CSSE335 Notes

Andy Miller
CM 1196

May 20, 2015

Contents

March 27, 2015

1 Qualitative Measurements Speed

Assume we have a task of size that needs to be completed. Let t_s be the time required for the best serial algorithm to complete our task. Let t_p be the time required for p processors to complete our task.

Definition 1.1. We define **speedup** as

$$s_p = \frac{t_s}{t_p} \tag{1.1}$$

Ideally, $t_p = \frac{t_s}{p}$ (all work is perfectly divided), but this is almost never the case. Sometimes we cannot divide work evenly (**load balancing**), and we don't need to account for "message" time in serial. Finally, the best serial algorithms are often dramatically different from the best parallel algorithms (different approaches). Still, we have this bound on our speedup

$$s_p = \frac{t_s}{t_p} \leq \frac{t_s}{\frac{t_s}{p}} \leq p. \tag{1.2}$$

This means we have at most a p times speedup.

Definition 1.2. We measure **efficiency** as

$$E_p = \frac{s_p}{p} = \frac{\frac{t_s}{t_p}}{p} = \frac{t_s}{p \cdot t_p}. \tag{1.3}$$

Assume that there is a certain fraction of our work that is inherently serial, f. Then, the amount of time spend on this work is assumed to be ft_s. Then, the time to complete in parallel is

$$t_p = ft_s + \frac{(1-f)t_s}{p}. \tag{1.4}$$

This is the best case, where $\frac{(1-f)t_s}{p}$ is a perfectly parallel-ized computataion of the rest of the work. The speedup of this work is

$$s_p = \frac{t_s}{ft_s + \frac{(1-f)t_s}{p}} = \frac{pt_s}{pft_s + (1-f)t_s} = \frac{p}{pf + (1-f)} = \frac{p}{1 + f(p-1)} \tag{1.5}$$

This final result is known as **Amdahl's Law**:

$$s_p = \frac{p}{1 + f(p-1)} \tag{1.6}$$

Ideally, we would find the maximum speedup by taking the limit of processors.

$$\lim_{p \to \infty} s_p = \lim_{p \to \infty} \frac{p}{1 + f(p-1)} = \frac{1}{f}. \tag{1.7}$$

This means that when an algorithm is inherently 5% serial, it is impossible to make it more than 20 times faster with parallel computation.

Amdahl assumes that t_s is fixed, and that we try to decrease t_p with p. Assume instead t_p is fixed, and t_s increases with p. Then,

$$t_s = ft_s + (1-f)t_s \tag{1.8}$$

$$t_p = ft_s + \frac{(1-f)t_s}{t_p} \tag{1.9}$$

$$\vdots \tag{1.10}$$

$$s_p = \frac{t_s}{t_p} \tag{1.11}$$

$$\vdots \tag{1.12}$$

$$s_p = p + (1-p)ft_s \tag{1.13}$$

This was stated by **Gustafson**: if we add more processes, then do more work. This causes speedup to be linear in p.

One thing these models haven't considered is **communication time**. We can break parallel processing time into

$$t_p = t_{comp} + t_{comm} = \frac{t_s}{p} + t_{comm} \tag{1.14}$$

Then,

$$s_p = \frac{t_s}{\frac{t_s}{p} + t_{comm}} = \frac{pt_s}{t_s + pt_{comm}} \tag{1.15}$$

If t_{comm} is not insignificant, then we lose a large amount of speedup. In the real world, communication time will dominate the time consumed by our algorithms.

2 Blocking

March 30, 2015 Our "blocking" send isn't that "blocking" anymore. Now there's a "buffer" that is used. We write what we send to the buffer, and to receive a process goes to the buffer.

Definition 2.1. A **locally blocking send** will block *until* we have finished writing into the buffer. We regain control **when we can write to the buffer again**.

Definition 2.2. A **locally blocking receive** will block until we finish reading data from the buffer. This functions essentially the same as a regular blocking receive.

Definition 2.3. A **synchronous send and receive** fully block the sending process until the receiving process has finished receiving the data.

3 Time Estimation

$$t_p = t_{calc} + t_{comm} \tag{3.1}$$

The calculation time t_{calc} is fairly straightforward, we will count the number of additions/multiplications/etc.

The time to complete a single communication message is modeled as

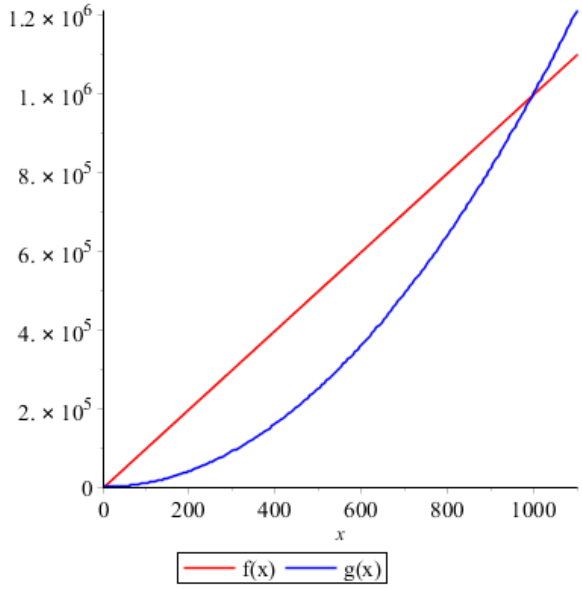
$$t_{comm} = t_{startup} + w \cdot t_{data} \tag{3.2}$$

where $t_{startup}$ is a constant startup cost, w is the amount of data, and t_{data} is the data transfer rate. Unfortunately, the most significant cost is the **startup cost**

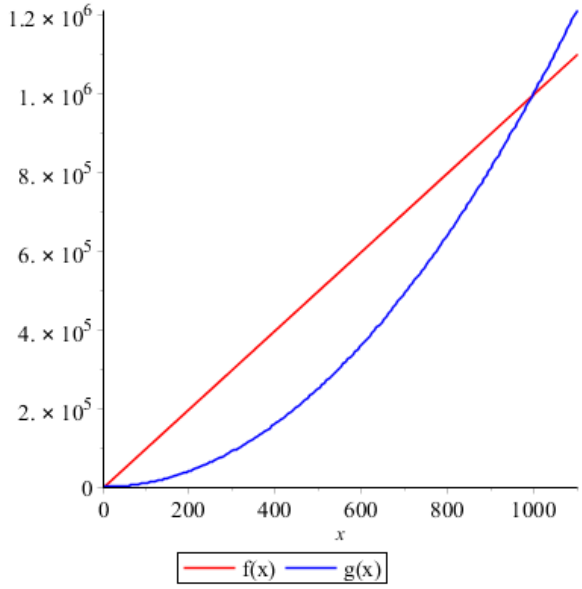
3.1 Asymptotic Notation

Definition 3.1. $f(x) = O(g(x))$ (**Big-Oh**) if there exists a constant c such that $0 \leq f(x) \leq cg(x)$ for all "large" x.

Example 3.1. $f(x) = 1000x$, $g(x) = x^2$. $f(0) = g(0)$, $f(1) > g(1)$, $f(2) > g(2)$. However, $f(1000) = g(1000)$ and for all $x > 1000$, $f(x) < g(x)$. So, $f(x) = O(g(x))$.



Example 3.2. $f(x) = 1000x$, $g(x) = x$. $f(x) > g(x)$ always, but $f(x) < 1500g(x)$. So, by the definition, $f(x) = O(g(x))$ (with $c = 1500$).



Remark. Big-Oh notation refers to how *fast* a function grows as x becomes large.

Note that $x = O(x^2)$ but $x^2 \neq O(x)$. This is because x^2 grows faster than all cx. We have an alternate definition for "grows slower."

Definition 3.2. $f(x) = \Omega(g(x))$ (**Big-Omega**) if there exists a constant c such that $f(x) \geq cg(x)$ for all "large" x.

Notice that $x = O(x)$, but $x = O(x^2)$ or even $x = O(x^{1000})$. These last two aren't very descriptive, but still satisfy the definition of $O(g(x))$. We have a "tighter" description.

Definition 3.3. $f(x) = \Theta(g(x))$ (**Big-Theta**) if there exists $c_1, c_2 > 0$ such that $0 \leq c_1 g(x) \leq f(x) \leq c_2 g(x)$ for all "large" x. Equivalently, $f(x) = O(g(x))$ and $f(x) = \Omega(g(x))$.

We want to strive to use Big-Theta as much as possible.

3.2 Time Analysis with Processors

Usually we aren't worried about how time grows with the size of the data so much as how it grows with the numbers of processors. This immediately brings up one deceptive thing about asymptotic notation: *constants are ignored*. Parallel computation is mostly dependant on the large constant overhead. This analysis does not consider that.

Example 3.3. Suppose we have n data, all beginning on processor p_0. We have P processors, and we want $\sum$ data.

Definition 3.4. A problem is called **embarassingly parallel** if it naturally decomposes into independent parts that may be computed in parallel with no (or little) communication between processors.

We have discussed **MPI_Gather**, **MPI_Scatter** and **MPI_AllGather**.

Example 3.4. Julia Set. Let z be a complex number and let c be another complex number. Define

$$z_0 = z \tag{3.3}$$
$$z_{k+1} = z_k^2 + c \tag{3.4}$$

z is in the c-Julia set if $|z_k|$ is bounded by all k.

The simplest test to see if z is in the c-Julia set is to set an upper bound M and iterations n. If $z_k < M$ for all $k < n$, then we assume z is in the set.

We will use an **escape time criteria** to "determine" if something is in the Julia set. Let z, c be given. Calculate $z_1, \ldots, z_{maxiter}$. If any $|z_i| > M$, then $z \notin J_c$. Otherwise, we assume it is.

If we use **static allocation**, we give each process say 10000 points each. This could be bad because the first 10,000 (processor 1) could finish much more quickly than the second 10,000 (processor 2). Instead, we can use **dynamic allocation**, where processors

1. Produce "julia" executable.
2. With infinity options. Mostly done though! atoi optargs
3. Make complex.h, complex number routines

   Complex struct: add, multiply, norm
4. Forcebyteswap=1 on cluster

4 Work Allocation

Definition 4.1. Static Work Allocation is where all tasks are assigned to processors at the very start of program execution.

Definition 4.2. Dynamic Work Allocation is where a small amount of work is assigned (statically) to each processor at the start, and there is more work left to do. When a processor completes its assigned work, more tasks are assigned to it. This is usually referred to as having a **pool** or processors.

Static allocation gives a well balanced number of tasks for each process. However, if different tasks are harder to complete (such as with the Julia set), then this does not properly allocate the amount of *work* (and thus time). Also, static allocation does not take into account the speed of an individual processor (if one is twice as fast, it will finish sooner).

Dynamic allocation is well balanced for time required by each processor (since they all end at roughly the same time). However, dynamic allocation requires far more communication or overhead.

Example 4.1. Goal: given N, produce the array P where $P[i] = 1$ if i is prime, and 0 otherwise.

Our primality test: to see if n is prime, try to divide by $2, 3, \ldots, n-1$. There are far, far better primality tests, but this is very simple to write and will take slightly longer for smaller integers.

Static Allocation:

1. Master does no work
2. If there are w processes, worker i will check primality of $k = (i-1)\frac{N}{w}$ to $k = i\frac{N}{w} - 1$.

5 Partitioning and Divide and Conquer

There are two main ways to divide up work

- Data Division (Domain Decomposition, Divide and Conquer)
  - Each processor executing the same instructions on a partition of the data
  - Example: Each process sums n numbers.
- Instruction Division (Pipelining)
  - Each processor executes a few instructions on all of the data
  - Example: The same idea as a processor pipeline, (Fetch, Decode, Execute, Write-back). Each processor could be a different stage of the pipeline).

5.1 Bucket Sort

Definition 5.1. (**Bucket Sort**) Given n numbers to sort, and m buckets with cutoffs a_1, b_1, $a_2, b_2, \ldots, a_m, b_m$. Bucket sort proceeds as follows.

1. Partition the data into buckets by putting data d_i into bucket j if $a_j \leq d_i \leq b_j$.
2. Sort each bucket individually.
3. There should be (little to) no work required for reassembly.

Example 5.1. Let $d = [-5, 3, 11, 6, -2, 3, 1, 9, -11, 7]$. We will use 4 buckets: $[-20, -4), [-4, 6), [6, 7), [7, 23$ This assignment covers all of the data, and each data point will only be in exactly one bucket.

1. Partition the data. The buckets are as follows
   (a) $[-5, -11]$
   (b) $[3, -2, -3, 1]$
   (c) $[6]$
   (d) $[11, 9, 7]$
2. Sort each bucket individually. Now the buckets rae
   (a) $[-11, -5]$
   (b) $[-3, -2, 1, 3]$
   (c) $[6]$
   (d) $[7, 9, 11]$
3. We concatenate the buckets to form a finished list
   (a) $[-11, -5, -3, -2, 1, 3, 6, 7, 9, 11]$

Time Complexity For serial bucket sort.

1. Partitioning. $t_{partition} = \Theta(n \cdot \log m)$. For each d_i, we look up the correct bucket. If the buckets are sorted, then it is a $\log m$ search.

2. Sorting Each Bucket. A sequential sort in a given bucket depends on the number of elements in the bucket. If we partition perfectly, each bucket will have exactly $\frac{n}{m}$ elements. Thus, the time for the sort is $t_{bucket} = \Theta(\frac{n}{m} \log(\frac{n}{m}))$. However, we must do this for each of the m buckets, so $t_{sort} = \Theta(n \log(\frac{n}{m}))$.

3. The total time is thus $t_s = t_{partition} + t_{sort} = \Theta(n \log m) + \Theta(n \log \frac{n}{m}) = \Theta(n \log \frac{n}{m})$ since we assume $\frac{n}{m} > n$.

If we have an effective (good distribution) and efficient (quick) way to partition the data, then this is very effective. However, we cannot pre-process the data to determine good buckets to use (if we spent time on that, we should've just used quicksort instead). Our goal is to get roughly $\frac{n}{m}$ data per bucket.

For now, we will assume our data is uniformly distributed in $[0, 1]$. So, for m buckets, we should make the cutoff for bucket j be $\frac{1}{m}(j-1)$ to $\frac{1}{m}j$.

Parallel Implementation In parallel, we assign each worker 1 bucket to sort. We are assuming the data is initially on the master, we have P processors, 1 master, and m workers. We also want to collect all data on the master again.

1. Give each worker *all* of the data.

2. Each processor partitions its portion of data into "little buckets"

3. Each processor sends its little bucket i to processor i (so processor i has everything in bucket i)

4. Each processor sorts its bucket in serial

5. Each processor sends its bucket information back to the master (so the master has the fully sorted list

Definition 5.2. A **broadcast** the sending of the exact same data from one process to all the others (e.g. the master sends all of the data to each process). This is implemented as `MPI_Bcast`

Definition 5.3. A **gather** is receiving data on one process from all the other processes (e.g. the master receives a result from each other process). This is implemented as `MPI_Gather`.

Time Complexity of Parallel Bucket Sort

1. The broadcast: $t_1 = t_{startup} + |d| t_{data}$

2. Partition, takes $t_2 = \frac{|d|}{m}$

3. Correct buckets: $t_3 = m(m)[t_{startup} + t_{data}\frac{d}{m^2}] = m^2 t_{startup} + |d| t_{data}$

4. Sort buckets: $t_4 = \frac{|d|}{m}\log(\frac{|d|}{m})$
5. Gather: $t_5 = t_{startup} + \frac{|d|m}{t}_{data}$

We need to add these all together

$$t_m = t_1 + t_2 + t_3 + t_4 + t_5 \tag{5.1}$$

$$= t_{startup} + |d|t_{data} + \frac{d}{m} + m^2 t_{startup} + d + \frac{d}{m}\log(\frac{d}{m}) + mt_{startup} + dt_{data} \tag{5.2}$$

$$= \Theta(\frac{d\log(\frac{d}{m})}{m}) \tag{5.3}$$

This is what we wanted, the serial efficiency $\Theta(d\log(\frac{d}{m}))$ but divided by the number of processors. If we have a good amount of processors relative to the data (we will — 100 processors vs 1 billion elements), then this scales very well.

6 Numeric Integration

$$\int_0^1 \mathrm{e}^{x^2}\,dx \tag{6.1}$$

There is not an antiderivative for e^{x^2}, so we have to find another way to integrate this. We use Riemann sums. Specifically, we will use the trapezoidal rule.

$$\int_a^b f(x)\,dx \approx \delta(\frac{f(x_0)+f(x_1)}{2}) + \frac{\delta}{2}(f(x_1)+f(x_2)) + \frac{\delta}{2}(f(x_2)+f(x_3)) + \cdots + \frac{\delta}{2}(f(x_{n-1})+f(x_n)) \tag{6.2}$$

$$\approx \delta(\frac{f(a)}{2} + f(x_1) + f(x_2) + \cdots + \frac{f(b)}{2}) \tag{6.3}$$

6.1 Adaptive Quadrature

Let $T(f, a, b, N)$ represent applying the trapezoidal rule from a to b of f with N nodes.

$$T(f, a, b, N) = \delta(\frac{f(a)}{2} + f(x_1) + f(x_2) + \cdots + f(x_n) + \frac{f(b)}{2}) \tag{6.4}$$

If the problem is worth solving, then f is *ridiculously* difficult to solve. So, we actually want to minimize the number of function evaluations we have to do. So, our basic idea is: if

$$|T(f, a, b, N) - T(f, a, b, 2N)| < \varepsilon \tag{6.5}$$

then $T(f, a, b, 2N)$ is accepted, otherwise calculate $T(f, a, b, 4N)$ and repeat. We call this **adaptive quadrature**. Some pseudocode:

```
AQ(f, a, b, eps, N):
  approx1 = T(f, a, b, N)
  approx2 = T(f, a, b, 2*N)
  if(abs(approx1 - approx2) < eps)
    return approx2
  else
    return AQ(f, a, b, eps, 2*N)
```

For some functions, we may be approximating the one half of the function far better than the other half (imagine a function with exponential decay, the beginning has wider variations). So, instead of simply doubling the nodes, we will integrate the "left half" and the "right half." This is truly adaptive because we don't need to get the entire integral in a single iteration (and is easier to parallelize).

```
AQ(f, a, b, eps, N):
  approx1 = T(f, a, b, N)
  approx2 = T(f, a, b, 2*N)
  if(abs(approx1 - approx2) < eps)
    return approx2
  else
    L = AQ(f, a, (a+b)/2, eps, N)
    R = AQ(f, (a+b)/2, b, eps, N)
    return L + R
```

This also frees up the processors to do more things.

We ideally want to use dynamic allocation to solve this problem. However, this is more difficult than the Julia set problem. With the Julia set, we knew how many tasks their were in total. In numeric integration, AQ may generate more tasks, so we don't know how many tasks there will be at the start. Distribution of tasks is more difficult.

7 N-Body Problem

Given N stars and initial positions, velocities, and mass, what do they end up after some time?

Let $\mathbf{x}_i(t)$ be the position of body B_i, and $\mathbf{v}_i(t)$ be the velocity of B_i at time t. Let m_i be the mass of B_i, assumed unchanging. Let $\mathbf{F}_i$ be the net force acting on B_i. Then

$$F_i = m_i a_i = m_i v_i' \tag{7.1}$$

In addition, using the law of gravitation

$$F_1 = g\frac{m_1 m_2}{d_{12}^2} + g\frac{m_1 m_3}{d_{13}^2} + \dots \tag{7.2}$$

In general, an equation is (writing $d_{ij}^2 = ||x_i - x_j||^2$.

$$F_i = g m_i \sum_{j \neq i} \frac{m_j}{||x_i - x_j||^2} \tag{7.3}$$

And so,

$$m_i \mathbf{v}_i' = G m_i \sum_{j \neq i} \frac{m_j}{||\mathbf{x}_i - \mathbf{x}_j||^2} \cdot \frac{\mathbf{x}_j - \mathbf{x}_i}{||\mathbf{x}_i - \mathbf{x}_j||} \tag{7.4}$$

$$\mathbf{v}_i' = G \sum_{j \neq i} \frac{m_j}{||\mathbf{x}_i - \mathbf{x}_j||^2} \cdot \frac{\mathbf{x}_j - \mathbf{x}_i}{||\mathbf{x}_i - \mathbf{x}_j||} \tag{7.5}$$

$$\mathbf{x}_i' = \mathbf{v}_i \tag{7.6}$$

The last two equations are our system of differential equations.

7.1 Euler's Method

Based on Taylor Series

$$x(t+h) \approx x(t) + h \cdot x'(t) \tag{7.7}$$

This is what we will use to approximate the solution of the n-body problem.

Then we talked about it and I didn't pay attention oops. I was doing other homework, that's right. It was a long day.

8 Pipelining

April 27, 2015 Given $d_1, \ldots, d_n$ and $x_1, \ldots x_n$. Compute $x_1 + d_1 + d_2 + \ldots d_n$, $x_2 + d_1 + d_2 + \ldots$, and so on. Suppose that for some reason we can't just precompute the $\sum d_i$. I don't know why, but those are the rules. In serial, we may use the following solution

```
for i = 1 to n
  compute xi + d1 + d2 + ... + dn
end
```

Specifically, there's this inherently serial "accumulator," but we want to do it several times.

```
acc = xi
for j = 1 to m
  acc += dj
end
```

We could break up parts of the function for each process, instead of distributing the function to each process.

Imagine worker W_j has the job "add d_j to input."

```
global dj

def compute(input):
  return input + dj

def worker_job:
  input = MPI_receive()
  output = compute(input)
  MPI_send(output)
```

This puts the job into a **pipeline**. After n steps, the pipeline will be full and every processor will be doing something.

To alter this slightly, we are not working with a hardware pipeline; every processor can do the same thing. Why not have processor 2 begin its input with x_2 instead of waiting for $x_1 + d_1$ to start doing anything? This can fill up the pipeline much more quickly.

Pipelining is a strategy for breaking up a serial process if we have to do the same process multiple times. The pseudocode for a workers job is above. It is fairly simple.

8.1 Complexity Analysis

We mainly analyze "pipeline cycles," which is one transition for a single processor. In our examples, we need $m+n-1$ cycles to complete this. There are m cycles for each processor (for each data), and there is an $n-1$ "delay" before the last cycle starts. Then, the work of a cycle is $2(t_{startup}+t_{data})+1$. Thus, the total time is

$$t_n = (2(t_{startup}+t_{data})+1)(m+n-1) \tag{8.1}$$

For a large number of jobs $m \gg n$, our cycle efficiency is

$$t_a = \frac{t_n}{m} \approx 2(t_{startup}+t_{data})+1 \tag{8.2}$$

This means we are getting roughly 1 full job's worth of work done every cycle.

8.2 Sorting Again

Suppose we can't partition our data, so we can't use bucket sort. So, let's do some kind of parallel insertion sort.

```
for i = 1..n
  for j = 1..length(sorted list)
    is xi the jth largest element?
      if yes, insert xi into sorted list
    else
      loop
```

We can have each processor determine if it has the jth largest data. When it receives data, it checks if x_j is the new jth largest element or not.

```
Recv(data, i-1)
if data < mymax
  send(data, i+1)
else
  send(mymax, i+1)
  mymax = data
```

I'm sure it works somehow. It totally does. It doesn't need to know which the first $i-1$ largest elements are, because workers $1..i-1$ will keep track of that for us. We just need the max, and that is auto-magically the right work. Hooray!

April 28, 2015

9 Sieve of Eratosthenes

I know what this is. Mark every multiple of p as not prime. This "generates" all the primes up to n.

Do it in parallel. Pipeline, send some numbers. Use a processor as a filter.

10 Solving Linear Systems

$$A\mathbf{x} = \mathbf{b} \tag{10.1}$$

This problem is the heart of all modern scientific computing.

Apparently we talked about this or something? But I have no notes. I do know we are doing Gaussian Elimination though, what a boss.

10.1 Broadcast Method

Broadcast row i to processors $i+1$ to n. Each processor subtracts the correct multiple of R_i from its local row.

Complexity

- We have n elimination stages.
- At stage i, we broadcast $n-1$ data to $n-i$ processors
- There are $n-i$ calculations done to eliminate the rest of the row.

The result is

$$\sum_{i=1}^{n-1} \Theta((n-i)\log(n-i)) + \Theta(n-i) = \Theta(\sum_{i=1}^{n-i}(n-i)\log(n-i) + (n-i)) \tag{10.2}$$

$$= \Theta(\sum_{i=1}^{n-i}(n-i)(\log(n-i)+1)) \tag{10.3}$$

$$= \Theta(\sum_{i=1}^{n-i} n^2 \log(n)) \tag{10.4}$$

Ta-da

Pipelining Pipeline this yo. Just like, have each processor slowly forward everything along. It will be wonderful. If it has useful anyformation, send it ahoy.

Each processor forwards any useful rows to next processor, and performs any possible eliminations. The latency of a processor is $\Theta((n-1)n) = \Theta(n)$. After a little bit of magic, we find that this is *actually* $\Theta(n^2)$ factoring in the necessary work.

Apparently this can also pivot. That sounds like a mess.

10.2 Back-Substitution

Also pretty annoying. Like, we pretty much just divide, subtract the old answers. We probably need to broadcast more. Find x_n, then broadcast it. Use it to find x_{n-1}. Repeat. In general, the formula for x_i is

$$x_i = \frac{b_i - \sum_{j=i+1}^{n} a_{ij}x_j}{a_{ii}} \tag{10.5}$$

This is a $\Theta(n^2)$ algorithm in serial. That means it is the *same* as our parallel Gaussian Elimination. So, it isn't *that* big of a deal if we don't parallelize it.

But if we want to, we'll run the same pipeline in reverse.

11 More Linear Algebra

11.1 Matrix Multiplication

$$(AB)_{ij} = A_{i:} \cdot B_{:j} \tag{11.1}$$

Where $A_{i:}$ is the ith row of A and $B_{:j}$ is the jth column of B.

Then we did an example of multiplication. I've taken DE2.

In general, $(AB)_{ij} = \sum_{k=1}^{n} A_{ik}B_{kj}$. We can easily write this in serial.

```
for i = 1 to n:
 for j = 1 to n:
  sum = 0
  for k = 1 to n:
   sum += A[i][k] + B[k][j]
  Ans[i][j] = sum
```

This is an $O(n^3)$ algorithm.

Parallel Algorithm Most basic: we have n^2 workers, one for each entry. We can't just send all the information with regular sends and receives. That's a ton. Instead, we can have the master broadcast information. Each processor gets some extra information, but instead of $O(n^2)$ sends, we have $O(n^2 \log(n^2)) = O(n^2 \log n)$. Then, at the end, the master can do a gather for a final $O(\log(n^2)) = O(\log(n))$ step.

Other: We could assign each processor a row, and have it compute that row of multiplications.

11.2 Block Matrices

Suppose that $A_{11} = \left(\begin{smallmatrix} 1 & 3 \\ 4 & 1 \end{smallmatrix}\right)$, $A_{12} = \left(\begin{smallmatrix} 4 & 6 \\ 3 & 0 \end{smallmatrix}\right)$, $A_{21} = \left(\begin{smallmatrix} 3 & 0 \\ 1 & 9 \end{smallmatrix}\right)$, and $A_{22} = \left(\begin{smallmatrix} 7 & 7 \\ 7 & 7 \end{smallmatrix}\right)$. We can write a full matrix A as

$$A = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} = \begin{bmatrix} 1 & 3 & 4 & 6 \\ 4 & 1 & 3 & 0 \\ 3 & 0 & 7 & 7 \\ 1 & 9 & 7 & 7 \end{bmatrix} \tag{11.2}$$

Note that block matrices do not need to be square, or even the same size. Kind of magically (which I totally kind of proved in Linear Algebra?) is that if A and B are block matrices, then $C = AB$, then

$$C = \begin{bmatrix} A_{11} & A_{12} & \dots & A_{n1} \\ A_{21} & \ddots & & \vdots \\ \vdots & & & \\ A_{n1} & \dots & & A_{nn} \end{bmatrix} \begin{bmatrix} B_{11} & B_{12} & \dots & B_{n1} \\ B_{21} & \ddots & & \vdots \\ \vdots & & & \\ B_{n1} & \dots & & B_{nn} \end{bmatrix} = \begin{bmatrix} C_{11} & C_{12} & \dots & C_{n1} \\ C_{21} & \ddots & & \vdots \\ \vdots & & & \\ C_{n1} & \dots & & C_{nn} \end{bmatrix} \tag{11.3}$$

Where, if the A matrices and B matrices are compatible to multiply, then the *matrix* $C_{ij} = \sum_{k=1}^{n} A_{ik}B_{kj}$. This is perfectly analogous to regular matrix multiplication, but A and B are matrix multiplications instead of regular multiplications.

If we want to find the product of A and B, we can divide the matrices into 4 sub-block matrices. Then, we can do 4 separate, smaller matrix multiplications to solve this problem.

$$C = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix} = \begin{bmatrix} A_{11}B_{11} + A_{12}B_{21} & A_{11}B_{12} + A_{12}B_{22} \\ A_{21}B_{11} + A_{22}B_{12} & A_{21}B_{12} + A_{22}B_{22} \end{bmatrix} \tag{11.4}$$

Each of these smaller matrices is $\frac{n}{2} \times \frac{n}{2}$ if A is $n \times n$. We can use this for a recursive strategy for matrix multiplication.

```
mult(A, B):
 if A and B are n-by-1:
  return AB
 else
  define the submatrices A11, B11, A12, etc.
  M1111 = mult(A11, B11)
  M1221 = mult(A12, B21)
  ...
  # After all of those
  C11 = M1111 + M1221
  C12 = M1112 + M1222
  ...
  return [C11 C12; C21 C22]
```

Parallel Algorithm We can use a workpool style algorithm. We have children compute M_{1111}, M_{1221}, etc. If we wanted (and we probably do), we can have child processes request more children, and continue the "recursion" all the way down to 1 by 1 matrices.

This creates a tree of processors that is $\log n$ deep. The first level of communication, if we only send info that is required to a process, then the first level requires $\frac{n^2}{4}$ sends and receives. The second level $\frac{n^2}{16}$, then $\frac{n^2}{64}$, and so on, until the final row is $\frac{n^2}{n^2} = 1$, which is the base case (a single multiply, sending single values). Then at each level we have additions. They require $\frac{n^2}{2^k}$ additions as well. The total time needed is

$$T = \frac{n^2}{4} + \frac{n^2}{16} + \frac{n^2}{64} + \cdots + \frac{n^2}{4^{\log_2 n}} \tag{11.5}$$

$$= n^2(\frac{1}{4} + \frac{1}{16} + \cdots + \frac{1}{4^{\log_2 n}} \tag{11.6}$$

$$= n^2(\frac{1}{4} + \frac{1}{16} + \cdots + \frac{1}{(2 \cdot 2)^{\log_2 n}} \tag{11.7}$$

$$= n^2(\frac{1}{4} + \frac{1}{16} + \cdots + \frac{1}{n^2}) \tag{11.8}$$

If we assume $n = 2^k$,

$$T = (2^k)^2(\frac{1}{4} + \frac{1}{16} + \cdots + \frac{1}{2^k 2^k}) \tag{11.9}$$
$$= 4^k(\frac{1}{4} + \frac{1}{16} + \cdots + \frac{1}{4^k}) \tag{11.10}$$
$$= 4^k(\frac{1}{4} + \frac{1}{4^2} + \cdots + \frac{1}{4^k}) \tag{11.11}$$

The sum inside the parenthesis is the form of a geometric series. The formula for a partial geometric sum is

$$\sum x^i = \frac{x^{k+1} - 1}{x - 1} \tag{11.12}$$

Thus, the time is

$$T = 4^k(\frac{1}{4} + \frac{1}{4^2} + \cdots + \frac{1}{4^k}) \tag{11.13}$$
$$= 4^k \sum_{i=1}^{k} \frac{1}{4^i} \tag{11.14}$$
$$= 4^k(\frac{\frac{1}{4^{k+1}} - 1}{\frac{1}{4} - 1} - 1) \tag{11.15}$$
$$= \frac{\frac{1}{4} - 4^k}{\frac{1}{4} - 1} - 4^k \tag{11.16}$$
$$= \frac{4^k - \frac{1}{4}}{\frac{3}{4}} - 4^k \tag{11.17}$$
$$= \frac{4^k - \frac{1}{4} - \frac{3}{4}4^k}{\frac{3}{4}} \tag{11.18}$$
$$= \frac{\frac{1}{4}4^k - \frac{1}{4}}{\frac{3}{4}} \tag{11.19}$$
$$= \frac{4^k - 1}{3} \tag{11.20}$$
$$= \frac{n^2 - 1}{3} \tag{11.21}$$
$$= \Theta(n^2) \tag{11.22}$$

This is the same as other algorithms, but in practice is still better.

11.3 Canaan's Algorithm

Also involves block matrices. If we use a grouping of 3-by-3 block matrices (i.e. 9 processors), then have each processor P_{ij} hold the subresult $A_{ij}B_{ij}$. We do some pretty crazy rearrangements. The idea is that if we compute the matrix multiplication with blocks, there is some slightly extra work to do. We do some different communication to make it work better.

To pipeline this, each of the p processors must compute an $\frac{N}{p} \times \frac{N}{p}$ matrix multiplication.

$$t_p = p(\Theta(\frac{n^2}{p^2}) + \Theta(\frac{n^3}{p^3}))$$

This is "cost optimal" because it divides our serial algorithm time by our processors.

12 Conway's Game of Life

The rules of Conway's Game of Life (note: Neighbors are all adjacent and diagonal cells)

1. If an occupied cell has 0 or 1 occupied neighbors then the cell becomes unoccupied (loneliness)
2. If an occupied cell has 2 or 3 occupied neighbors, the cell stays occupied.
3. If an occupied cell has 4 or more occupied neighbors, the cell becomes unoccupied (overcrowding)
4. If an unoccupied cell has exactly 3 occupied neighbors, the cell becomes occupied. Otherwise it stays empty