15-492: Parallel Algorithms **Lecturer:** Guy Blelloch
Topic: Parallel Models 1 **Date:** August 30, 2007
Scribe: Jason Knichel

2.1 Pervasive Parallel Computing: An Historic Opportunity for Innovation in Programming and Architecture

- Moore's law: `http://en.wikipedia.org/wiki/Moore%27s_law`
- Single core performance is stagnant
  - Frequency limited by leakage and power.
  - Transistor counts continue to increase
- Parallel cores in widespread and increasing use
- The full talk can be found at http://www.cs.cmu.edu/afs/cs/academic/class/15492-f07/www/cmuonly/AChienPPoPP.pdf

2.2 Agenda

4 topics for today:

- Scalable Parallelism
- Efficiently Scalable Parallelism
- Work
- Work Efficient

2.2.1 Scalable Parallelism

- You want a parallel algorithm to be scalable
  - As you get more processors, performance improves
- At some point, adding more processors cannot possibly help increase performance for a given n
- Scalable parallelism means that as n gets larger, the maximum number of processors you can effectively use gets larger

$T_p(n,p)$: amount of parallel time on n elements and p processors

$T_s(n)$: amount of sequential time

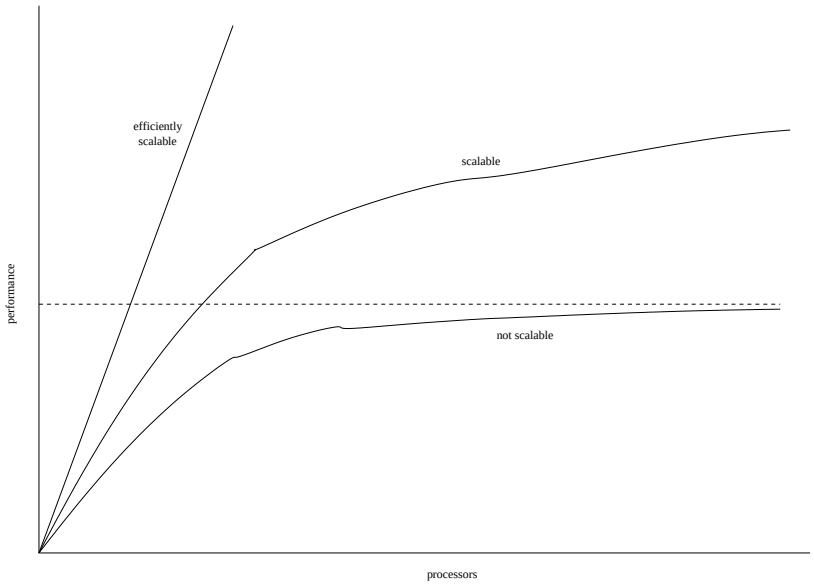
Scalable Parallelism :

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

2.2.2 Efficiently Scalable

An algorithm is *efficiently scalable* if

$$\lim_{n\to\infty}\frac{T_p(n,p)}{T_s(n)}=\frac{1}{p}.$$



Example: Example of a not scalable algorithm: Consider a vision algorithm with 4 parts where each step cannot be done until the steps before it are completed:

- line recognition
- feature detection (find corners, etc)
- grouping (find objects)
- matching (tries to match object to some entity like a wrench, car, bed, etc)

One thought of how to make this a parallel algorithm is to pipeline the 4 steps and have each step run in a different processor. However, this does not result in a scalable algorithm. The most amount of processors you can use is 4. Another thought is that if you have p processors, devote

$p/4$ processors to each step and process many frames in parallel. However, if one step takes longer than the other steps, this step becomes a bottleneck, and the algorithm still is not scalable. ■

See Amdahl's Law (http://en.wikipedia.org/wiki/Amdahl's_law) for information about the maximum amount of speedup you can get for an algorithm by only speeding up a portion of the algorithm.

2.2.3 Work

W = T * P

- T = time
- P = number of processors

If the number of processors varies over time, then the work is the integral of the number of processors vs time graph.

2.2.4 Work Efficient

$W_p(n, p) = O(T_s(n))$

Examples of not work efficient algorithms:
Example: Circuit Simulation

Divide the circuit into segments. Each segment is simulated by a different processor. Assume each segment has the same number of gates on it so each segment should take about the same time to simulate. Every step, simulate each segment.

This parallel algorithm was found to be slower than the best known sequential algorithm. In a real circuit, not many gates switch each step. The sequential algorithm takes this into account and looks at outputs that change and only run areas of the circuit that are affected by those output changes. The parallel algorithm does not do this, and instead runs all the segments every step. This results in the parallel algorithm doing a lot more work than the sequential algorithm. ■

Example: Rendering

One common way of parallelizing rendering was to let each pixel be controlled by a different processor. You would then go through all the objects that need to be rendered and broadcast their location to all the processors so the processor can decide if the pixel it controls needs to be changed. However, for any given object, a lot of the pixels won't be anywhere close to it but will still have to process the object anyway to decide if the pixel should be updated. This results in a lot of extra work being done to check if a pixel intersects an object.

One way to improve this algorithm is to divide the screen up into squares and only broadcast the location of an object to the pixels in the square the object is contained in. ■

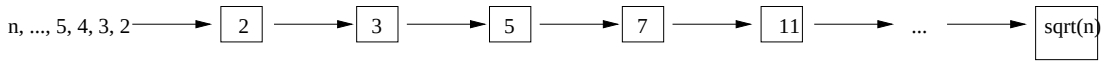
Example: Primes up to n

Pseudo-code for a common sequential algorithm:

```
for i = 0 to n
  primes[i] = 1
for i = 2 to sqrt(n)
  if primes[i] = 1 then
    for j = 2*i to n by i do
      primes[j] = 0
```

This is $O(n \log \log n)$.

Parallel Alg 1: Stream Sieve



For this algorithm, you set the processors up in a stream, with them all originally as "unset". You feed the numbers into the beginning of the stream, one at a time (starting from 2), from low to high. If a processor is given a number and that processor is "unset" then it becomes "set" and takes the value of the number it was given. If a processor receives a number and the processor is already "set" then it checks if the received number is a multiple of the value of the processor. If the number is a multiple of the value, then that number gets discarded. If it is not a multiple, then it gets passed through the stream to the next processor in line.

You do not need any processors to have a value above $\sqrt{n}$. This is because if a number is between $\sqrt{n}$ and n, and it is not prime, one of its factors must be less than $\sqrt{n}$.

Any number that makes it out of the stream is a prime number and any number which is a value of one of the processors is a prime number.

There is a factor of about $\log(n)$ between two consecutive primes. Thus, for this algorithm, you would need approximately

$P = \frac{\sqrt{n}}{\log(n)}$ processors

$T = n$ time

and thus $W = \frac{n^{\frac{3}{2}}}{\log(n)}$

so this algorithm is not work efficient.

Parallel Alg 2:

Each processor has the value of one of the numbers between 2 and n and each processor has 3 states: unprocessed, processed prime, and processed unprime.

Steps for algorithm:

1 Find the unprocessed processor with the smallest value

2 Broadcast that value to the rest of the processors and mark yourself processed prime

3 The rest of the processors receive that value and mark themselves as processed unprime if they are a multiple of that value

4 Repeat until there are no unprocessed processors left

$P = n$ processors

$T = \sqrt{n}$ time

and thus $W = n^{3/2}$.

so again this algorithm is not work efficient ■