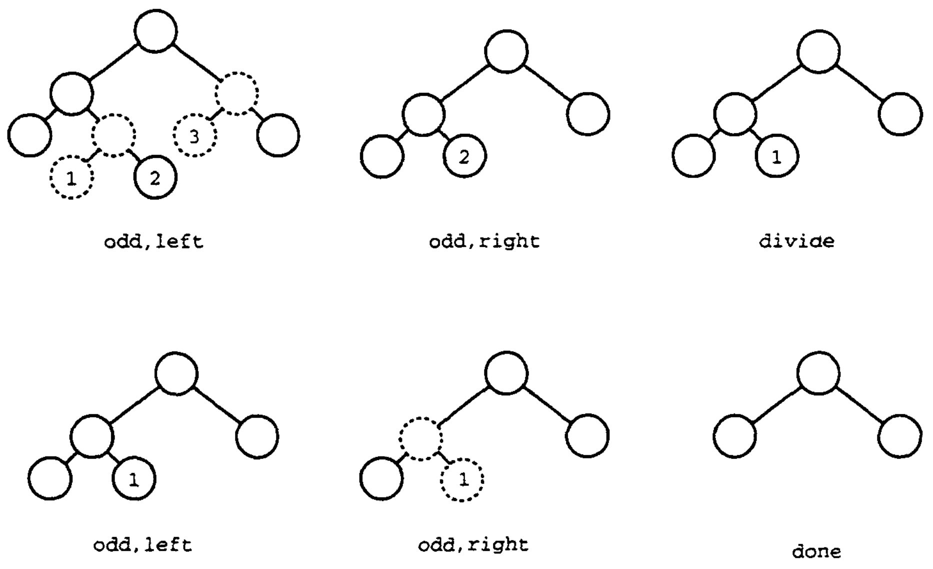


Figure 2: Left-rake/right-rake

3 Arithmetic Expression Evaluation

Given a tree with nodes containing either arithmetic operators (such as +,−,×,/) or values, we want to reduce the tree to one node containing the answer. Once again, we would like to do this in time proportional to the number of nodes, regardless of the number of leaves or tree height. In order to do this, we need to be able to contract any node, not just ones labeled with values. We do this by labeling nodes with "continuations" or "futures"—descriptions of how the results of child subtrees will influence the node's value. If we limit ourselves to multiplication and addition, each node's continuation consists of two numbers, a multiplier and an offset. When each node is evaluated, its result is then multiplied and added to before being passed up the tree. Nodes begin with continuations of $(1,0)$.

The continuations allow us to compress nodes which have at least one evaluated child. For example, imagine we are compressing a +-node which already has a continuation of (5, 6) (see Figure 3). This node has one evaluated child, with value 4 and continuation (1, 0), and one unevaluated child (some subtree with arbitrary structure) with an unknown value x and a continuation of (2, 1).

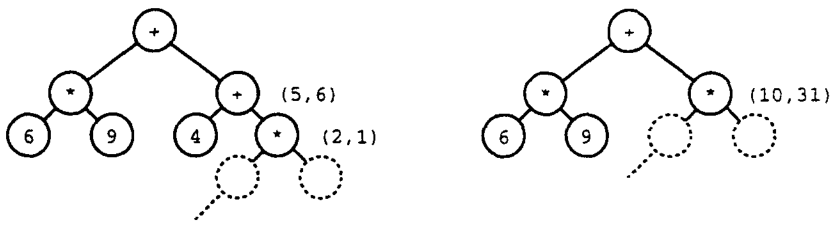


Figure 3: Arithmetic evaluation

The subtree will contribute $2x + 1$ and the evaluated child will contribute 4. Since the +-node has a continuation which specifies multiplication by 5 and then an addition of 6, the value the +-node will contribute to the eventual result is

$$5((2x + 1) + 4) + 6$$

or

$$10x + 31$$

which means that we can remove the + and 4 nodes and change the continuation on the unevaluated subtree from (2, 1) to (10, 31). This generalizes to a rule of the form: given a +-node (m_1, a_1), children c and (m_2, a_2), delete the parent and the constant node and update the other node to be $(m_1 \times m_2, m_1 \times (a_2 + c) + a_1)$.

There is a similar rule for collapsing a $\times$-node. Since division is not commutative, trees involving division have more complex continuations.

4 Variable Binding Trees

Imagine a compiler confronted with

```
(with ((x 1))
      (with ((x (+ 4 x)))
            x))
```

The problem is to discover, for each reference to x, which is the correct definition. Serial solutions to this problem involve stacks which are pushed on definitions, consulted on references, and popped on closing parentheses which terminate `with` clauses. Obviously this doesn't parallelize well.

Let's look at the single-variable case first. If we could collapse an expression tree so that it contained only nodes which used or referred to x, then each reference to x would be the immediate child of the defining node. Then propagating the information back to the full tree would be trivial.

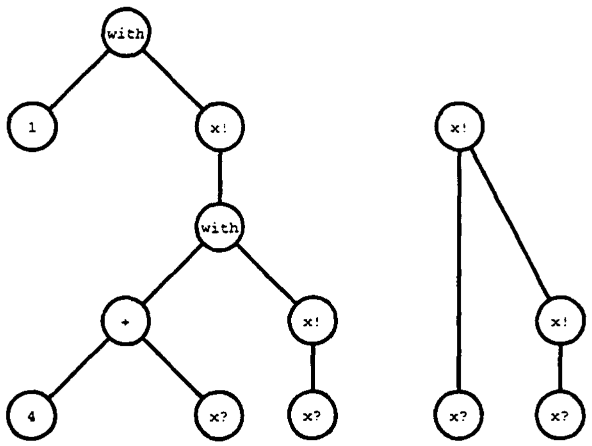


Figure 4: Contracting "with" trees

Luckily, we have work-efficient parallel algorithms which, given a parenthetical representation of a tree, can delete any number of vertices.

For the multi-variable case, first map all variables to the integers $1 \ldots n$ (the naming problem). Keep not only the variable definition and "use" nodes, but also the binding ("with") nodes. In addition to a boolean flag indicating whether a node is kept or deleted, keep a vector of these variable indices. After packing out the deleted nodes, use collect (splitting via the variable indices) to generate a vector of trees, one for each variable. This process is stable, so that each reference will end up being a sibling of the correct definition.

5 For More Information

Tree contraction is discussed in Chapter 4 of Jájá [15]. The rake-compress algorithm is due to Miller and Reif [20], and the left-rake/right-rake algorithm is due to Kosaraju and Delcher [17]

15-840: Programming Parallel Algorithms Lecture #14
Scribe: Xuemei Wang Tuesday 11/3/92

Overview

- Practical applications of graph algorithms.
- Some interesting graph problems.
- Study in detail: connected components.

1 Graphs

Graphs are frequently used in practical applications. The following are some examples:

- Operations on sparse matrices. Matrices are really one way of representing graphs. Rows and columns in a matrix represent nodes in a graph, and the entries in the matrix represent edges. Therefore, many matrix operations become algorithms on graphs.
- Optimization, e.g., of maxflow problems.
- Compilers.
- AI applications, especially in vision.

2 Some Interesting Problems On Graphs

- Connected components—label the nodes in the same connected component with the same label.
- Minimum spanning tree.
- Biconnected components—any two nodes in a biconnected component are connected by at least two paths.
- Shortest path, topological sort.
- Maximum flow.
- Maximal independent set.
- Matching problem.

Lots of research is currently being done on parallel graph algorithms. Work and step complexity bounds improve almost monthly.

3 Connected Components

Let m be the number of edges and n be the number of nodes in a graph. If we represent graphs with adjacency lists, depth-first or breadth-first search is $O(m)$, since each edge gets visited exactly once. But these searches are difficult to parallelize, because depth-first search is inherently serial and we cannot parallelize breadth-first search beyond the diameter of the graph (consider the pathological case of a linked list). To better solve this problem, we will use *graph contraction*. We will talk about the random mate algorithm for graph contraction today, and in the next lecture we will discuss deterministic graph contraction algorithms.

3.1 Random mate graph contraction

Given a graph G as shown in Figure 1, randomly select 50% of the nodes as parents, and 50% as children (see Figure 2). Every child that neighbors at least one parent picks a parent to contract into. All edges from the child are carried over to the parent. See Figure 3 for an example. Notice that one parent may have several children, but a child can have only one parent. In the average case, after each contraction at least 1/4 of the vertices will be removed (1/2 of the nodes are expected to be children, and at least 1/2 of the children will have parents). The graph after contraction is shown in Figure 4. We continue this process until eventually only one node is left. If we need to label each component, we have to expand the graph after all the contractions are finished.

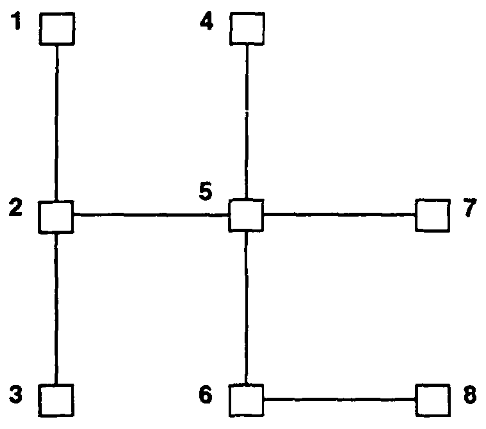


Figure 1: Input graph G

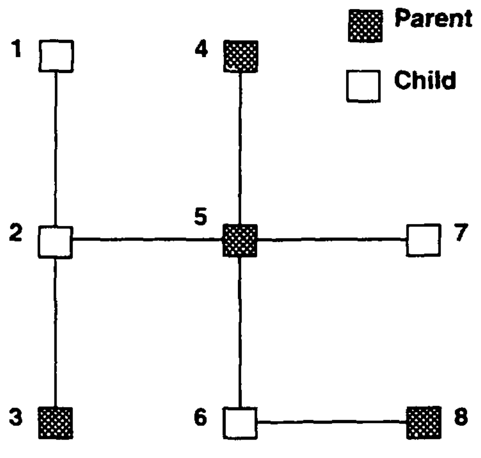


Figure 2: Selecting parents and children by coin flipping

The implementation of this algorithm depends on the representation of the graph. The following three schemes can be used to represent a graph (no' that we don't want to use linked lists for representing graphs because they make parallelizing very difficult).

1. Sequence/vector/list of edges. Each edge is represented as a pair of connecting nodes. Thus, graph G in Figure 1 can be represented as [(1, 2), (2, 3), (2, 5), (4, 5), (5, 6), (5, 7), (6, 8)].

2. Adjacency list. The i^{th} entry in the vector represents all the nodes that are connected to the i^{th} node. So graph G can be represented as [[2], [1, 3, 5], [2], [5], [2, 4, 6, 7], [5, 8], [5], [6]].

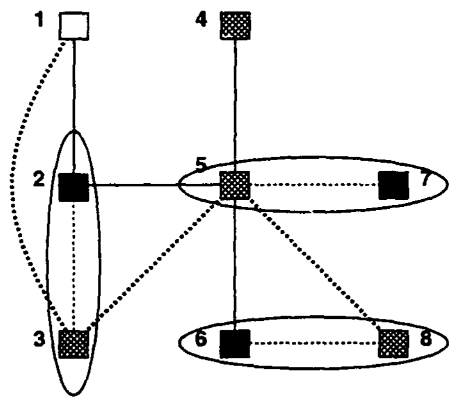


Figure 3: Contracting the children

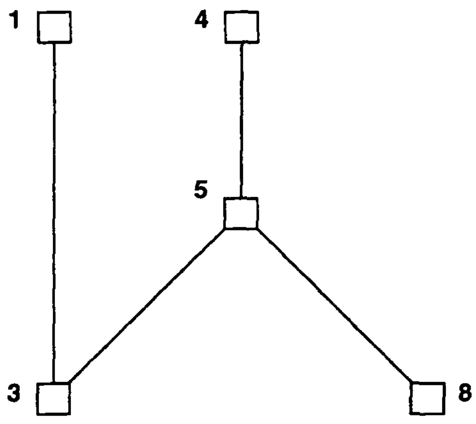


Figure 4: After contraction

3. Adjacency matrices.

In NESL, to do the sum-all-neighbors operation in scheme 1 is difficult without a fetch-and-add instruction, but in scheme 2 it is very easy since we already have locality of node information. We can convert a vector of edges to an adjacency list by sorting the vector of edges on the first node.

Scheme 1, i.e., a vector of edges, is perfectly good for random mate graph contraction. The contraction is done as follows:

1. Create an array of length n (where n is the number of nodes in the graph), initialized to an index array so that every node points to itself. For graph G, this array is $[1,2,3,4,5,6,7,8]$.
2. Choose parents and children by coin flipping. For example, as shown in Figure 2, nodes 1, 2, 6, 7 are chosen to be children, and nodes 3, 4, 5, 8 are parents.
3. For each edge, if one node is a parent and the other node is a child, the edge writes the index of its parent into its child. For example, for edge (2, 3), since 2 is a child and 3 is a parent, the edge would write the value 3 to position 2 in the array. Notice that multiple parents may be writing to the same child at the same time, but only one (chosen arbitrarily) gets written. For example, in Figure 3, both of the edges (2, 3) and (2, 5) are writing the indices of their parents into position 2, but only 3 gets written in this case.
4. Update the edges. This is done by having each edge fetch from the array the new pointers for the nodes. Self edges, e.g., (3, 3), will be removed, but there still may be redundant edges.

which can be expressed in NESL as:

```
function graph_comp_rmate(D,E,seed) =
if (#E == 0) then D
else
```

```
let
  % Select hooking edges based on random mate %
  H = {(e1,e2) in E | hash(e1,seed) and not(hash(e2,seed))};
  % Hook using hooking edges %
  D = D <- H;
  % Update edges and throw out self edges %
  E = {(e1,e2) : e1 in D->{e1: (e1,e2) in E} ;
                 e2 in D->{e2: (e1,e2) in E}
               | e1 /= e2 };
  % Call routine recursively %
  R = graph_comp_rmate(D,E,nextseed(seed))
in % Expand tree back out %
  R <- {(e1,en): en in R->{e2: (e1,e2) in H} ; (e1,e2) in H}
```

The complexities of this algorithm are:

$$\begin{aligned} S(n) &= O(\log n) && \text{(with high probability)} \\ W(n) &= O(n + m\log n) && \text{(with high probability for the worst case)} \\ W(n) &= O(m) && \text{(in practice for planar graphs)} \end{aligned}$$

Note that this algorithm is not work efficient. To understand this, we can look at the following worst case scenario. Imagine a random graph with a large number of nodes, where each node has three edges. At each contraction, we almost double the number of edges per node until the graph becomes very dense. This way, the work we have to do is approximately $W(n) = O(n + m\log n)$.

4 For More Information

For more on random mate algorithms, see Reif [29].

15-840: Programming Parallel Algorithms — Lecture #15
Scribe: Bob Wheeler — Thursday, 6 Nov 92

Overview

- Connected components (continued).
- Minimum spanning trees.
- Breadth first search.
- Biconnected components.

1 Deterministic Graph Contraction

1.1 Building a spanning tree

In the previous lecture we discussed a random mate graph contraction algorithm for finding connected components. In this lecture we will discuss a deterministic approach to the same problem. Consider the following graph:

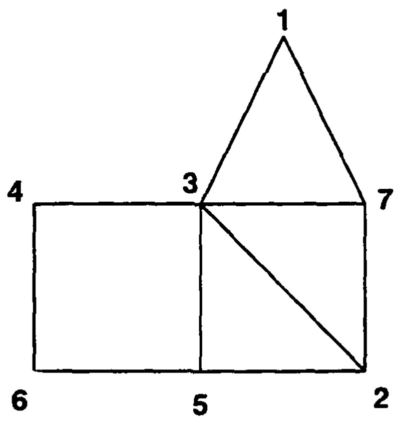


Figure 1: Sample graph

What we really want to do is contract all the nodes of the graph into a single node. If we had a spanning tree that was imposed on top of the graph, we could contract the graph by contracting the tree. The requirement for this tree is that it spans the whole graph and does not include any cycles. Unfortunately this turns out to be as hard as finding the connected components of the graph. Instead we'll settle for finding a number of trees that cover the graph, contract them and then deal with the remaining nodes in the same manner.

The basic idea is to hook the nodes into one or more trees, contract the trees and then repeat the process until the entire graph is contracted into a set of unconnected nodes. These will be the connected components.

One way to get a tree that doesn't have a cycle is to require that the node of a tree have a smaller index than any of the children below it. Each edge in our graph connects two nodes, each of which has a distinct index. Thus, one way to get such a tree from the graph is to have each edge write the smaller of the two indices that it connects into the node with the larger index.

A single node may be connected to multiple other nodes, all of which might attempt to write a value into the node. One of these writes will be the last and will thus succeed in connecting the two nodes into a tree. After this first step we might find ourselves with a graph that looks like this:

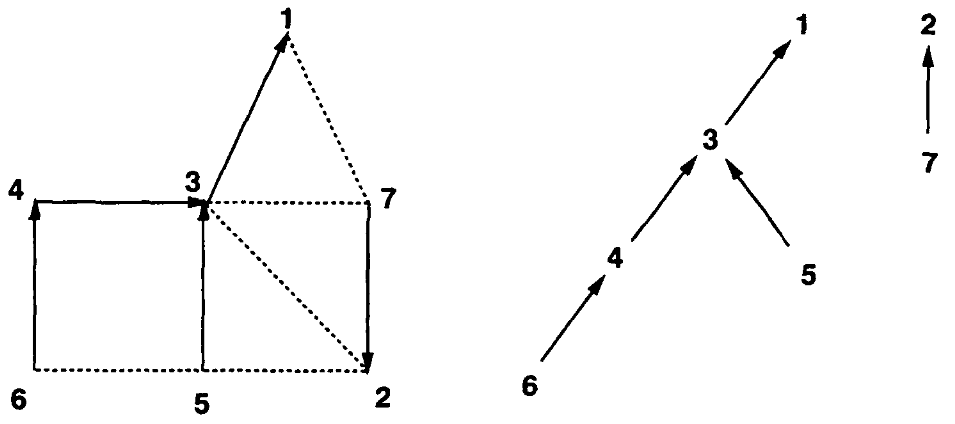


Figure 2: Graph and tree after first hooking operation

If every node were connected by a tree then we could remove half of the nodes with a single contraction of the tree. The problem is that we can't guarantee that all nodes will be connected—consider this counter example:

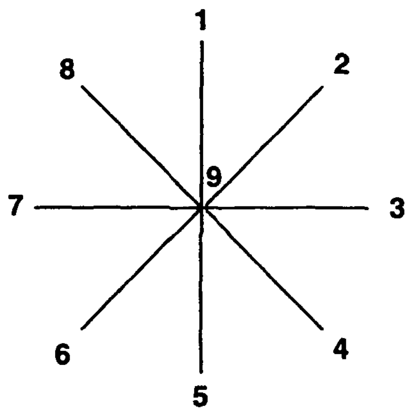


Figure 3: Counter example

In the hooking phase, all edges will write the value of the outer nodes into the center node. Only one write will succeed and the resulting graph will connect the inner node to only one of the other nodes. Thus each hooking step removes just a single node from the graph.

One solution to this problem is to alternate hooking lesser nodes to greater nodes and then greater nodes to lesser nodes. This guarantees that every node will be hooked on one of these steps, and that we will end up removing at least half the nodes on each pair of steps (only the root of each tree needs to remain).

1.2 Graph contraction and pointer jumping

In the above algorithm we were required to contract the trees that we built. One way to do this is by *pointer jumping*. First make the root point to itself so that it is the only node that is its own parent.

Pointer jumping consists of taking each node in the graph and moving its pointer to instead point to its parent's parent. (Point yourself at your grandparent.) We can show that tree contraction is guaranteed to terminate in $O(\log d)$ steps where d is the depth of the tree. The way we determine whether we've contracted the entire tree is to check after two iterations of pointer jumping if everyone still points to the same place. If so then we're done.

Note that in practice the trees encountered are not very deep so pointer jumping works fine. In theory we should use a work-efficient version of tree contraction (discussed in lecture 13).

After we've contracted the tree we are still left with the nodes of the original graph that were not connected to the same tree. In the above example we would have node 1 and 2 left. We would then run the algorithm again on these remaining nodes.

If we use a work-efficient algorithm, contraction of the tree has work complexity $W(V, E) = O(|E|)$ and step complexity $S(V, E) = O(\log |V|)$. After each step of the algorithm (hooking and tree contraction) we are guaranteed to have eliminated at least half the nodes in the tree. Thus the work complexity of the entire algorithm is $W(V, E) = (|E| \log |V|)$ and the step complexity is $S(V, E) = O(\log^2 |V|)$.

In practice the time is much less than this upper bound. Only in pathological cases does the time really take $O(\log^2 |V|)$, and in these rare cases a renumbering of the nodes will usually reduce the time.

1.3 A provable $O(\log |V|)$ algorithm

The above algorithm runs with worst case $O(\log^2 |V|)$ steps. We can do even better and get an algorithm that always runs in $O(\log |V|)$ steps and does no more work. To do this we need to merge smaller trees together during the tree contraction process. The big problem is making sure that we don't introduce cycles into the tree.

The invariant that we used above to ensure that we didn't introduce a cycle in the tree was to require that all parent nodes are smaller than the children. If we make an additional rule that when we graft the root of one tree into another that the root must connect to a non-leaf node then we can relax the invariant for leaf nodes. It turns out that this is a useful thing to do. Thus, our invariant will be that for any non-leaf node we will require[illegible]parent, and we will only graft into parent nodes.

With this invariant we can create an algorithm where we create the trees as before but instead of completely contracting each tree, we do a single pointer jumping step and then try to connect trees together.

One way to do the tree connection phase is as follows: each node looks at its parent and if its parent is a root of a tree it attempts to connect its parent into another tree if possible. Take the example in Figure 2. The parents of both node 2 and node 7 are roots, so we can try to connect the smaller tree into the larger. First we try using the edge $(3, 7)$. This won't work because hooking 2 into node 3 violates the principle that values in the tree always get larger as we move down.

Similarly using edge (2,3) doesn't work either. The answer is to use something called *conditional hooking*, which works like this:

```
for each edge $(i,j)$ do
  if parent $(i)$ is a root then
    if parent $(i) >$ parent $(j)$ then
      hook parent $(i)$ onto parent $(j)$
```

In our example assume that we are looking at edge (7,3); *parent* (7) is a root, and since *parent*(7) > *parent*(3) we can hook node 2 onto node 1. Of course if this hadn't worked we would have tried the edge (3,7).

In order to prove that this algorithm is $O(\log|V|)$ we need to hook every node on every pass. We still have the problem of the pathological star graph in Figure 3. We got around that before by alternating the direction of the hook. That worked because we always contracted the trees entirely before the next hook operation. If we tried it now, we might violate the invariant for numbering, which in turn might introduce a cycle in the tree.

The solution to this is called an *unconditional hook*. Here's the idea. Define a tree to be a star if it has a height that is at most 1. Stars will always look like Figure 3. An unconditional hook goes like this:

```
for each edge $(i,j)$ do
  if $i$ is a member of a star then
    if parent$(i) \neq$ parent$(j)$ then
      hook parent$(i)$ onto parent$(j)$
```

To see how this works, imagine that we're using our existing graph using minus edges (3,2) and (3,7) (see Figure 4).

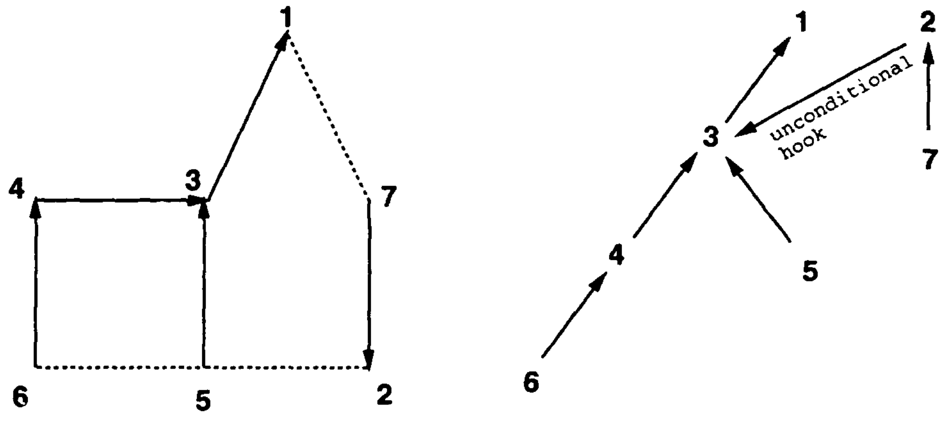


Figure 4: Unconditional hooking

Now take edge (2,5); 2 is a member of a star and *parent*(2) ≠ *parent*(5), so we can hook *parent*(2) or 7 onto *parent*(5) or 3. This temporarily violates the invariant since 2 is less than 3 and is not a leaf. However, if we follow the unconditional hook with a pointer jump, all nodes that were one away from a leaf will become leaves. This will guarantee that the invariant is maintained. As a result, there will never be a chance to introduce a cycle in the tree as long as we follow an unconditional hook with a pointer jump.

Thus, the complete algorithm is:

```
repeat until done
  conditional hook
  unconditional hook
  pointer jump
```

and can be implemented in NESL as:

```
function StarCheck(D) =
  let
    G = D->D
    ST = {D == G: D in D; G in G}
    ST = ST <- {(G,F): D in D; G in G | G /= D}
  in ST->G

function ConComp(D,E1,E2) =
  let
    % Conditional Hook %
    D = D <- {(Di,Dj) : Di in D->E1; Dj in D->E2; Gi in D->(D->E1)
                      | (Gi==Di) and (Di>Dj)}
    % Unconditional Hook %
    D = D <- {(Di,Dj) : Di in D->E1; Dj in D->E2
                      ; ST in StarCheck(D)->E1
                      | ST and (Di /= Dj)}
  in
  if all(StarCheck(D))
    % If done, return D %
    then D
    % If not, shortcut and repeat %
    else ConComp(D->D,E1,E2)
```

In practice we only really have to do the unconditional hook every fourth or fifth time, since it's only there to nail the special case star graphs.

To prove the bounds on the running time we claim that every node gets hooked on each step, and that pointer jumping eliminates half the total depth when summed across all the trees. The exact details of the proof can be found in Jájá [15].

The real trick here is that we only violate the ordering of the tree at the leaves, and since we never hook into leaves we're still guaranteed not to have a cycle in the tree.

2 Minimum Spanning Trees

Suppose now that each edge on the graph has a weight. A minimum spanning tree for a graph is a tree that spans the entire graph whilst minimizing the sum of the edges used in the graph. The crucial idea for getting this to work is to realize that for a given node or supernode (contracted group of nodes), the minimum edge out of that node must belong to the minimum spanning tree. Thus we can use an algorithm similar to the connected components algorithm, but when we hook the edges in the tree, we always pick edges that are the minimum incident on a node, instead of picking random edges.

One way to determine the edge to use would be to use the adjacency list representation for the graph. Then for each node we just do a `min` on the weights and that gives us the edge to use. The problem with this is that the adjacency list representation needs to be updated when doing a contraction. This can be done using scans and is left as an exercise for the reader.

3 Breadth First Search

The work for doing breadth first search is $W(E, V) = O(|E| + |V|)$ and the step complexity is $S(E, V) = O(\textit{width of graph})$. Consider the graph in Figure 5.

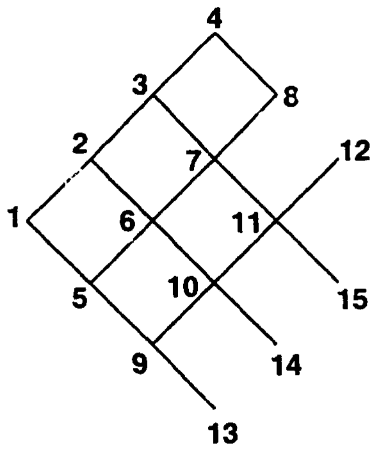


Figure 5: Sample graph

A parallel breadth-first search works much like the serial algorithms. Pick a starting node in the graph (say 1) as the root of the tree, and put this initial node into a set called the frontier set $\{1\}$. As we add a node to the frontier set we mark it as having been visited.

For each node in the frontier set, add the edges that connect that node to any other node which has not yet been visited ((1,2) and (1, 5)) to the tree, and then form a new frontier set with these new nodes $\{2, 5\}$, marking them as having been visited. Repeat the process until all nodes have been visited.

In the serial case this works fine, because in the next step node 6 will get connected to either node 2 or node 5 but not both, since one or the other will come across it first, add the edge to the tree, put the node into the frontier set and mark it as visited. When the other node encounters it will see that it has been visited and not use it.

In the parallel case both node 2 and node 5 could try to add the edge to 6, thus putting node 6 into the frontier set twice. This would quickly cause the algorithm to explode exponentially. Thus, we now have to be careful about how we add nodes to the frontier set, in order to avoid duplicates.

The basic algorithm goes as follows. Represent the graph as an adjacency list—each node has a list of nodes that it's connected to. Then create the frontier set which represent the nodes that we have just discovered. The initial frontier set is just the root node.

Mark all the nodes in the frontier set as having been visited. In parallel, grab all the nodes that are connected to nodes in the frontier set, remove any duplicates, remove any nodes that have already visited, and call this the new frontier set. Repeat this process until the frontier set becomes empty.

In the example above we would have as an initial frontier set $\{1\}$. We would then grab $\{2\ 5\}$ and remove duplicates and nodes that have been visited (none in this case). Then we would grab everything connected to 2 or 5, $\{3\ 6\ 6\ 9\}$, remove duplicates to get $\{3\ 6\ 9\}$, mark these as visited and then grab $\{4\ 7\ 7\ 10\ 10\ 13\}$. To remove duplicates we can simply use an auxiliary array of length $|V|$, which each element writes its index into. So for example, the array $\{4\ 7\ 7\ 10\ 10\ 13\}$ would write 0 into location 4, write 1 and 2 into location 7, and so forth. Now each elements reads the value back from the same location and if the index is not its own, it drops out. For example, either 1 or 2 will be written into location 7 and one of them will read the other's index back, and therefore drop out.

4 Biconnected Components

In a biconnected component, for every two nodes there are at least two independent paths (paths which share no common edge) connecting the nodes. Thus, in the graph in Figure 6 there are two biconnected components; the first contains nodes 1-7 and the second contains nodes 8-11.

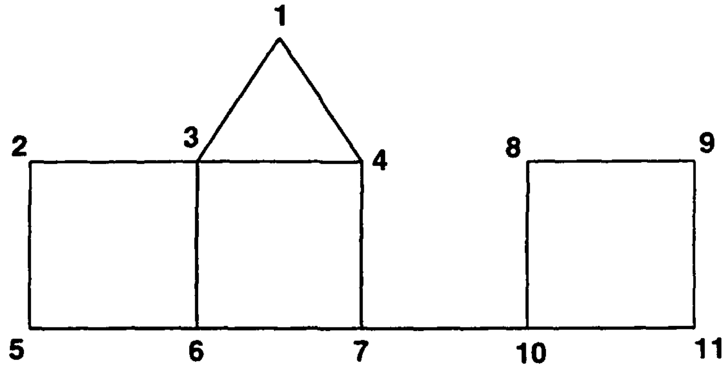


Figure 6: Biconnected components

The parallel algorithm to identify biconnected components uses bits and pieces of everything from connected components, and Euler tours to spanning trees. The basic idea goes as follows:

Find a spanning tree of the graph, and look for cycles by taking each edge not in the tree and seeing if it connects two nodes that are on different branches of the tree. If it does, then all the nodes from the point where the branches connect (the least common ancestor) to the points that the edge connects are in a biconnected set.

The trick is to make sure that the edge connects two nodes in different branches of the tree because only then does the edge create a cycle. Thus the algorithm needs to be able to find the least common ancestor. Details of the algorithm are in Jájá [15]. The limiting step in the algorithm is finding the connected components; thus the step complexity is $S(V, E) = O(\log |V|)$ and the work complexity is $W(V, E) = O(|E| \log |V|)$.

5 For More Information

The $O(\log^2 |V|)$ connected components algorithm is due to Savage and Jájá [31], while the $O(\log |V|)$ version is due to Awerbuch and Shiloach [3]. The parallel breadth-first search is due to Ullman and Yannakakis [34]. A summary of these and other graph algorithms can be found in Chapter 5 of Jájá [15].

15-840: Programming Parallel Algorithms — Lecture #16
Scribe: Margaret Reid-Miller — Tuesday, 10 Nov 92

Overview

- Computational geometry and graphics:
  - Convex hull
  - Closest pair
  - Visibility
  - Line drawing and graphics

1 2-D Convex Hull

Input: A set of points in the plane.
Output: A list of points that form the smallest convex region that surrounds all the points, namely the convex hull (i.e., if nails were placed in a board at the input points, the points of the convex hull would be those nails that an elastic band surrounding all the nails would touch.)

There are many parallel algorithms for finding the 2-D convex hull. Many are based on serial algorithms, and use divide and conquer.

1.1 Quick hull

Quick hull is an algorithm similar to quicksort. The algorithm is:

1. Find the points, x_{min} and x_{max}, with the minimum and maximum x values. These are guaranteed to be on the convex hull.
2. Find the points on the convex hull from amongst the points above and below the line from x_{min} to x_{max} as follows:
   (a) Given a set of points, and two points p_1 and p_2 on the convex hull, find the points on the "outside" of the line formed by p_1 and p_2 by packing those points that have a positive cross product value.
   (b) Find the point p_{max} which is furthest away from this line (i.e., the point with the maximum cross product). p_{max} is guaranteed to be on the convex hull.
   (c) Recursively find the points on the convex hull that lie between p_1 and p_{max} and between p_{max} and p_2.
   (d) Return these sets of points and p_{max} in order.

See the NESL manual for a fuller description and a NESL implementation of the algorithm.

Complexity

As with quicksort, quick hull has complexities of

$$\begin{aligned} S(n) &= O(n) \\ W(n) &= O(n^2) \end{aligned}$$

in the worse case. However, this case is very contrived and would never appear in practice. Consider the case when all the points lie on a circle and have the following unlikely distribution. x_{min} and x_{max} appear on opposite sides of the circle. There is one point that appears half way between x_{min} and x_{max} on the sphere and this point becomes the new x_{max}. The remaining points are defined recursively. That is, the points become arbitrarily close to x_{min} (see Figure 1).

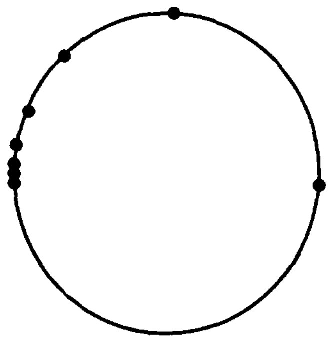


Figure 1: Contrived set of points for worst case quick hull

Since there are no points on the interior of the convex hull and all the points fall to one side of the new p_{max} the problem size is reduced by only one on each recursive call. More realistically, quick hull on average takes

$$\begin{aligned} S(n) &= O(\log n) \\ W(n) &= O(n). \end{aligned}$$

We can show that if the points are randomly placed on the plane then there are a polylogarithmic number of points that lie on the convex hull. Since most of the points lie in the interior many points are removed on each recursive call.

When all the points are randomly distributed on a circle so that none of the points lie in the interior, we can show that on average quick hull takes

$$\begin{aligned} S(n) &= O(\log n) \\ W(n) &= O(n \log n). \end{aligned}$$

1.2 A provably good convex hull algorithm

Consider the following algorithm outline:

1. Presort the points according to their x value.

2. Divide the points evenly into two halves according to their x values.
3. Recursively find the two convex hulls, H_1 and H_2, for the two halves.
4. Sew the two convex hulls together to form a single convex hull H.

Step 4 is the only tricky part. We want to find upper and lower points u_1 and l_1 on H_1 and u_2 and l_2 on H_2 such that u_1, u_2 and l_1, l_2 are successive points on H. The lines b_1 and b_2 joining these upper and lower points are called the upper and lower bridges, respectively. All the points between u_1 and l_1 and between u_2 and l_2 on the "outer" sides of H_1 and H_2 are on the final convex hull, while the the points on the "inner" side are not on the convex hull (see Figure 2). Without loss of generality we only consider how to find the upper bridge b_1. Finding the lower bridge b_2 is analogous.

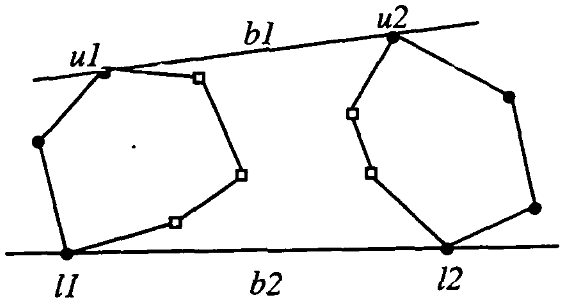


Figure 2: Sewing together two convex hulls

How do we find these upper points? One might think that taking the points with the maximum y value would be a good start. However, u_1 can actually lie as far down as the point with the minimum x or maximum x value (see Figure 3). Thus, taking the point with maximum y value gives us no information.

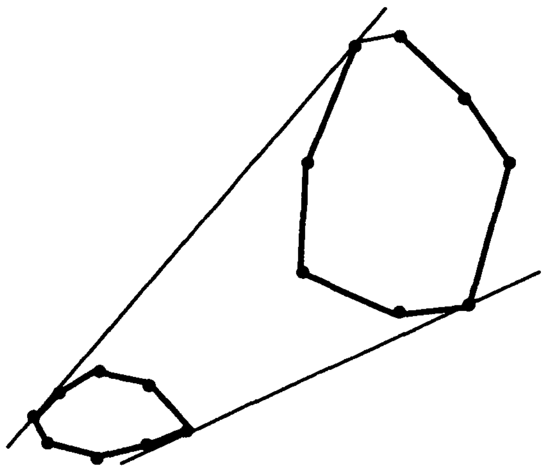


Figure 3: Bridge that is far from the top of the convex hull

Overmars provided a solution based on binary search [22]. Assume that the points on the convex hulls are given in order (e.g., clockwise). At each iteration we will eliminate half the points from consideration in either H_1 or H_2 or both. Let L_1 and R_1 on H_1 and L_2 and R_2 on H_2 be the points with the minimum and maximum x values respectively. Repeat until done:

1. Find the points M_1 and M_2 half way between L_1 and R_1 and between L_2 and R_2 and draw a line between M_1 and M_2
2. If the line just touches both convex hulls then the line is on H and we are done (see Figure 4a).
3. If the line to the left of M_1 intersects H_1, and is tangent to H_2 at M_2, then M_1 is the new R_1 and M_2 is the new L_2 (see Figure 4b).
4. Else if the line to the right of M_1 goes through H_1, and is tangent to H_2 at M_2, then M_1 is the new L_1 and M_2 is the new L_2 (see Figure 4c).
5. Else if the line to the left of M_1 goes through H_1, and the line to the left of M_2 goes through H_2, then M_1 is the new R_1 (see Figure 4d).
6. Else if the line to the right of M_1 goes through H_1, and the line to the left of M_2 goes through H_2, and the tangents through M_1 and through M_2 cross at a point s to the left of the line dividing the points in H_1 from H_2, then M_1 is the new L_1 (see Figure 4e).
7. Else if the line to the left of M_1 goes through H_1, and the line to the right of M_2 goes through H_2, then M_1 is the new R_1 and M_2 is the new L_2 (see Figure 4f).
8. Otherwise repeat steps 3 through 6 taking the mirror image while keeping H_1 on the right and H_2 on the left.

Since this is a binary search $S(n) = O(\log n)$ and $W(n) = O(\log n)$ Once we have found the upper and lower bridges we need to remove the points on H_1 and H_2 that are not on H and append the remaining convex hull points. This takes $S(n) = O(1)$ and $W(n) = O(n)$. Thus the overall complexities are:

$$\begin{aligned} S(n) &= S(n/2) + \log n = O(\log^2 n) \\ W(n) &= 2W(n/2) + \log n + n = O(n \log n). \end{aligned}$$

1.3 Two variations to get $S(n) = O(\log n)$

Variation 1: Divide the problem horizontally into $\sqrt{n}$ subproblems of size $\sqrt{n}$. To sew the hulls together consider all pairs of hulls in parallel. The $n/2$ pairs each take $W(n) = S(n) = O(\log n)$ steps to find their bridges. Next, each hull finds the bridges with the largest absolute angle to the vertical in both directions (i.e., the largest (positive) and smallest (negative) angles); see Figure 5. If the angle of these two bridges is convex then the endpoints of the bridges and any points between the endpoints on the convex hull are on the combined convex hull. Otherwise none of the points on that half of the convex hull are on the combined convex hull. Finding the participating bridges takes $W(n) = O(n)$ and $S(n) = O(1)$. Thus, the overall algorithm complexity is:

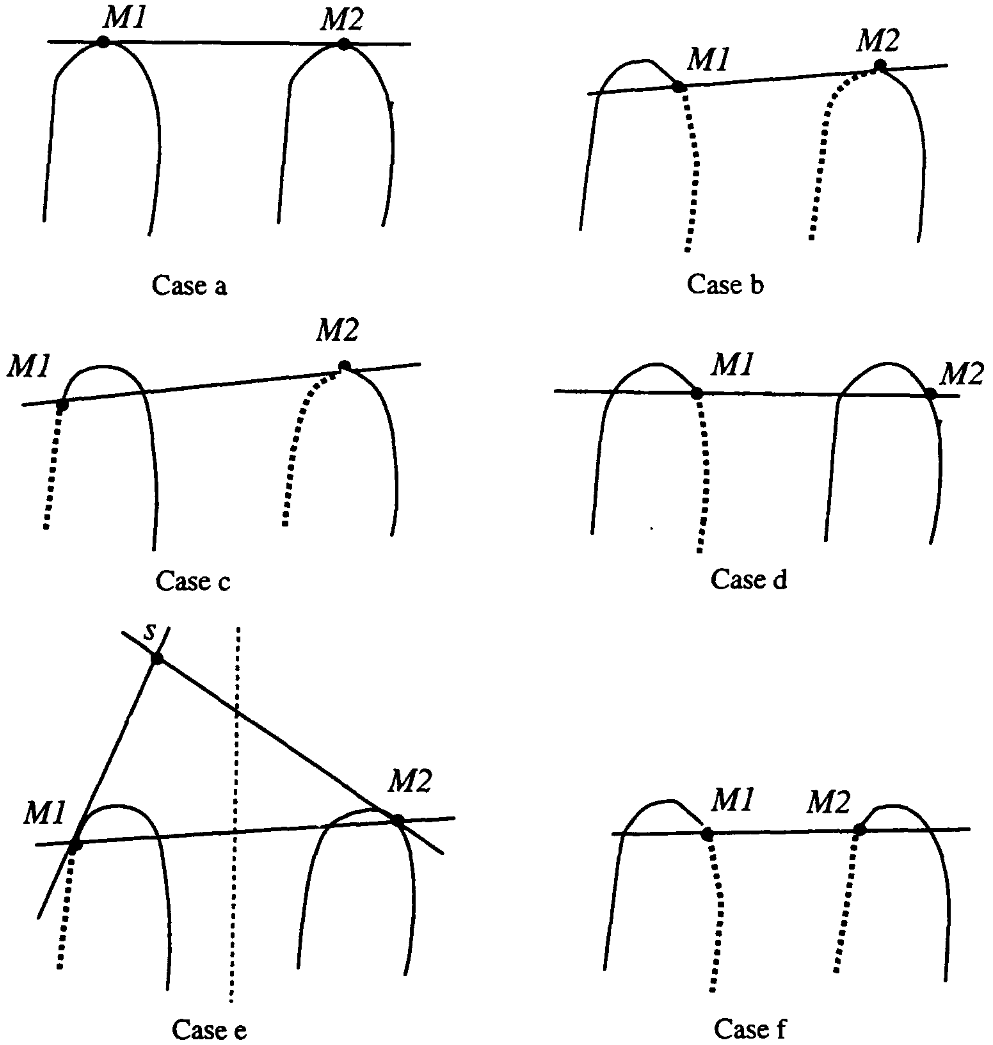


Figure 4: Cases used in the binary search for a bridge. The mirror images of cases b-e are also used.

$$
\begin{aligned}
S(n) &= S(\sqrt{n}) + \log n = O(\log n) \\
W(n) &= \sqrt{n}W(\sqrt{n}) + n\log n \\
&= n\log n + n^{\frac{1}{2}}(n^{\frac{1}{2}}\log n^{\frac{1}{2}}) + +n^{\frac{1}{4}}(n^{\frac{1}{4}}\log n^{\frac{1}{4}}) + \cdots \\
&= n\log n + \frac{n}{2}\log n + \frac{n}{4}\log n + \cdots \\
&= O(n\log n).
\end{aligned}
$$

Variation 2: Use the algorithm in Section 1.2, but find the bridge of two hulls in constant time. Briefly, select the first out of every $\sqrt{m}$ points on each convex hull, where m is the number of points

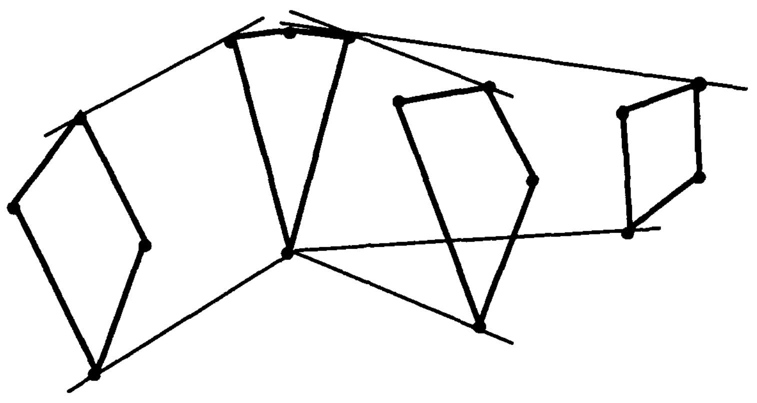


Figure 5: Bridges for the second convex hull

on the hull. It is possible to show that by finding the angles of these $\sqrt{m}$ points of one hull to the $\sqrt{m}$ points in the other hull (m angles total) we can reduce the set of possible bridge points on each hull to two contiguous regions of $O(\sqrt{m})$ points. Then, in parallel, find the angles of every point in one contiguous region to every point in the corresponding region on the other convex hull ($O(m)$ angles), to find the final bridge points. Since m can be as large as $n/2$, the complexity of finding the bridges is $S(n) = O(1)$ and $W(n) = O(n)$. See Jájá [15, pages 264–272] for more details. Thus, the overall convex hull complexity is

$$\begin{aligned} S(n) &= S(n/2) + O(1) = O(\log n) \\ W(n) &= 2W(n/2) + O(n) = O(n \log n). \end{aligned}$$

2 Closest Pair

Input: Set of points on the plane.
Output: The pair of points which are closest together. (This is a special case of the all-closest-pairs problem, which is harder.)

Naive algorithm: Find the distance between every pair of points in parallel and take the minimum. This has $W(n) = O(n^2)$ complexity.

Optimal algorithm: We want an algorithm which has $W(n) = O(n \log n)$. This is optimal because we can show that the unique element problem can be reduced to the closest pair problem [26]. We show a divide and conquer algorithm.

1. Presort the points by their x values and also by their y values.
2. Divide the data by a vertical line into two equal halves and recursively find the closest pair in each half. Suppose that the two pairs are δ_1 and δ_2 apart.

3. The overall closest pair is either one of the two closest pair in either half, or a pair of points that are within $\Delta = \min(\delta_1, \delta_2)$ of the vertical line splitting the two halves. Note that some points may lie on the vertical line itself. Pack the points within this vertical strip of size 2Δ in the y sorted list. In the worse case all the points are within Δ of the vertical line.
4. For each point remaining in the y-sorted list check those points immediately preceding it that are within Δ in the y direction. There are at most 3 points in any $\Delta \times \Delta$ region abutting the vertical line since these points must be at least Δ apart. Therefore, at most 6 of the points preceding a point in the y sorted array could be within Δ away of that point (see Figure 6).

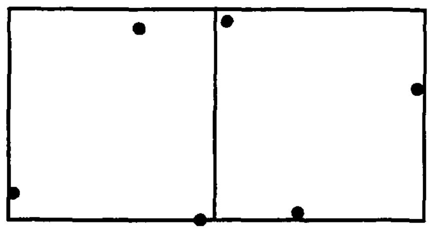


Figure 6: Possible placement of points in a $\Delta \times 2\Delta$ region with one point on the bottom edge

This optimal algorithm has the following complexities:

$$\begin{aligned} S(n) &= S(n/2) + O(1) = O(\ln n) \\ W(n) &= 2W(n/2) + O(n) = O(n \log n). \end{aligned}$$

3 Visibility (line of sight)

Input: $n \times n$ grid of altitudes of a surface, and an observation point on or above the surface.
Output: All the points on the grid visible from the observation point.
Application: Find where to put the city dump so that it is not visible from the Mayor's office.

1. Draw $4n$ rays from the observation point to each border grid point. That is, find all points on the rays in parallel and return a segmented list of coordinates (see [5]).
2. Find the visibility of the points along all $4n$ rays in parallel. That is, for each point:
   (a) Find its altitude.
   (b) Find the distance from the observation point, and given the altitude of the observation point find the angle with the observation point.
   (c) If the angle of the point is greater than the angle of any point between it and the observation point, then the point is visible (i.e., use `max_scan` and if its angle is less than the `max_scan` it is not visible).

3. Permute the visible points back to the grid form. Note that near the observation point several rays will include the same points, but will all show the same visibility results for duplicate points.

It is easy to show that there are a total of at most $2n^2$ points on the $4n$ rays. Therefore,

$$W(n) = O(n^2)$$
$$S(n) = O(1).$$

4 For More Information

The convex hull algorithm we described as variation 1 is due to Aggarwal et al. [1], and variation 2 is due to Atallah and Goodrich [2]. A summary can be found in Chapter 5 of Jájá [15]. The line-of-sight algorithm is from Blelloch [5].

15-840: Programming Parallel Algorithms Lecture #17
Scribe: Jeff Shufelt Thursday, 12 Nov 92

Overview

Computational geometry and basic graphics algorithms:

- Drawing lines
- Contours
- Plane sweep tree

1 Drawing Lines

The problem: given a pair of endpoints on a grid, such as an image or a display screen, light up the points in between to form a line.

The serial algorithms for this problem involve stepping through the pixels from endpoint to endpoint, adjusting the position incrementally as the line drawing proceeds. Bresenham's algorithm, which you can find in any decent graphics textbook, is such an algorithm. The serial algorithms are typically *O(l)*, where *l* is the length of the line in pixels.

We can do line drawing work-efficiently in parallel, in a constant number of steps. The basic idea of the parallel algorithm is to compute the position of each pixel on the line in parallel. In other words, a processor is allocated for each pixel on the line, and each of the processors determines the position of its pixel.

There follows some pseudocode illustrating the basic idea of the algorithm on a pair of points $p_1 = (1,1)$ and $p_2 = (2,6)$. Note that we can compute the length of a line in pixel space by *length* = $\max(\Delta x, \Delta y)$. We can also compute the x and y increments by computing Δx and Δy and computing the proper fraction of each of these to add to an endpoint. If we want both endpoints of a line, we need to create *length* + 1 total points.

```
$p_1 = (1,1)$, $p_2 = (2,6)$
$\Delta x = 1$, $\Delta y = 5$
$length = \max(1,5) = 5$
$i = \text{index}(length + 1)$          [  0    1    2    3    4    5  ]
$step = i/length$                       [ 0.0  0.2  0.4  0.6  0.8  1.0 ]
$xinc = \Delta x \times step$           [ 0.0  0.2  0.4  0.6  0.8  1.0 ]
$yinc = \Delta y \times step$           [  0    1    2    3    4    5  ]
$xline = \text{round}(p_1.x + xinc)$    [  1    1    1    2    2    2  ]
$yline = \text{round}(p_1.y + yinc)$    [  1    2    3    4    5    6  ]
```

The final x and y positions of each pixel are in the *xline* and *yline* vectors, respectively.

Hard-core graphics hackers may be worried by the fact that this algorithm uses floating-point operations. In the early days, much effort was expended in finding line drawing algorithms that only used integer arithmetic, speeding up the line drawing process. However, today's supercomputers (such as the Cray) have floating-point operations that are highly optimized, and these can be faster than the corresponding integer operations.

The algorithm outlined above can be trivially extended to draw multiple lines in parallel, by computing the lengths and increments for each line in parallel, and then applying the algorithm to handle each line simultaneously. This requires a concurrent-write model (i.e., one of multiple values being written to the same location wins), since a pixel might appear in multiple lines. Polygon drawing is an equivalent problem, and the basic idea of the algorithm can also be used to draw circles in parallel (left to the reader as an exercise).

2 Contours

We now turn our attention to a problem that occurs in computer vision. We are given a list of grid points P, containing all points on the boundary of a polygon. We assume that P is sorted in order around the boundary of the polygon (either clockwise or counterclockwise). The problem is to approximate this polygon to a specified degree of accuracy with a set of line segments.

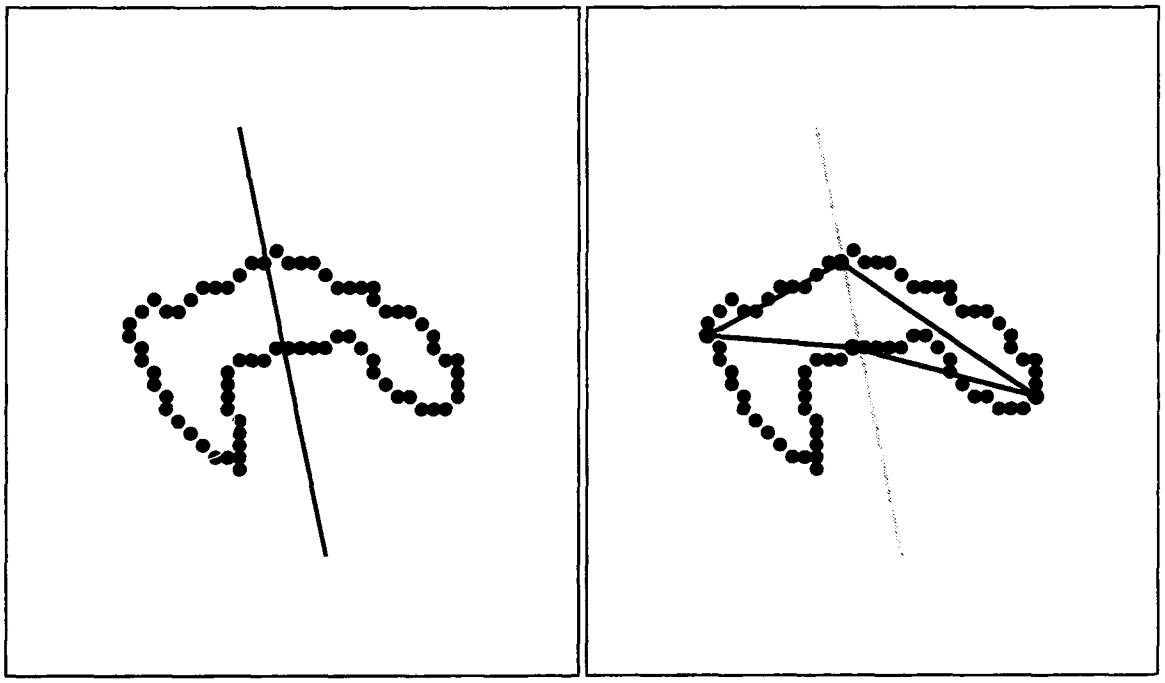


Figure 1: Selection of initial line and first split

This problem is important for computer vision and pattern recognition, since a typical recognition strategy attempts to match the shape of an polygon against a database of objects. Using the point-by-point representation P (typically produced by edge-detectors) leads to greater space requirements and increased search time in the database.

We use a divide-and-conquer approach. First, we generate an arbitrary line through the polygon (one way to do this, if P contains n points, is to select the first point and the $\frac{n}{2}^{th}$ point). We find the point on each side of the line segment that has the greatest distance from the line, and we split our initial line segment twice to form a total of four new lines. Figure 1 shows these first two steps.

We then recursively subdivide the problem. For each line segment, the point furthest from the segment is found, and the segment is split at that point, forming two new line segments. Each line segment continues to be split until the distance from the furthest point to the segment is smaller than some user-specified distance ϵ. Figure 2 shows two more recursive subdivisions.

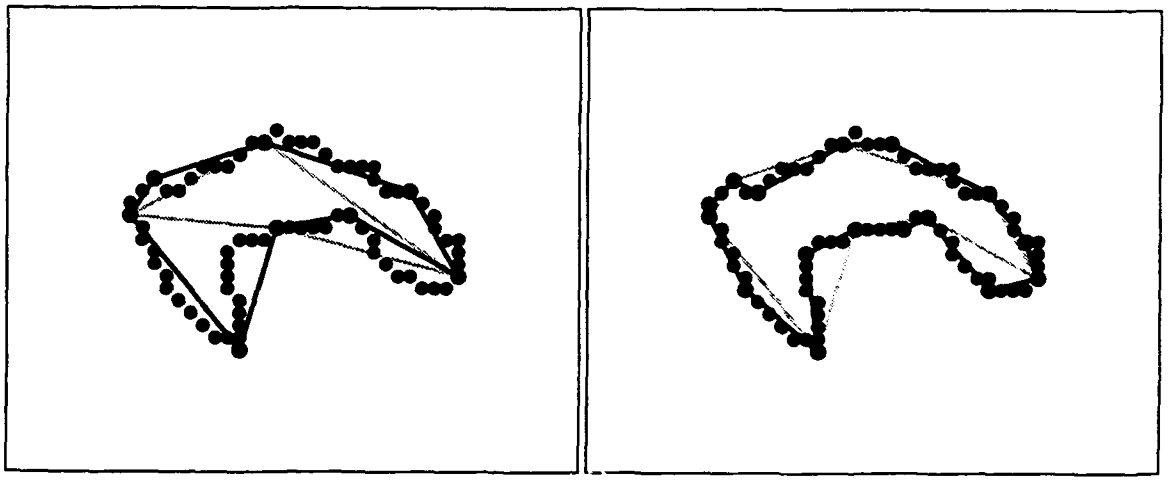


Figure 2: Second and third split

We can see that for a "reasonable" polygon with n points, the algorithm will have step complexity $S(n) = O(\log n)$ time steps and work complexity $W(n) = O(n \log n)$. However, in the worst case (one example being star-shaped polygons with lots of spokes) the algorithm could exhibit complexities of $S(n) = O(n)$ and $W(n) = O(n^2)$.

3 Plane Sweep

A pervasive technique in computational geometry is the *plane sweep* method. In this section, a serial implementation of the technique is described, and a parallel version (referred to as a *plane sweep tree*) is developed, and applied to the problem of finding intersections in a set of horizontal and vertical line segments.

3.1 Serial plane sweep

For the moment, we will be considering the serial algorithm only. To motivate the discussion, we consider the following problem: given a set of horizontal and vertical line segments, find their intersections (for simplicity, assume that the endpoints of these segments have distinct coordinates). The naive serial algorithm would compare each segment against every other segment and check for intersection, resulting in a step complexity of $S(n) = O(n^2)$.

Unfortunately, the standard idea of using a divide-and-conquer approach doesn't help in this problem. Any subdivision of the plane that intersects a segment will leave pieces of the segment on both sides of the line, which means that more comparisons will need to be performed.

Enter the plane sweep technique. The intuition behind the method is this: imagine sweeping a vertical line from left to right across the plane. We only need to consider those positions of the vertical line at which some event happens. For the current application, this means that whenever the sweep line hits the left endpoint of a horizontal segment, the horizontal segment must be added to the list of segments under consideration for intersection; whenever the sweep line hits the right endpoint of a horizontal segment, the segment can removed from consideration for intersections; and when the sweep line hits a vertical line, an intersection test is done between the vertical line and all currently "active" horizontal segments.

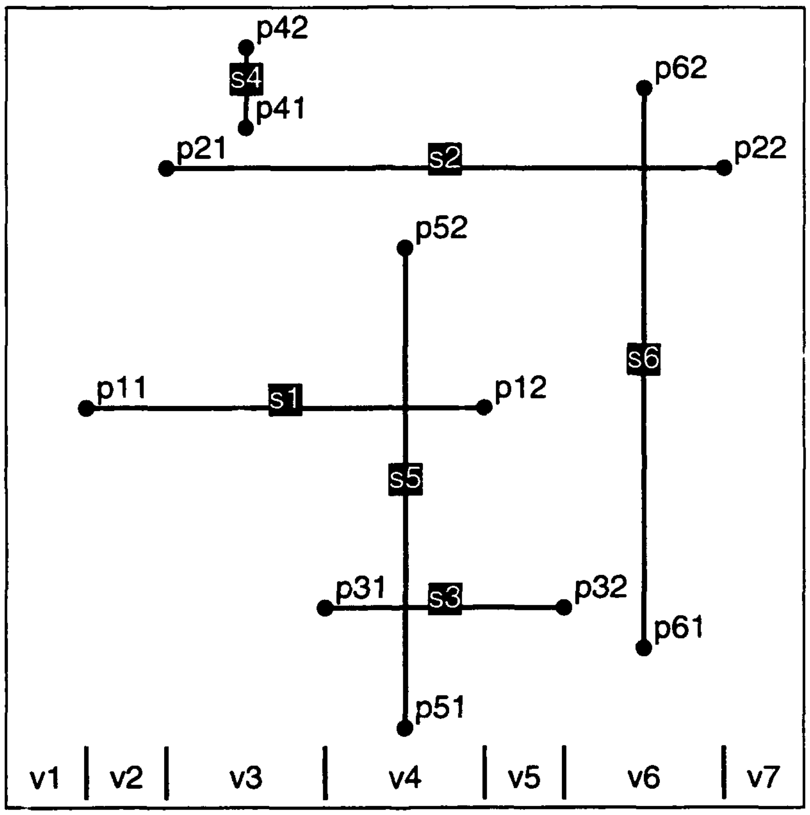


Figure 3: An example set of horizontal and vertical segments

This leads to the following algorithm for the intersection problem. First, sort the endpoints from left to right in the x-direction, and from bottom to top in the y-direction. Then, iterate the sweep line through the sorted x-coordinates. At each x-coordinate, do the following:

1. If the x-coordinate is the left endpoint of a horizontal line segment, `insert` the segment (based on its y-coordinate) into an ordered set.
2. If the x-coordinate is the right endpoint of a horizontal line segment, `delete` the segment (based on its y-coordinate) from the ordered set.
3. If the x-coordinate is the position of a vertical line segment, then `search` the ordered set for the lower and upper y-coordinates on the segment, and return all segments in between as intersections (i.e., a `subregion` operation).

So, at any position of the sweep line, the ordered set only holds those segments which intersect the sweep line, and thus if the sweep line sits on a vertical line segment, then the segment need only be tested for intersection against those segments in the ordered set. As stated above, this can be done by searching the ordered set for the lower and upper points of the vertical line segment, and returning all the segments in the ordered set that fall in between. The following example illustrates the technique with an ordered set T, using the segments in Figure 3.

Sweep line posn.	Action	Ordered elements in T
$-\infty$		$\{\}$
p11	`insert`(T, s1)	$\{$s1$\}$
p21	`insert`(T, s2)	$\{$s1, s2$\}$
s4	`subregion`(T, p41, p42) no intersection	$\{$s1, s2$\}$
p31	`insert`(T, s3)	$\{$s3, s1, s2$\}$
s5	`subregion`(T, p51, p52) intersection: s3, s1	$\{$s3, s1, s2$\}$
p12	`delete`(T, s1)	$\{$s3, s2$\}$
p32	`delete`(T, s3)	$\{$s2$\}$
s6	`subregion`(T, p61, p62) intersection: s2	$\{$s2$\}$
p22	`delete`(T, s2)	$\{\}$

For T, we desire an ordered set that supports `insert` and `delete` operations in $O(\log n)$ time, and the `subregion` operation in $O(\log n + m)$ time, where m is the size of the subregion. We can use one of a variety of balanced trees to accomplish this; examples include 2-3 trees, AVL trees, and red-black trees.

The overall time complexity of this algorithm is then $O(n \log n + i)$, where i is the number of intersections. Of course, there can be $O(n^2)$ intersections, which will lead to quadratic time complexity. We now consider translating this serial algorithm into the parallel domain.

3.2 Plane sweep tree

In the serial case, we imagined a vertical line, sweeping across the plane from left to right, and in our intersection problem, the actual sweeping was performed by iterating through the sorted list of x-coordinates of endpoints of segments. In the parallel case, however, we will simulate the sweep line's movement by generating all possible sweep line events in parallel.

The basic idea is to divide the plane into vertical strips, where the boundaries between strips are determined by the endpoints of the horizontal segments. Each of these strips is represented by a leaf in a height-balanced tree. A non-leaf node represents the union of the strips represented by its children; hence the root represents the entire plane.

Each node maintains two lists, the H and W lists, which are kept in sorted order by the ordinate values of the horizontal segments in the lists. The lists are defined as follows (s_i is a horizontal segment, v is a node in the tree):

$H(v) = \{s_i | s_i$ spans v's vertical strip, but not v's parent's strip$\}$

$W(v) = \{s_i |$ some portion of (a, b) and an endpoint of s_i, where a and b are the endpoints of s_i, is stored in v's vertical strip$\}$

It can be seen that any horizontal interval can be stored in at most $2 \log n$ nodes in this tree; by selecting the nodes that contain the interval in their H lists, and hence for n intervals, there are never more than $n \log n$ items of data in the tree.

To compute $W(v)$ for a leaf v corresponding to an interval $[a, b]$, we observe that $W(v)$ contains at most two segments; one having an endpoint whose abscissa is equal to a, and one having an endpoint whose abscissa is equal to b. For non-leaf nodes, we can simply merge the W lists of the children of those nodes.

To compute $H(v)$, we make use ot the W lists. If v is the root, then we have $H(v) = \emptyset$, since no horizontal segment spans the plane. Otherwise, let z be the parent of v. We test each segment in $W(z)$ to see whether it belongs to $H(v)$ by comparing the x coordinates of its endpoints with $[a_v, b_v]$ and $[a_w, b_w]$, where w is the sibling of v. We can then mark the segments that belong to $H(v)$, and pack each marked copy.

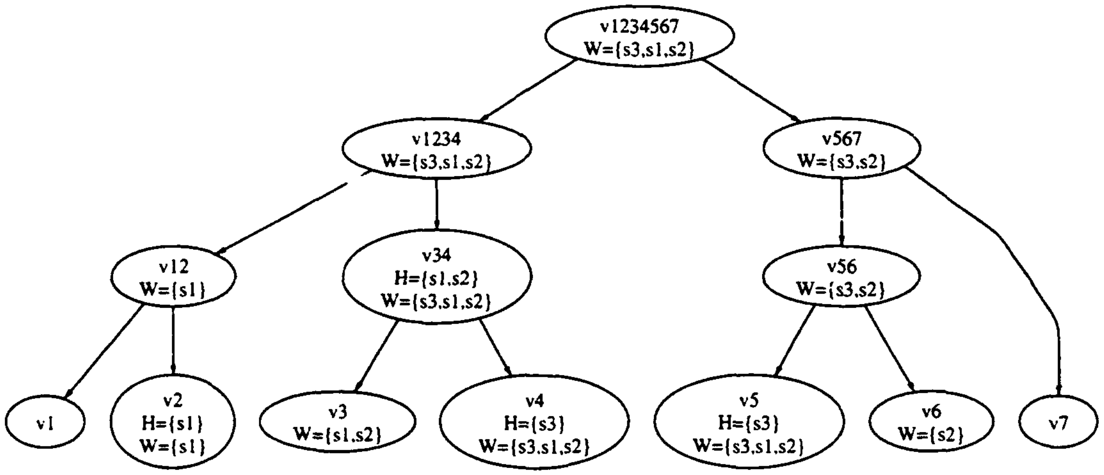


Figure 4: The plane sweep tree for Figure 3

The plane sweep tree for our example is given in Figure 4. Null H and W lists are not shown in the diagram. Note that nodes $v1$-$v7$ correspond to the vertical bands indicated at the bottom of Figure 3.

We state without proof the following theorem; given a set S of n horizontal segments, we can construct the plane-sweep tree of S in $O(\log n)$ steps, using a total of $O(n \log n)$ operations.

Now, for the intersection application, each vertical line is processed in parallel and used as a search key on the plane sweep tree. At every node v encountered during a search in the plane sweep tree, the ordinate of the vertical line is used to do a binary search on $H(v)$. Starting at that point, the vertical line is compared against segments in $H(v)$ until it does not intersect a segment. In this manner, all intersections can be computed in parallel with step complexity $S(n) = O(\log^2 n)$ and work complexity $W(n, i) = O(n \log n + i)$, where i is the number of intersections.

For completeness, we note that it is possible to achieve a solution to the above problem with a plane sweep tree in $O(\log n)$ time. The algorithm requires the use of a pipelined merge sort to compute the W lists, and is quite involved. See Jájá [15, pages 280–283] for details.

4 For More Information

The parallel line-drawing algorithm is from Blelloch [6], and the polygon rendering algorithm is from Salem [30]. Plane sweeping is due to Aggarwal et al. [1], and is discussed in Chapter 6 of Jájá [15].

15-840: Programming Parallel Algorithms | Lecture #18
Scribe: Guangxing Li | Tuesday, 17 Nov 92

Overview

- Review some basic properties of parallel algorithms.
- Introduce a practical parallel algorithm: object recognition.

1 Properties of Parallel Algorithms

Most of the parallel algorithms we have seen so far have the following properties:

1. Fine grained—we divide the problem into small subproblems and solve them concurrently.
2. High communication—we need efficient communication between processors.
3. Dynamic—the properties of the subproblems are very dynamic.
4. Nested parallelism—we recursively use parallelism in subproblems.

Thus, when choosing a parallel language, we have to consider whether it supports these properties. For example, NESL, CM-Lisp and Paralation Lisp support nested parallelism, but Fortran 90 and C* do not.

Example 1 : Random mate for connected components

Recall that in this algorithm, we randomly call half of the nodes parents, and the other half children. Every child that neighbors at least one parent picks one parent to contract into. All edges from the child are carried over to the parent. We can see that the sizes of the subproblems are very dynamic—we only know that on average 1/4 of the vertices will be removed on each contraction. We can also see that communication is very high. Note that in this example there is no nested parallelism.

Example 2 : Breadth first search

This algorithm works as follows. We represent the graph by its adjacency list. Create the frontier set, consisting of the nodes we just encountered, and initialize it to contain the root node. Mark all the nodes in the frontier set as having been visited, then grab all the nodes that are connected to nodes in the frontier set, remove all the duplicates and those nodes marked as having been visited, and call this the new frontier set. Repeat this process until the frontier set is empty.

We can see that the size of the frontier set is very dynamic. It depends on the graph and the root. Although we don't have explicit nested parallelism, we do exploit nested parallelism in the adjacency lists.

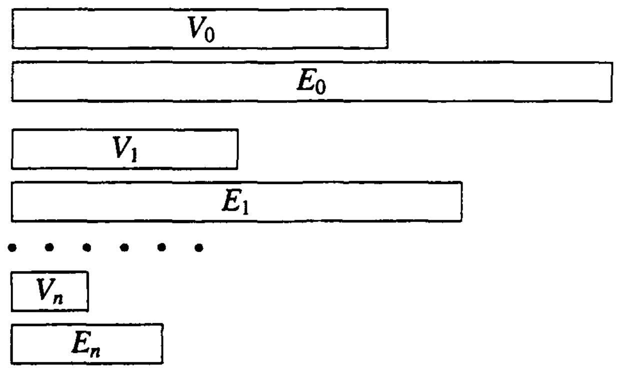


Figure 1: Random mate for connected components.

Example 3 : Quick hull

This is an example with high nested parallelism. In this algorithm, we first find two points x_{min} and x_{max} which are definitely on the convex hull, and then find the points on the convex hull for the points above and below the line connecting x_{min} and x_{max}. The key idea here is that for two points on the convex hull p_1 and p_2, the point which has the greatest distance from the line p_1p_2 must be on the hull. Here we are using nested parallelism. We can also see that it is very dynamic; we have no idea of the size of the sets of points above and below the line p_1p_2.

2 Object Recognition

We start with a set of objects that we wish to identify, and a data base containing all known objects. Rotations and translations are considered invariant. The algorithm has several phases:

1. Edge detection
2. Curve decomposition
3. Corner features
4. Hypothesis generation
5. Hypothesis ordering
6. Verification

The first task is the extraction of features from the image. This is performed in parallel using a pixel-per-processor image representation. A Canny edge detector [11] adapted for parallel execution provides an edge-point map. This works as follows. Each pixel is assigned to a processor, and has a signal associated with it. Processors perform Gaussian convolution in order to filter the image noise and generate the edge candidates. Since each pixel only examines the signals from its neighbors, this can be implemented in parallel. Two gradient magnitude thresholds *high* and

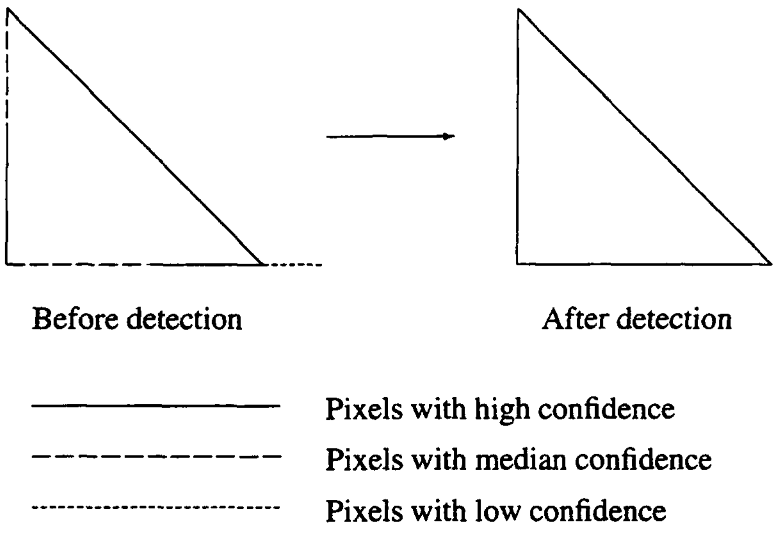

Before detection
After detection
Pixels with high confidence
Pixels with median confidence
Pixels with low confidence

Figure 2: Edge detection

low are computed from an estimate of image noise. Pixels whose gradient magnitude is maximal in the direction of the gradient are selected. In the next step, we eliminate selected pixels with gradient magnitudes below *low*, and retain all pixels above *high*. All pixels with values between *low* and *high* look at their neighbors, and become edges if they can be connected to a pixel above *high* through a chain of pixels above *low*. All others are eliminated. We also call pixels below *low* "with low confidence" and those above *high* "with high confidence" (see Figure 2).

Edge points are linked using a "pointer-jumping" algorithm [14] which assigns each edge-pixel to a labeled set of processors representing connected boundary segments. Within each boundary segment, a parallel curve decomposition algorithm breaks each component into straight line segments [23]. All line pairs are examined in parallel, and corner features are generated wherever line segments intersect within some distance of their end points.

The next step is hypothesis generation. We only consider corners here. As shown in Figure 3, image corner features obtained from the local boundary detection phase are located and returned to the object data base. All model objects in the database, in parallel, examine the image looking for features which match their feature expectations. Whenever such a match is detected, a hypothesis is made stating that an object exists at a specified location and orientation. Once all image features are processed, hypotheses are verified in parallel. This verification takes the form of creating an instance of the model according to the parameters of a spatial transformation specified by each hypothesis, i.e., (x, y, θ), where x, y and θ correspond to the translation and the rotation respectively. The instantiated hypothesis binds an instance of the model to a location and orientation in the scene. This binding projects all features of each model to specified locations and orientations in the scene. Verification may then be performed for all instances in parallel.

3 For More Information

The object recognition system is described in more detail in Tucker et al. [33]. Other vision algorithms can be found in Little et al. [16].

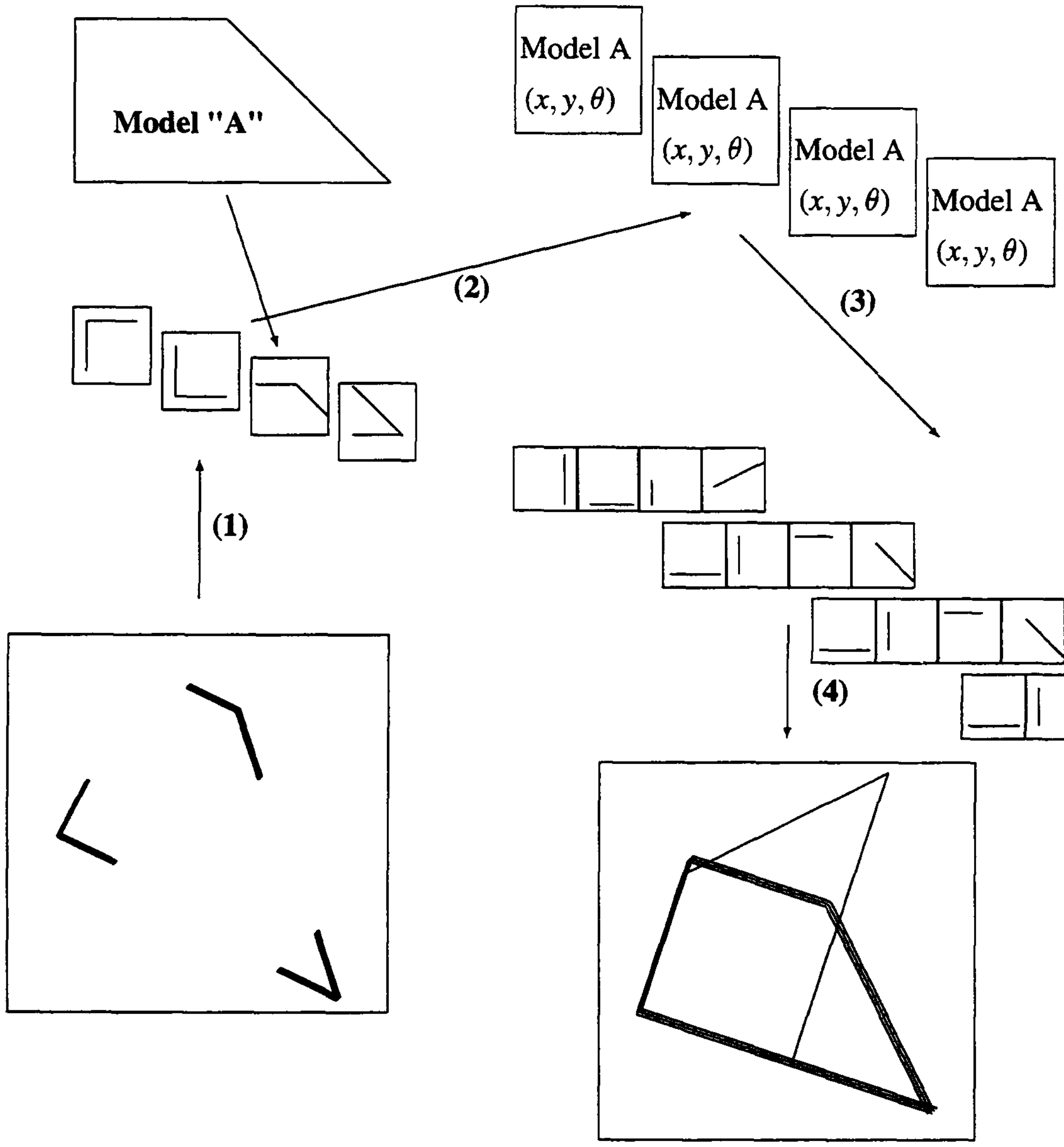


Figure 3: Parallel object recognition steps: (1) Features in the image are matched to features in the model database. (2) Each match generates a hypothesis specifying the location and orientation of an object in the scene. (3) Hypotheses are instantiated by applying the spatial transformation to the set of features associated with the corresponding model. (4) Each instance feature checks for the presence of a corresponding feature in the scene, accumulating evidence for each hypothesis as a whole.

15-840: Programming Parallel Algorithms Lecture #19
Scribe: Jose Brustoloni Thursday, 19 Nov 92

Overview

We will cover the following topics today:

- Fast Fourier Transform (FFT).
- Matrix multiplication.
- Matrix inversion.
- Introduction to sparse matrices.

1 Fast Fourier Transform

The Fourier transform is an analytical tool often used in science and engineering [24]. The continuous Fourier transform of a function $f(t)$ is given by

$$F(w) = \int_{-\infty}^{\infty} f(t)e^{2\pi iwt}dt$$

where $i = \sqrt{-1}$ and $e^{2\pi iwt} = \cos(2\pi wt) + i\sin(2\pi wt)$. Conceptually, the Fourier transform decomposes the signal $f(t)$ into its components at each frequency w.

An approximation to the Fourier transform suitable for computation by digital computers is the so-called discrete Fourier transform (DFT):

$$F_k = \sum_{j=0}^{n-1} f_j e^{2\pi ijk/n}$$

DFTs are extensively used in signal processing [21]. One common application is filtering, e.g. of signals above a certain frequency. The domain of the function transformed is often time (e.g., in sound), but can instead be space (e.g., in image filtering) or some other variable.

DFTs are also used in their own right (not as an approximation to the continuous transform) for polynomial multiplication [12]. The straightforward algorithm to multiply two polynomials of degree n in coefficient form takes $O(n^2)$ time. The DFT of the coefficients of each polynomial gives a point-wise representation of the same polynomial. The point-wise representations allow multiplication in $O(n)$ time. The result is converted back to coefficient form by the inverse discrete Fourier transform. Using a technique called FFT (Fast Fourier Transform), the DFT and its inverse can be computed in $O(n \log n)$ time, giving a total time of $O(n \log n)$ for polynomial multiplication.

The fast Fourier transform is a divide-and-conquer algorithm that computes the discrete Fourier transform. The main idea is to separate the even and odd components and solve recursively two subproblem only half as large, as follows

$$F_k = \sum_{j=0}^{n-1} f_j e^{2\pi ijk/n}$$

$$
\begin{aligned}
&= \sum_{j=0}^{n/2-1} f_{2j} e^{2\pi i k(2j)/n} + \sum_{j=0}^{n/2-1} f_{2j+1} e^{2\pi i k(2j+1)/n} \\
&= \sum_{j=0}^{n/2-1} f_{2j} e^{2\pi i jk/(n/2)} + w^k \sum_{j=0}^{n/2-1} f_{2j+1} e^{2\pi i jk/(n/2)} \\
&= F_k^E + w^k F_k^O
\end{aligned}
$$

where $w = e^{2\pi i/n}$ and F^E and F^O correspond to the DFT computed only on the even or odd elements of the input function, respectively. Note that $F_k = F_{k+n}$, so $F_k^E = F_{k+n/2}^E$ and $F_k^O = F_{k+n/2}^O$. This means that the two recursive calls are each of half the size, since we only need to determine elements up to $F_{n/2}^E$ and $F_{n/2}^O$ and can then just copy them for $n/2 \leq k < n$.

The recurrences for the work and step complexities of the FFT are therefore:

$$
\begin{aligned}
W(n) &= 2W(n/2) + O(n) = O(n \log n) \\
S(n) &= S(n/2) + O(1) = O(\log n)
\end{aligned}
$$

Note that to implement real algorithms with these complexities we either have to precalculate all the w^k, or be somewhat careful about calculating them on the fly.

The FFT can also be implemented in parallel using a fixed circuit of depth $\log n$, each stage consisting of $n/2$ ***butterflies***. Each butterfly is a circuit with two inputs and two outputs that performs one multiplication, one addition and one subtraction.

2 Matrix multiply

Matrix multiplication is easily parallelizable using divide-and-conquer techniques. The idea is to recursively divide each matrix into four submatrices, multiply the submatrices, and add the results, as follows:

$$
\begin{pmatrix} r & s \\ t & u \end{pmatrix} = \begin{pmatrix} a & b \\ c & d \end{pmatrix} \begin{pmatrix} e & g \\ f & h \end{pmatrix}
$$

or:

$$
\begin{aligned}
r &= ae + bf \\
s &= ag + bh \\
t &= ce + df \\
u &= cg + dh
\end{aligned}
$$

This gives 8 independent multiplications of matrices half as large, and 4 matrix additions, each involving $O(n^2)$ elements, where n is the maximum dimension of the matrices being multiplied. Being independent, the multiplications can be done in parallel, followed by the additions. The recurrence for work complexity is

$$
W(n) = 8W(n/2) + O(n^2) = O(n^3)
$$

i.e., the algorithm is work efficient. Since we can do the eight multiplications in parallel, and the addition of two matrices can be done in $O(1)$ steps, the step complexity is

$$S(n) = S(n/2) + O(1) = O(\log n).$$

The algorithm can thus be solved in $O(\log n)$ time using $n^3/\log n$ processors. This is far more processors than typically available if the dimensions of the matrices are more than modest (for $n = 1000$, about 10^8 processors would be called for).

Strassen's algorithm (see Cormen, Leiserson and Rivest [12, pages 739–744]) can also be parallelized. It saves one matrix multiplication over the above scheme, at the expense of more additions. The work complexity is

$$W(n) = 7W(n/2) + O(n^2) = O(n^{\log_2 7})$$

(also work efficient), and the step complexity is the same as above.

Another parallel algorithm for matrix multiplication, due to H.T. Kung [18], uses *systolic computation.* It involves n^2 processors and takes n steps, giving a total of n^3 work. The processors are arranged in a grid mirroring the matrices. At each step, intermediate results are exchanged only with nearest neighbors.

3 Matrix inversion

On serial machines, the complexity of matrix inversion is identical to that of matrix multiplication [12]. In particular, we can show that $T_I(n) = \Omega(T_M(n))$, where $T_I(n)$ is the time to invert a matrix of size $n \times n$ and $T_M(n)$ is the time to multiply matrices of size $n \times n$. Unfortunately an equivalent result for parallel time has not been shown, and inverting a matrix turns out to be harder to do in parallel, work efficiently, than multiplying matrices.

For symmetric, positive-definite matrices, the inverse can be readily computed by LUP decomposition (very similar to Gaussian elimination). Let

$$A = \begin{pmatrix} B & C^T \\ C & D \end{pmatrix}$$

and let $S = D - CB^{-1}C^T$ be the Schur complement of A with respect to B. Then

$$A^{-1} = \begin{pmatrix} B^{-1} + B^{-1}C^TS^{-1}CB^{-1} & -B^{-1}C^TS^{-1} \\ -S^{-1}CB^{-1} & S^{-1} \end{pmatrix}$$

as can be verified by performing the multiplication $AA^{-1} = I_n$.

We can thus convert an inverse of size n (A^{-1}) into two inverses of size $n/2$ (S^{-1} and B^{-1}). One addition and only four multiplications are also required:

- CB^{-1}
- $(CB^{-1})C^T$

- $S^{-1}(CB^{-1})$
- $(CB^{-1})^t(S^{-1}CB^{-1})$

since $B^{-1}C^T = (CB^{-1})^T$ and $B^{-1}C^TS^{-1} = (S^{-1}CB^{-1})^T$. The recurrence for work complexity is thus

$$W_I(n) = 2W_I(n/2) + 4W_M(n/2) + O(n^2)$$

where W_M is the work for matrix multiplication. If $W_M(n) = O(n^{\log_2 7})$, it follows that $W_I(n) = O(n^{\log_2 7})$ as well.

Unlike matrix multiplication, in this algorithm the two matrix inversions cannot be done in parallel. In particular the inversion of B has to be done before we can invert S. The recurrence for step complexity is therefore:

$$\begin{aligned} S_I(n) &= 2S_I(n/2) + 4S_M(n/2) + O(1) \\ &= 2S_I(n/2) + O(\log n) \\ &= O(n) \end{aligned}$$

No work-efficient parallel algorithm with logarithmic step complexity is known for the matrix inversion problem.

The above technique can be generalized to non-symmetric matrices as follows. For any non-singular matrix A, the matrix A^TA is symmetric and positive-definite. The inverse of A can be computed by

$$A^{-1} = (A^TA)^{-1}A^T$$

i.e., multiply the matrix by its transpose, calculate the inverse, and multiply by the transpose again to obtain the inverse of the original matrix.

4 Introduction to sparse matrices

The above sections dealt with *dense* matrices, i.e., matrices where every element is considered. If a large portion of the elements of an array are 0 (or some other constant value), it may be beneficial to treat the matrix as *sparse*. Sparse matrices can be represented as graphs, where there is a node for each row and column, and an edge for each non-zero element, connecting the element's row and column. The value of the element is represented as a weight associated with the edge.

In NESL, sparse matrices can be represented by "compressed rows", very similar to the adjacency list representation for graphs. A vector is defined where each element corresponds to a row in the matrix. These elements are themselves nested vectors, with pairs containing the column and value of each non-zero element in the row.

5 For More Information

The Fast Fourier Transform, and dense matrix operations in general, are discussed in Chapter 8 of JáJá [15].

15-840: Programming Parallel Algorithms — Lecture #20
Scribe: Ken Tew — Tuesday, 24 Nov 92

Overview

Today's topics:

- Sparse matrices.
- Sparse matrix transposition.
- Sparse matrix multiplication.
- An example—finding word correlations.

1 Sparse Matrices

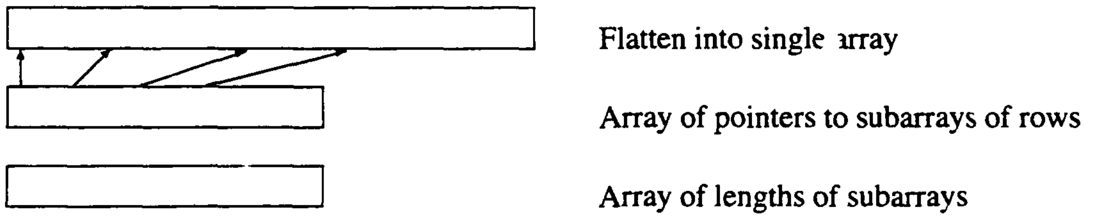
Sparse matrices are represented as adjacency lists. For each non-zero element we store its column index and its value. For example, the matrix:

$$\begin{pmatrix} 0 & 0 & 0 & .6 & 0 & 0 & 2 & 0 & 0 & 0 \\ 0 & .2 & 3 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ .5 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & .7 \end{pmatrix}$$

is represented as:

$$\begin{array}{lccc} \text{Row number:} & 0 & 1 & 2 \\ \begin{array}{l}\text{Column number:}\\ \text{Value:}\end{array} & \begin{pmatrix} 3 & 6 \\ .6 & 2 \end{pmatrix} & \begin{pmatrix} 1 & 2 \\ .2 & 3 \end{pmatrix} & \begin{pmatrix} 0 & 8 & 9 \\ .5 & 1 & .7 \end{pmatrix} \end{array}$$

In a serial implementation this could be stored as an array of linked lists. In a parallel implementation we don't want to use linked lists; instead we could use a flattened array of elements and an array of pointers to the subarrays. This is a standard method of representing sparse matrices in e.g., Fortran:


Flatten into single array
Array of pointers to subarrays of rows
Array of lengths of subarrays

2 Sparse Matrix Transposition

Given a flattened array representation in row-major order the transpose is equivalent to storing the array in column-major order.

To transpose, sort the elements by the column index (keeping track of which row each element came from), and then replace the old column indices with the old row indices. A radix sort would typically be used, since we know in advance the range of the column indices.

3 Sparse Matrix Multiplication

Consider the following example of matrix multiplication:

$$\begin{pmatrix} 5 & 0 & 8 & 0 \\ 0 & 6 & 0 & 0 \\ 9 & 0 & 0 & 1 \\ 0 & 4 & 0 & 0 \end{pmatrix} \times \begin{pmatrix} 0 & 2 & 3 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 4 & 0 & 0 \\ 0 & 0 & 6 & 3 \end{pmatrix} = \begin{pmatrix} 0 & 42 & 15 & 0 \\ 6 & 0 & 0 & 0 \\ 0 & 18 & 33 & 3 \\ 4 & 0 & 0 & 0 \end{pmatrix}$$

3.1 Serial algorithm

Going columnwise through the second matrix, multiply all non-zero columns with the corresponding rows of the first matrix and sum the results. Each sum produces the *(row, column)* element of the result. The work is proportional to the total number of products.

3.2 Parallel algorithm

A standard parallel algorithm for sparse matrix multiplication has the following steps:

1. Transpose the first matrix.
2. Take the outer product of corresponding rows.
3. Sort into groups by rows and columns.
4. Sum up the groups.

Since each row i of the first matrix matches with each column j of the second matrix, transposing the first matrix allows us to work with columns and columns. The outer product gives us the row and column that each multiplication will contribute to. And finally, steps 3 and 4 sum up the contributions for each element of the result.

For example, given the following problem:

$$\left(\begin{pmatrix} 0 & 2 \\ 5 & 8 \end{pmatrix}\begin{pmatrix} 1 \\ 6 \end{pmatrix}\begin{pmatrix} 0 & 3 \\ 9 & 1 \end{pmatrix}\begin{pmatrix} 1 \\ 4 \end{pmatrix}\right) \times \left(\begin{pmatrix} 1 & 2 \\ 2 & 3 \end{pmatrix}\begin{pmatrix} 0 \\ 1 \end{pmatrix}\begin{pmatrix} 1 \\ 4 \end{pmatrix}\begin{pmatrix} 2 & 3 \\ 6 & 3 \end{pmatrix}\right)$$

Step 1. Transpose first matrix:

$$\left(\begin{pmatrix} 0 & 2 \\ 5 & 9 \end{pmatrix}\begin{pmatrix} 1 & 3 \\ 6 & 4 \end{pmatrix}\begin{pmatrix} 0 \\ 8 \end{pmatrix}\begin{pmatrix} 2 \\ 1 \end{pmatrix}\right)$$

Step 2. Take the outer product of corresponding rows:

$$\begin{pmatrix} 0 & 2 \\ 5 & 9 \end{pmatrix} \times \begin{pmatrix} 1 & 2 \\ 2 & 3 \end{pmatrix} = \begin{pmatrix} 0 & 0 & 2 & 2 \\ 1 & 2 & 1 & 2 \\ 10 & 15 & 18 & 27 \end{pmatrix}$$

$$\begin{pmatrix} 1 & 3 \\ 6 & 4 \end{pmatrix} \times \begin{pmatrix} 0 \\ 1 \end{pmatrix} = \begin{pmatrix} 1 & 3 \\ 0 & 0 \\ 6 & 4 \end{pmatrix}$$

$$\begin{pmatrix} 0 \\ 8 \end{pmatrix} \times \begin{pmatrix} 1 \\ 4 \end{pmatrix} = \begin{pmatrix} 0 \\ 1 \\ 32 \end{pmatrix}$$

$$\begin{pmatrix} 2 \\ 1 \end{pmatrix} \times \begin{pmatrix} 2 & 3 \\ 6 & 3 \end{pmatrix} = \begin{pmatrix} 2 & 2 \\ 2 & 3 \\ 6 & 3 \end{pmatrix}$$

Step 3. Sort by rows and columns:

$$\begin{pmatrix} 0 & 0 \\ 1 & 1 \\ 10 & 32 \end{pmatrix} \begin{pmatrix} 0 \\ 2 \\ 15 \end{pmatrix} \begin{pmatrix} 1 \\ 0 \\ 6 \end{pmatrix} \begin{pmatrix} 2 \\ 1 \\ 18 \end{pmatrix} \begin{pmatrix} 2 & 2 \\ 2 & 2 \\ 27 & 6 \end{pmatrix} \begin{pmatrix} 2 \\ 3 \\ 3 \end{pmatrix} \begin{pmatrix} 3 \\ 0 \\ 4 \end{pmatrix}$$

Step 4. Sum the groups:

$$\begin{pmatrix} 0 \\ 1 \\ 42 \end{pmatrix} \begin{pmatrix} 0 \\ 2 \\ 15 \end{pmatrix} \begin{pmatrix} 1 \\ 0 \\ 6 \end{pmatrix} \begin{pmatrix} 2 \\ 1 \\ 18 \end{pmatrix} \begin{pmatrix} 2 \\ 2 \\ 33 \end{pmatrix} \begin{pmatrix} 2 \\ 3 \\ 3 \end{pmatrix} \begin{pmatrix} 3 \\ 0 \\ 4 \end{pmatrix}$$

Step 3 is the bottleneck in this algorithm, since the number of items sorted is the number of products calculated, which can be bigger than the size of the matrix. By using an $O(n)$ sort (e.g., radix sort) we can get a work-efficient algorithm. A step complexity of $O(\sqrt{n})$ is relatively easy to achieve; $O(\log n)$ is harder.

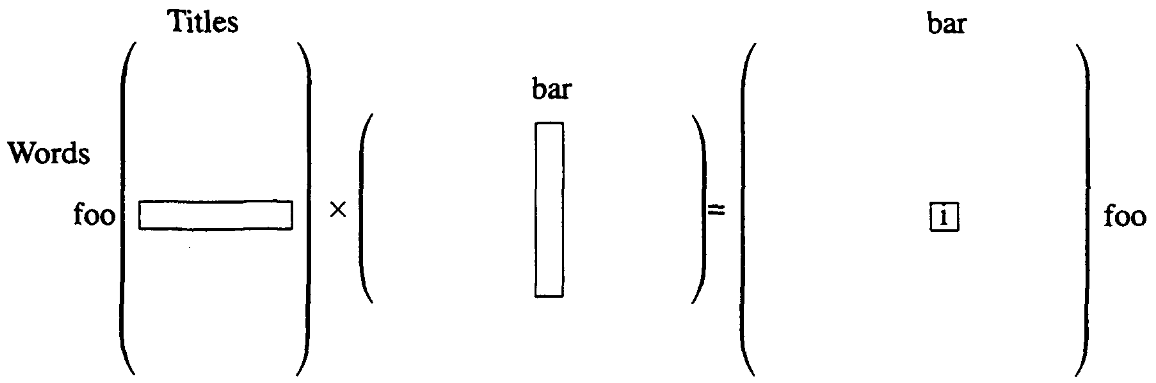
A problem for large matrices is memory consumption. Step 2 can take up to n^3 memory. In practice, if the size of step 2 is too large then the outer products are broken up into subsets and solved one at a time.

4 Example: Word Correlation

Given a set of article titles and a set of words, we want to know the correlation amongst words that appear in article titles. The obvious solution is to create a (very) sparse matrix with article titles for columns and words for rows. If a word appears in the title that element contains a 1, otherwise a 0.

$$\begin{array}{cc|cccccccc|}
 & & \multicolumn{8}{c}{\text{Article Titles}} \\
 & & 0 & 1 & 2 & 3 & . & . & . & n \\
\hline
 & 0 & & & 1 & & & & & 1 \\
 & 1 & & 1 & & 1 & & & & \\
 & 2 & 1 & & & & & & & \\
\text{Words} & 3 & & 1 & & & & & & 1 \\
 & . & & & & & & & & . \\
 & . & & & & & & & & . \\
 & . & & & & & & & & . \\
 & m & 1 & & 1 & & . & . & . & \\
\hline
\end{array}$$

In our example, n is approximately 50,000 and m is approximately 10,000. To count the number of times combinations of words appear together in a title multiply the above matrix by its transpose:



Thus, i is the number of times *foo* and *bar* appear in the same title.

15-840: Programming Parallel Algorithms Lecture #21
Scribe: Wayne Sawdon Tuesday, 1 Dec 92

Overview

We will cover the following topics today:

- Random numbers
- Random permutations
- Fortran 90
- Case study: ID3

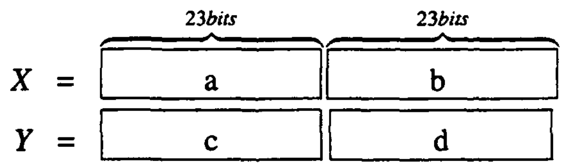
1 Random numbers

Pseudorandom numbers can be generated in parallel by computing powers of large prime numbers modulo another large number. For example, primes generated via $((5^{13})^i \bmod 2^{46})/2$ are proven to generate all 2^{45} values. The numbers used are chosen as follows: $(5^{13}) \approx 2^{30}$, which is close to the size of a word. 2^{46} is the size of a double precision floating point mantissa, and $(5^{13})^2 > 2^{46}$. The final $/2$ is necessary to drop the low order bit, which is always 1.

A set of pseudorandom numbers can be generated as follows:

1. Generate index vector i
2. Calculate pseudorandom number of each i

By combining the modulo operation with raising 5^{13} to the i^{th} power we can reduce the work necessary to generate each value. Each 46-bit multiplication can be computed as follows:



$$X \times Y = ((ad + cb) << 23) + bd$$

2 Random permutations

The easiest way to generate a random permutation of a set of numbers is to generate a random number for each element, and then rank the elements according to their assigned numbers. For this to generate a random permutation, we must guarantee there are no duplicates in the random numbers. For a set of n elements, the random numbers should be generated in the range $[0 \ldots n^2]$ to minimize the possibility of duplicates. The resulting numbers must still be checked and duplicates eliminated by either regenerating the entire set of random numbers or by regenerating only those that were duplicates. Thus, the procedure is as follows:

1. Generate a random number for each element in the set.
2. Eliminate duplicates.
3. Rank result using radix sort.

This algorithm has complexity $W(n) = O(n)$ and $S(n) = O(\sqrt{n})$.

3 Fortran 90

```
Subroutine PlusScan(A,N)
  Integer S,N,A(N),T(N)

  S = 1
  Do While (S .LE. N/2)
    A[2*S:N:2*S] = A[S:N:2*S] + A [2*S:N:2*S]
    S = S*2
  End Do

  A[N] = 0
  S = N/2
  Do While (S .GE. 0)
    T[2:N:2*S] = A[2*S:N:2*S]
    A[2*S:N:2*S] = T[S:N:2*S] + A[2*S:N:2*S]
    S = S/2
  End Do
  Return
End
```

Figure 1: Fortran 90 code for `plus_scan`.

Figure 1 shows the Fortran 90 code for a parallel prefix function (i.e., the equivalent of the NESL `plus_scan` operator). Fortran 90 is the language used by most supercomputers for parallel and vector operations. It is important to note the differences between it and NESL. Fortran 90 provides an implicit extension of + to become a vector operation. In NESL, + must be explicitly extended by an apply-to-each form to become a vector operator. Likewise, Fortran 90 automatically converts scalar values to vectors for vector operations.

Figure 2 gives a sample execution of the Fortran 90 `PlusScan` routine. The first line shows the input to the routine. The next three lines show the results of the up-sweep. The last four lines show the results of the down-sweep, with the final line being the result values.

Figure 3 shows the Fortran 90 code for `Quicksort`. Unlike NESL, Fortran 90 offers no nested parallelism. The only functions which can be executed in parallel are those that are built-in to the language. Thus the Fortran 90 version of `Quicksort` calls itself recursively `N` times, and each

$$
\begin{array}{cccccccc}
[\,\boxed{3} & \boxed{1} & \boxed{7} & \boxed{0} & \boxed{4} & \boxed{1} & \boxed{6} & \boxed{3}\,] \\
[\,3 & \boxed{4} & 7 & \boxed{7} & 4 & \boxed{5} & 6 & \boxed{9}\,] \\
[\,3 & 4 & 7 & \boxed{11} & 4 & 5 & 6 & \boxed{14}\,] \\
[\,3 & 4 & 7 & 11 & 4 & 5 & 6 & \boxed{25}\,] \\
\\
[\,3 & 4 & 7 & 11 & 4 & 5 & 6 & \boxed{0}\,] \\
[\,3 & 4 & 7 & \boxed{0} & 4 & 5 & 6 & \boxed{11}\,] \\
[\,3 & \boxed{0} & 7 & \boxed{4} & 4 & \boxed{11} & 6 & \boxed{16}\,] \\
[\,\boxed{0} & \boxed{3} & \boxed{4} & \boxed{11} & \boxed{11} & \boxed{15} & \boxed{16} & \boxed{22}\,]
\end{array}
$$

Figure 2: Sample execution of Fortran 90 `PlusScan`.

call is performed serially. For a language to support nested parallelism, it must support segmented scans and segmented operations.

Also note that the parameters to `Quicksort` must be more than the array length and a pointer to the first element. The compiler must know how the data is distributed among the processors in order to balance the work load.

4 Case study: ID3

ID3 is an AI learning algorithm that builds decision trees for the classification of an object based on the object's attributes. In ID3, the decision tree is built automatically by training the algorithm on a subset of the data. Once the tree is built, the remaining data can then be classified. The operation of ID3 is best explained with an example. Details of ID3 can be found in Quinlan [27].

One use of ID3 is for word pronunciation. The correct phoneme for the pronunciation of a letter can be based upon the previous three letters and subsequent three letters in the word. The training data for ID3 is a set of tuples, each containing the letter being pronounced, the prior three letters, the subsequent three letters, and the phoneme for the correct pronunciation of the letter. The decision tree is built by examining the statistical correlation between each of the letter positions and the phoneme. The position with the highest correlation to the phoneme is placed at the root of the decision tree. Typically, this is the letter being pronounced. Then, for each letter in the alphabet, the position with the next highest correlation to the phoneme is determined. This is stored in the decision tree as a child of the root node, with one child node for each letter in the alphabet. The process is repeated and children are added to the tree until there is no longer a statistical correlation between positions and the phoneme. An example decision tree is shown in Figure 4.

ID3 provides a tremendous opportunity for parallelism. The input data must be sorted based upon the character in each position and upon the output phoneme. The statistical correlation of

```
Subroutine Quicksort(A,N)
  Integer N,Nless,Less(N),Greater(N),A(N)

  If (N .LE. 1) Return

  Pivot = A(1)
  Nless = Count(A .LT. Pivot)
  Less = Pack(A, A .LT. Pivot)
  Greater = Pack(A, A .GE. Pivot)

  Call Quicksort(Less, Nless)
  A[1:Nless] = Less
  Call Quicksort(Greater, N - Nless)
  A[Nless+1:N] = Greater

  Return
End
```

Figure 3: Fortran 90 code for `Quicksort`.

positions to the phoneme requires computing the "entropy" of each data value and provides two additional levels of nested parallelism. In practice, ID3 provides more parallelism than can be exploited on current machines, and parts of the algorithm must be run serially. The limiting factor for the amount of parallelism is the memory capacity of the parallel machine.

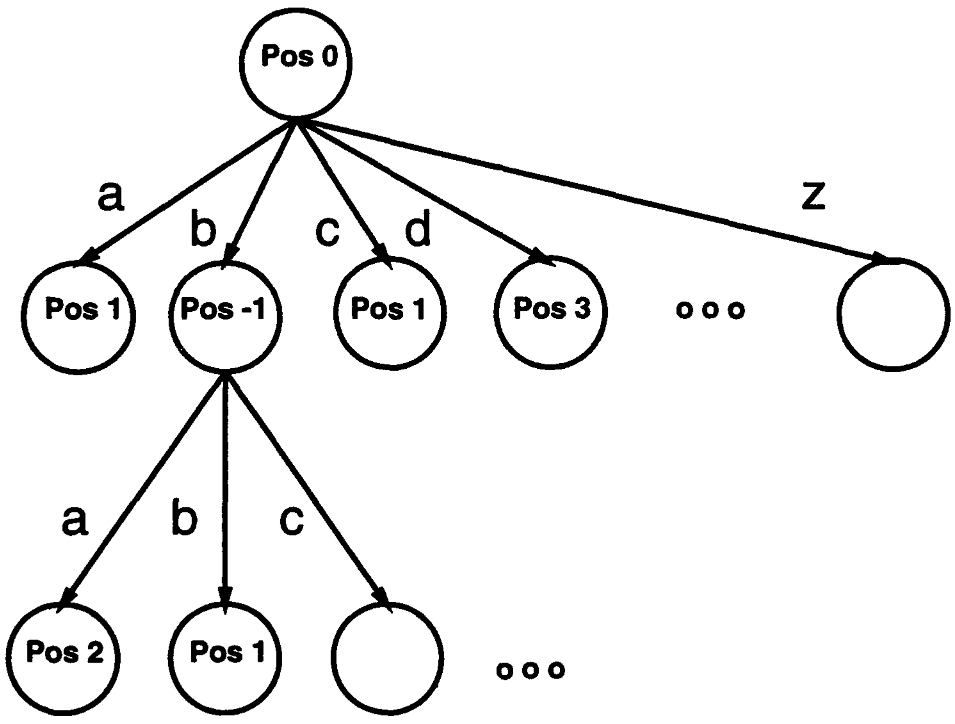


Figure 4: Example ID3 phoneme decision tree.

Assignment 1

The first assignment is to design a parallel algorithm that finds all the primes. The input to the algorithm should be a positive integer n, and the output should be a packed array with all the primes less than n. This algorithm must be work efficient, this is to say

$$T_P(n) = O(T_s(n)/P)$$

where $T_P(n)$ is the time taken by the parallel algorithm on P processors and $T_s(n)$ is the time taken by the serial sieve algorithm (C code for the serial algorithm is given below). The goal is to minimize the parallel running time $T_P(n)$ under the above constraint. The model you must use is a concurrent-read concurrent-write (CRCW) parallel random access machine (PRAM). You can assume the serial algorithm runs on the Random Access Machine (RAM) model. The description of your algorithm must be no more than two pages. The best solution (smallest $T_P(n)$ with a reasonable description) gets a free capuccino and croissant at the graduate student coffee house. You are welcome to discuss the problem with other students, but must write up your own solution.

```
int primes(int result[], int length)
{
  int i, j, sqln;
  /* set all flags to 1 */
  for (i = 0; i < length; i++) result[i] = 1;
  /* 0 and 1 are not primes, by definition */
  result[0] = 0;  result[1] = 0;
  sqln = sqrt((double) length);
  for (i = 2; i < sqln; i++) {
    if (result[i])
      /* i is a prime:  mark all elements that are a multiple of i */
      for (j = i+i; j < length; j += i)
        result[j] = 0;
  }
  /* pack the indices of all the primes */
  for (i = 0, j = 0; i < length; i++) {
    if (result[i]) result[j++] = i;
  }
  return j;
}
```

Assignment 2

Put your solutions to the following problems into a single file and save it into the file `/afs/cs/project/classes-guyb/handin/username/assign2.nesl`, where `username` is your username. If you don't have access to AFS, send me your solution via email.

Problem 1: Implement your own version of the NESL integer `plus_scan` function with work complexity $O(n)$ and step complexity $O(\lg n)$. Don't use any of the built-in scan operations. You can assume n is a power of 2. Remember that the NESL `plus_scan` function actually does a prescan. For example:

```
    my_plus_scan([8, 2, 9, 1, 3, 5, 2])
 ⇒  [0, 8, 10, 19, 20, 23, 28] : [INT]
```

You might find the functions `odd_elts`, `even_elts`, and `interleave` useful. The first two return the odd and even elements from a vector, respectively, and the third interleaves two equal length vectors.

Problem 2: Implement your own version of the NESL `pack` function using your `plus_scan` with work complexity $O(n)$ and step complexity $O(\lg n)$ (or better). Assume that it only needs to work on vectors of integers.

Problem 3: The stock market problem is as follows: given the price of a stock at each day for n days, determine the biggest profit you can make by buying one day and selling on a later day. For example:

```
    stock([12, 11, 10, 8, 5, 8, 9, 6, 7, 7, 10, 7, 4, 2])
 ⇒  5 : INT
```

since we can buy at 5 on day 5 and sell at 10 on day 11. This has a simple linear time serial solution. Write a NESL program to solve the stock market problem with work complexity $O(n)$ and step complexity $O(\lg n)$.

Problem 4: Write a routine to sort n numbers in NESL with work complexity $O(n^2)$ and step complexity $O(1)$. You can assume that each key is unique (only appears once).

Problem 5: (Extra credit.) Implement a segmented version of the `plus_scan` function. It should take a vector of flags as well as the vector of values and start over wherever the flag is set. The position where the flag is set should include the sum of the previous segment. For example:

```
    seg_plus_scan([7, 2, 9, 1, 5, 3, 6, 4], [t, f, f, t, f, t, f, f])
 ⇒  [0, 7, 9, 18, 1, 6, 3, 9] : [INT]
```

You can assume the length is a power of 2. The work and step complexities should be the same as in Problem 1.

Assignment 3

Put your solutions to the following problems into a single file and save it into the file `/afs/cs/project/classes-guyb/handin/username/assign3.nesl`, where `username` is your username. If you don't have access to AFS, send me your solution via email.

Problem 6: Implement a dictionary in NESL with the following operations. Note that the required complexity bounds are expected-case.

- `MakeDictionary(n)` `int -> dictionary`
  Make a dictionary which will contain at most n elements. This should be done with $O(n)$ work and $O(\lg n)$ steps.
- `Insert(keys, dictionary)` `[int], dictionary -> dictionary`
  Insert a vector of m keys into the dictionary. This should be done with $O(m)$ work and $O(\lg m)$ steps.
- `Search(keys, dictionary)` `[int], dictionary -> [bool]`
  Take a vector of m keys and return a vector of booleans with a T is every position in which the key was in the dictionary. This should be done with $O(m)$ work and $O(\lg m)$ steps.

Here is an example use:

```
    let d1 = MakeDictionary(100);
        d2 = Insert([11, 25, 2, 4], d)
    in Search([2, 10, 24, 11, 5], d2)

⇒   [t, f, f, t, f] : [BOOL]
```

Problem 7: Implement a function `CountCycles` that takes a permutation and returns the number of cycles in the permutation. The permutation should be represented as a length n vector of indices less than n with no index repeated. For example:

```
    CountCycles([1, 2, 0, 4, 3])

⇒   2 : INT

    CountCycles([4, 2, 1, 3, 8, 0, 11, 6, 7, 5, 10, 9])

⇒   4 : INT
```

It should run with $O(n)$ work and $O(\lg n)$ steps.

Final Projects

The final assignment due in this course is a project. Each project should be an implementation of a parallel algorithm or set of parallel algorithms. Everyone can work individually or in groups of 2 (if you work in a group of 2, the project should be appropriately larger). Here are some suggestions for projects. You are welcome to do any of these or to choose your own.

1. Implement and compare some optimal string searching algorithms.
2. Implement a parser for NESL. It should return a tree structure that represents the parse tree of the program.
3. Implement algorithms for some problems in computational geometry, such as all-closest-pairs, convex-hull or visibility.
4. Implement list-ranking using deterministic coin flipping.
5. Implement algorithms for some graph problems, such as maximum-flow, biconnected-components or minimum-spanning-tree.
6. Implement algorithms for manipulating "bignums" (numbers with arbitrary precision). This should include addition, multiplication, and division.
7. Implement expression evaluation for arbitrary expressions with addition and multiplication. (This would use tree contraction.)
8. Implement finding the least-common-ancestor of a tree and other related algorithms on trees.
9. Implement some algorithms for various problems on matrices.

Feel free to come talk to me about your ideas. I can give your information on any of the above mentioned problems. You will be expected to turn in the following:

1. A working implementation of an algorithm or group of algorithms. You are encouraged to implement your project in NESL, but if you have a particular parallel machine in mind, you are welcome to do it in some language that maps directly onto the machine.
2. A writeup describing the algorithms and data-structures you used, and interesting implementation techniques. This should also include a discussion of whether you think it is a practical algorithm and things you would do differently if you were to do it again.

The project is due on December 10. You also need to hand in a 1/2 page project description by November 3, and a 2-3 page detailed project description by November 17.

List Of Final Projects

Jose Brustoloni	Max-flow
Sergio Campos	Lowest common ancestor
Ming-Jen Chan	Convex hull
Denis Dancanet & Jeff Shufelt	Pipelined mergesort and plane sweep tree
Jürgen Dingel	String matching
David Eckhardt	Parallel parsing
Andrzej Filinski	Bignum arithmetic
Jonathan Hardwick	NAS benchmarks
Darrel Kindred	String matching
Corey Kosak	Image analysis
Guangxing Li	Tree contraction
Qingming Ma	Convex hull
Richard McDaniel	Biconnected components
Arup Mukherjee	Expression evaluation
Chris Okasaki	Graph operations
John Pane	3D volume rendering
Zoran Popović	Closest pair
Wayne Sawdon	String matching
Erik Seligman	Deterministic list ranking
Sriram Sethuraman	Singular value decomposition
Ken Tew	EEG analysis
Eric Thayer	Speech recognition
Xuemei Wang & Bob Wheeler	Matrix operations
Matt Zekauskas	FFT and polynomial multiplication
Xudong Zhao	Ordered binary decision diagrams

Course Announcement

Computer Science 15-840B

Programming Parallel Algorithms

Fall Semester 1992

Instructor: Guy E. Blelloch
Time: Tuesday and Thursday, 10:30-Noon
First Class: Tuesday, September 15
Place: Wean Hall 5409
Credit: 1 Graduate Core Unit (Negotiable for Undergraduates)

Course Description:

This course will be a hands on class on programming parallel algorithms. It will introduce several parallel data structures and a variety of parallel algorithms, and then look at how they can be programmed. The class will stress the clean and concise expression of parallel algorithms and will present the opportunity to program non-trivial parallel algorithms and run them on a few different parallel machines. The course should be appropriate for graduate students in all areas and for advanced undergraduates. The prerequisite is an algorithms class. Undergraduates also require permission of the instructor.

There will be no text, but some course notes and papers will be distributed. Grading will be based on a set of programming assignments and a class project. We will use the programming language NESL. NESL runs on the Cray Y-MP, and Connection Machine CM-2 as well as on standard workstations. Class accounts will be available at the Pittsburgh Supercomputing Center (PSC) for using the Cray and Connection Machine. NESL should also be running on the CM-5 sometime during the semester.

Class Outline

Here is a rough outline of what we will cover during the semester.

1. Parallel Machine Models

   Parallel Random Access Machine (PRAM)
   Vector Random Access Machine (VRAM)
   Brent's Theorem
   Complexity Measures
   Message Passing Models

2. The NESL Language

   Parallel Primitives
   Nested Parallelism
   Typing Scheme
   Combining Complexity

3. Operations on Sequences and Sets

   Summing and Scans
   Selecting (Pack)
   Append, Reverse
   Removing Duplicates
   Union, Intersection and Difference
   Power Sets

4. Sorting, Merging and Medians

   Selection Sort
   Quicksort and Quickmedian
   Radix Sort
   Halving Merge and Mergesort
   Sorting nested structures
   Optimal Median

5. Operations on Strings

   String matching
   Breaking Text Into Words and Lines
   Parenthesis Matching
   Hashing and Searching
   Line Breaking

6. Linked Lists and Trees

   List Ranking

Euler Tour Order of a Tree
Rootfix and Leaffix Operations on a Tree
Deleting Vertices from a Tree

7. Representing and Manipulating Graphs

   Representations
   Maximal Independent Set
   Breadth First Search
   Connected Components
   Minimum Spanning Tree
   Biconnected components

8. Computational Geometry and Graphics

   Quickhull
   Merge Hull
   Closest Pair
   Figure Drawing
   Hidden Surface Removal
   Clipping

9. Sparse and Dense Matrices

   LUD Decomposition
   Sparse Linear Systems
   Simplex
   Strassen's Matrix Multiply

References

[1] Alok Aggarwal, Bernard Chazelle, Leo Guibas, Colm Ò'Dùnlaing, and Chee Yap. Parallel computational geometry. In *Proceedings Symposium on Foundations of Computer Science*, pages 468–477, October 1985.

[2] Mikhail J. Atallah and Michael T. Goodrich. Efficient parallel solutions to geometric problems. In *Proceedings International Conference on Parallel Processing*, pages 411–417, 1985.

[3] Baruch Awerbuch and Yossi Shiloach. New connectivity and MSF algorithms for Ultracomputer and PRAM. In *Proceedings International Conference on Parallel Processing*, pages 175–179, 1983.

[4] Kenneth E. Batcher. Sorting networks and their applications. In *AFIPS Spring Joint Computer Conference*, pages 307–314, 1968.

[5] Guy E. Blelloch. Prefix sums and their applications. Technical Report CMU-CS-90-190, School of Computer Science, Carnegie Mellon University, November 1990.

[6] Guy E. Blelloch. *Vector Models for Data-Parallel Computing*. MIT Press, 1990.

[7] Guy E. Blelloch. NESL: A nested data-parallel language. Technical Report CMU-CS-92-103, School of Computer Science, Carnegie Mellon University, January 1993. (Updated version).

[8] Guy E. Blelloch, Charles E. Leiserson, Bruce M. Maggs, C. Gregory Plaxton, Stephen J. Smith, and Marco Zagha. A comparison of sorting algorithms for the Connection Machine CM-2. In *Proceedings Symposium on Parallel Algorithms and Architectures*, pages 3–16, Hilton Head, SC, July 1991.

[9] R.P. Brent. The parallel evaluation of general arithmetic expressions. *Journal of the Association for Computing Machinery*, 21(2):201–208, 1974.

[10] D. Breslauer and Z. Galil. An optimal O(log log n) time parallel string matching algorithm. *SIAM Journal on Computing*, 19(6):1051–1058, December 1990.

[11] J. Canny. A computational approach to edge detection. *IEEE Trans. on Pattern Analysis and Machine Intelligence*, 8(6), November 1986.

[12] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. MIT Press/McGraw Hill, 1990.

[13] Allan Gottlieb, R. Grishman, Clyde P. Kruskal, Kevin P. McAuliffe, Larry Rudolph, and Marc Snir. The NYU Ultracomputer—designing a MIMD, shared-memory parallel machine. *IEEE Transactions on Computers*, C-32:175–189, 1983.

[14] W. Daniel Hillis and Guy L. Steele Jr. Data parallel algorithms. *Comm. ACM*, 29(12), December 1986.

[15] Joseph Jájá. *An Introduction to Parallel Algorithms*. Addison Wesley, 1992.

[16] James J.Little, Guy E. Blelloch, and Todd A. Cass. Algorithmic techniques for computer vision on a fine-grained parallel machine. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 11(3), March 1989.

[17] S.R. Kasaraju and A.L. Delcher. Optimal parallel evaluation of tree-structured computations by raking. In John H. Reif, editor, *Proceedings of AWOC88*, volume 319 of *Lecture Notes in Computer Science*, pages 101–110, New York, 1988. Springer-Verlag.

[18] H. T. Kung. Systolic algorithms for the CMU WARP processor. Technical report, Design Research Center, Carnegie Mellon University, 1984.

[19] Frank Thomson Leighton. *Introduction to Parallel Algorithms and Architectures: Arrays, Trees, Hypercubes*. Morgan Kaufmann, 1992.

[20] Gary L. Miller and John H. Reif. Parallel tree contraction and its application. In *Proceedings Symposium on Foundations of Computer Science*, pages 478–489, October 1985.

[21] Alan V. Oppenheim and Ronald W. Schafer. *Discrete-Time Signal Processing*. Prentice-Hall, Englewood Cliffs, NJ, 1989.

[22] Mark H. Overmars and Jan Van Leeuwen. Maintenance of configurations in the plane. *Journal of Computer and System Sciences*, 23:166–204, 1981.

[23] Theodosios Pavlidis. *Structural Pattern Recognition*. Springer-Verlag, New York, 1977.

[24] Bent E. Petersen. *Introduction to the Fourier transform and pseudo-differential operators*. Pitman, Boston, 1983.

[25] Pratt and Stockmeyer. A characterization of the power of vector machines. *Journal of Computer and System Sciences*, 12, 1976.

[26] Franco P. Preparata and Michael I. Shamos. *Computational Geometry—An Introduction*. Springer-Verlag, 1985.

[27] J. Ross Quinlan. Induction of decision trees. *Machine Learning*, 1:81–106, 1986.

[28] Abhiram G. Ranade. How to emulate shared memory. In *Proceedings Symposium on Foundations of Computer Science*, pages 185–194, 1987.

[29] John H. Reif. Optimal parallel algorithms for integer sorting and graph connectivity. Technical Report TR-08-85, Harvard University, March 1985.

[30] James B. Salem. *Render: A data parallel approach to polygon rendering. Technical Report VZ88–2, Thinking Machines Corporation, January 1988.

[31] Carla Savage and Joseph Jájá. Fast, efficient parallel algorithms for some graph problems. *SIAM Journal on Computing*, 10(4):682–691, 1981.

[32] Robert E. Tarjan and Uzi Vishkin. An efficient parallel biconnectivity algorithm. *SIAM Journal on Computing*, 14(4):862–874, 1985.

[33] L. W. Tucker, C. R. Feynman, and D. M. Fritzsche. Object recognition using the Connection Machine. *Proceedings CVPR '88*, June 1988.

[34] J.D. Ullman and M. Yannakakis. High-probability parallel transitive-closure algorithms. *SIAM Journal on Computing*, 20(1), February 1991.

[35] Uzi Vishkin. An optimal parallel algorithm for selection. *Advances in Computing Research*, 1987.

[36] Uzi Vishkin. Deterministic sampling—a new technique for fast pattern matching. *SIAM Journal on Computing*, 20(1):22–40, February 1991.

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213-3890