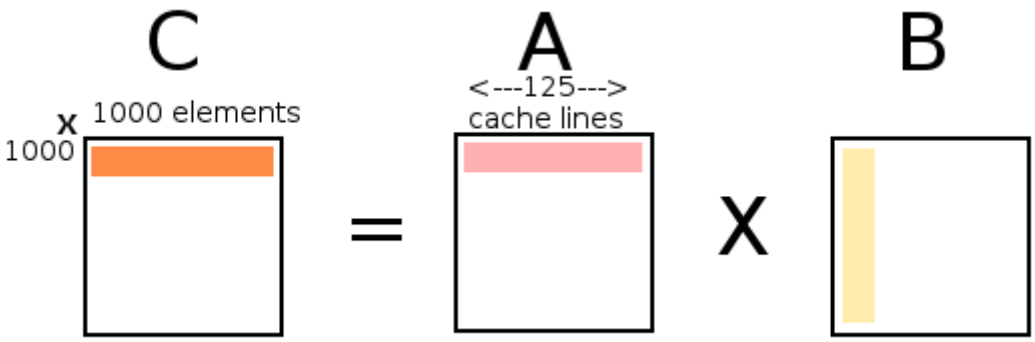
Notes on CS61 Lecture on 11/20/2012
Charles Cabot, Tommy Heffernan, Iva Milo

Lecture Title: Concurrency

1. Matrix example: Analyzing processor cache loads
2. Network Programming

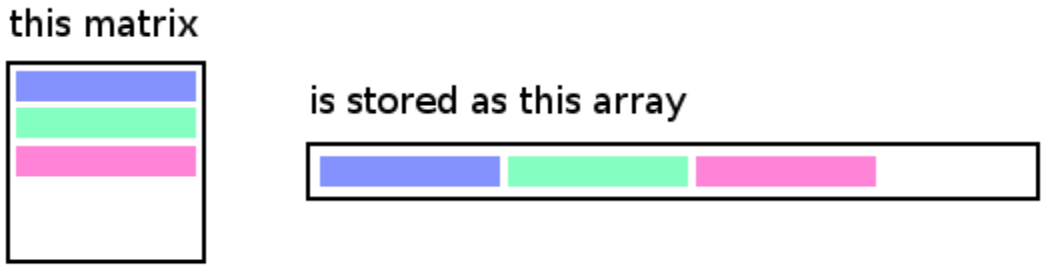
Matrix Multiplication
C = A X B



(diagram: elem in C is equal to dot product of row of A and column of B.)

Each cache line can store 64 bytes. Assuming this is a matrix of doubles, that means we can fit 8 elements per cache line.

Map elements of A in *Row Major Order*. Linear representation of a matrix which contains each row in order.



m[i,j] ≡ m[i*sz+j]

for (i = 0; i < sz; ++i)

```
for (j = 0; j <sz, ++j)
    c[i,j] = 0;
    for (k = 0; k < sz; ++k)
        c[i,j] += a[i,k] * b[k,j]
```

Many ways to multiply matrices. Lots of them come from reordering indices. “i”, “j”, “k” can be outer loop, middle loop, inner loop. Different choices for this will have different speeds because of Row Major representation.

Assuming we start with cold cache, this misses 1.125 per iteration. b access misses every time, a access misses every 8 times. c misses once every 8000 times (negligible).

Cache will often be warm though. For example, multiply 8x8 matrices, 24 total cache lines. For matrix size 1000 though, this number is enormous. Greater size of working set means access pattern is less local, which increases the chance of misses.

Rearrange to be j,i,k. This doesn’t change much. c was missing 1/8000 before, now 1/1000. However, a and b still miss at the same rate.

Now consider k,i,j. This removes the vertical access of matrix B. However, this leads to a strange pattern. Each iteration of the inner loop updates an element of C, it does not set it to its final value. Basically, the first inner loop adds the product of a00 and b0n to c0n for each n. The next inner loop adds the product of a10 and bn0 to c1n for each n. Once that loop has completed n times, the remaining index is incremented and the process starts again. The general rule is that for each entry in a, the loop adds aXY times bYn to cXn.

Read more about this at http://www.scribd.com/doc/91794393/26/Matrix-Multiplication-kij.

Network Programming

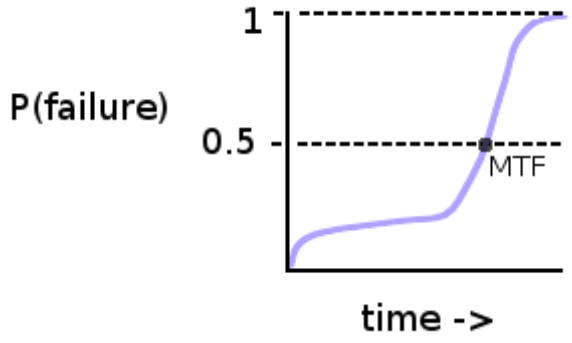
Reasons for excitement:

- Gain access to and use resources of many computers at once
- Faster

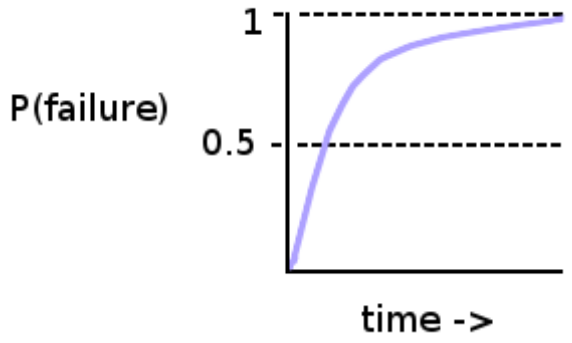
Problems:

- Security
- Tendency to fail
    - This can be measured with the metric “mean time to failure”: amount of time a component takes to fail with probability 0.5

Single component:



Closely-coupled, 100 component system:



Leslie Lamport: “A distributed system is one in which the failure of a computer you’ve never heard of prevents you from getting your work done.”

Tendency to fail: The probability that a system fails by time T is equivalent to:

1 - (1 - probability one component fails by time T)^(number of components)

(This assumes that all components have an equal probability of failing.) This drastically increases the rate of failure for large networks.

A Tempting Analogy: Remote Procedure Call
getservbyname() is slow, it reads a large table of into memory every time.

IDEA: To speed this up, instead of having per process cache, store the cache (/etc/services) in one process’s memory on one machine. To lookup, contact that machine.

getservbyname() from hard disk latency: 1/100sec
getservbyname() via distributed system latency: request + lookup + response

- requests and responses are fast, but not at the speed of light. longer the more geographic distance it has to travel.
- much less reliable - the other component could be down.

(message sequence chart diagram, lines represent machines, time moves downwards, diagonal lines are messages)

This looks like procedure calls, request is call, response is port number