Fast Programs for Big Problems

John Burkardt
Department of Scientific Computing
Florida State University

....

Hosted by Professor Hyung-Chun Lee,
Mathematics Department,
Ajou University, Suwon, Korea.

...

https://people.sc.fsu.edu/∼jburkardt/presentations/...
matlab_fast_2011_ajou.pdf

27/29 September 2011

Fast Programs for Big Problems

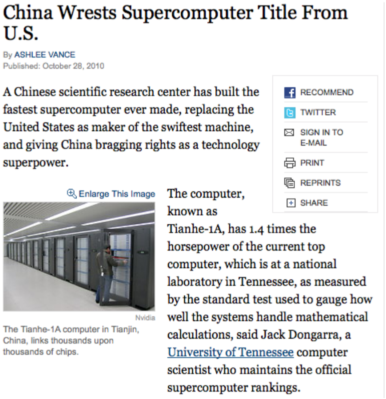
- **Introduction**
- Speed = Work / Time
- MD: Performance of a Molecular Dynamics Program
- NEIGHBORS: Performance of a Neighbor Program
- Complexity: How Calculations Grow
- Some Sample Calculations
- Conclusion

Introduction: A Machine This Big Is Necessary!

In June 2010, the fastest computer in the world was the Jaguar system, at Oak Ridge National Laboratories. It can run at 1.75 petaflops. (This is **part** of the machine!)

Introduction: Now Jaguar is a "Slow" Machine!

At the November 2010 Supercomputing meeting in New Orleans, the list of the biggest and fastest computers in the world was updated. China's 2.57 petaflop Tianhe machine became now the fastest.



China Wrests Supercomputer Title From U.S.

By ASHLEE VANCE
Published: October 28, 2010

A Chinese scientific research center has built the fastest supercomputer ever made, replacing the United States as maker of the swiftest machine, and giving China bragging rights as a technology superpower.

The Tianhe-1A computer in Tianjin, China, links thousands upon thousands of chips.

The computer, known as Tianhe-1A, has 1.4 times the horsepower of the current top computer, which is at a national laboratory in Tennessee, as measured by the standard test used to gauge how well the systems handle mathematical calculations, said Jack Dongarra, a University of Tennessee computer scientist who maintains the official supercomputer rankings.

Introduction: Problems, Algorithms, Machines Have Grown

Who needs these big machines?

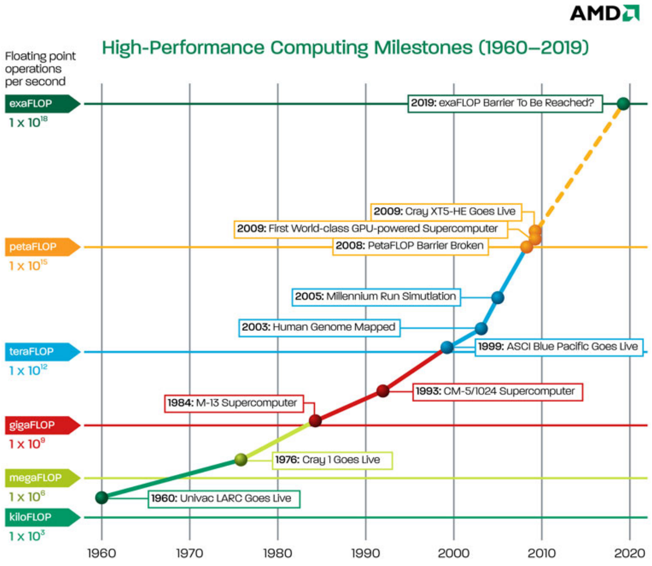
What kind of problems could possible be this large?

What does a **petaflop** measure?

These questions might seem to have nothing to do with the machines on our desktops. We don't really feel that it takes any significant time to calculate anything we are interested in.

And 20 years ago our desktop machines would have counted as supercomputers.

Introduction: An ExaFLOP = 1000 PetaFLOPs



Introduction: Measuring Computational Speed

A **megaflop** is a computational rate of a million floating operations per second. (Gigaflops are billions, teraflops trillions, and Petaflops are quadrillions).

A “floating point operation” is addition or multiplication, applied to a floating point (real) number.

If you can estimate the number of operations in a calculation, and if you can determine how long it took the computer to carry them out, you have computed the computational rate or performance for that computer on that problem.

To factor and solve a linear system of size **N** takes about $\frac{2}{3}N^3 + 2N^2$ operations. The program **linpack_bench** carries out this operation and reports the performance in megaflops.

Do this now: Run **linpack_bench** on your machine and note the MFLOPS number.

Introduction: Good Algorithms Beat Bad Ones

The performance value you get only tells you how fast the computer solved the problem as you described it. Sometimes there are faster ways to solve the same problem.

MATLAB can solve a linear system using the “backslash” notation. The backslash operation is highly optimized, so it can carry out the work as fast as possible.

Do this now: Run **linpack_bench_backslash** and note the MFLOPS.

I expect you will see that the same problem has been solved significantly faster. If the problem takes a long time to solve, then this is an important discovery, and we must pay attention to good computer performance!

But what is a “big” problem? How can we measure computer speed? And how can we try to improve a slow program?

Introduction: Sample Programs Available

During the talk, we'll refer to these programs:

- array1.m
- array1_first.m
- array1_twice.m
- array1_vector.m
- compute_old.m
- dot_product_graph.m
- floyd_graph.m
- heapsort_graph.m
- linpack_bench.m
- linpack_bench_backslash.m
- md.m
- sort_graph.m
- ticker.m
- triangulation_display.m
- triangulation_triangle_neighbors.m

Introduction: Sample Programs Available

During the talk, we'll also refer to various text files:

- big_cavity_elements.txt
- big_cavity_nodes.txt
- box3_elements.txt
- box3_nodes.txt
- ell3_edges.txt
- ell3_element_neighbors.txt
- ell3_elements.txt
- ell3_nodes.txt
- greenland_elements.txt
- greenland_nodes.txt
- lake_coarse_elements.txt
- lake_coarse_nodes.txt

The M files and TXT files are available at
http://people.sc.fsu.edu/∼jburkardt/latex/ajou_fast_2011/...
ajou_fast_2011.html

Fast Programs for Big Problems

SPEED: Relating Work and Time

To call a problem "big", we need to be able to measure the work the computer has been asked to do.

To call an algorithm, computation, or computer "fast", we need to be able to measure time.

If we can make these measurements, we can generalize the formula

```
Speed = Distance / Time,
```

to define

```
Computer Performance = Work / Time
```

SPEED: Measuring Time

tic and **toc** are a pair of MATLAB functions that allow you to measure the time it takes to carry out your work.

tic starts or restarts the timer; **toc** reports the elapsed time in seconds since **tic** was called.

Try this now:

```
tic
a = rand(1000,1000);
toc
tic
a = rand(1000,1000);
toc
```

Why will some people get very different answers?

SPEED: Measuring Time

If we use a MATLAB M file to define the work to be done, then MATLAB won't be measuring the time it takes us to type each line! Repeat the experiment, but do so by putting those 5 lines into an M file, called, for example, **ticker.m**.

Try this now:

```
ticker
```

Run the program and let's all tabulate our results.

Now, even if you run the program five times in a row, the results should be about the same.

SPEED: Measuring Work

Measuring the work involved in a computer program is harder than measuring time. For one thing, the computer does not execute our source code, but a lower level assembly language.

MATLAB, in particular, is partly an *interpreted* language; that means that often a lot of the computer's work is tied up in trying to figure out what you want, rather than in computing.

MATLAB includes precompiled or executable code that can run very fast, but it is always possible to do things in the worst possible way! Let us take a moment to consider what that means!

SPEED: An Experimental Calculation

Suppose we want to set the elements of a matrix A(I,J) using a formula based on I and J.

The obvious approach is to use a pair of **for** loops:

```
for i = 1 : m
  for j = 1 : n
    a(i,j) = sin ( i * pi / m ) * cos ( j * pi / n );
  end
end
```

As long as **m** and **n** are small numbers, there's no way it could matter whether this is the most efficient approach.

But we make two bad MATLAB programming decisions here.

SPEED: TIC and TOC to Time the Calculation

To see that something's wrong, let's make an M file called **array1.m**, and include an initial **tic** and final **toc** as well:

```
tic;
for i = 1 : m
  for j = 1 : n
    a(i,j) = sin ( i * pi / m ) * cos ( j * pi / n );
  end
end
toc
```

We are free to set the values of **m** and **n** outside the script.

SPEED: Slow, then Fast

The first thing to notice is a little peculiar:

Try this now:
Set M and N to 100, and then run **array1** 5 times.

What do you notice? Can you make it happen again?

I told you MATLAB is an interpreted language. The first time it reads your commands, it has to do a lot of set up, make sure it knows where the data is and the functions it needs to call. If the same command is repeated while MATLAB still has this setup information available, it can run much more quickly.

Debatable point: *do we want the fast time or the slow time?*

The **clear** command wipes out all that setup information, so you can go back to the slow code anytime.

SPEED: Counting the Work

To estimate the performance of this calculation, we ought to know how much work is involved in evaluating the formula. But **sin()** and **cos()** are not simple floating point operations, so we can't count the work that way. However, let's simply assume that computing each entry of the matrix costs the same work **W**. In that case, the total work in evaluating the whole matrix is

```
Work = M * N * W
```

So a matrix with 100 times as many elements has 100 times as much work, and presumably takes 100 times as much computer time.

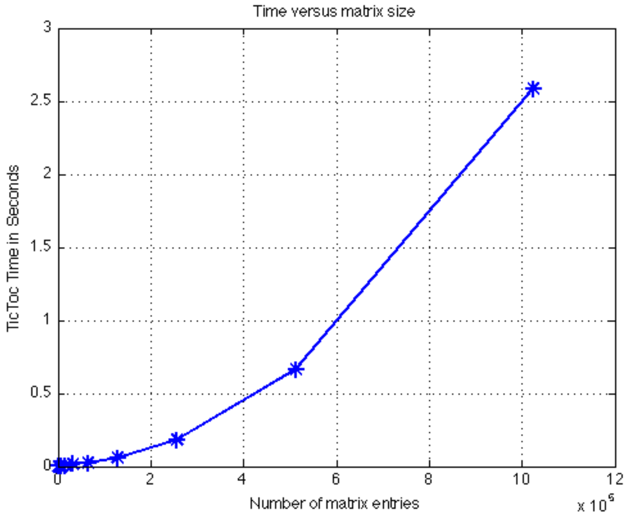
It's not hard to check this using a graph!

SPEED: A Sequence of Tasks

array1_once.m:

```
m = 1000;
n = 1;
for logn = 0 : 10
  tic;
  for i = 1 : m
    for j = 1 : n
      a(i,j) = sin ( i * pi / m ) * cos ( j * pi /n );
    end
  end
  x(logn+1) = m * n;
  y(logn+1) = toc;
  n = n * 2;
end
plot ( x, y, 'b-*', 'LineWidth', 2, ...
  'MarkerSize', 10 );
```

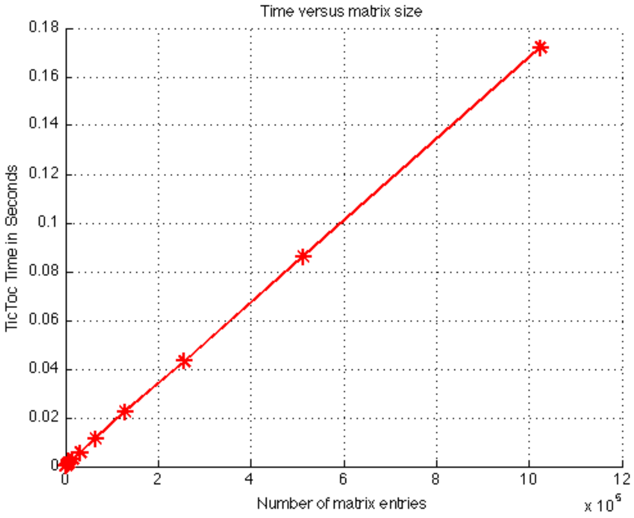
SPEED: Timing Data for ARRAY1_ONCE



Introduction

What happens if we run the program again, right away?

SPEED: Timing Data for Second Run of ARRAY1_ONCE



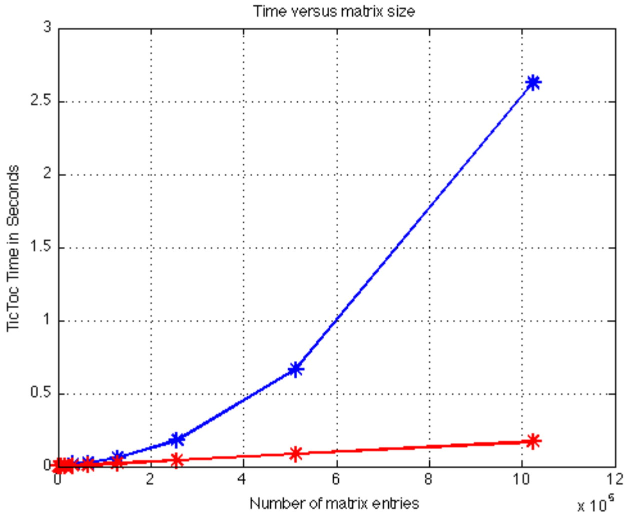
SPEED:

The program **array1_twice.m** runs the computation twice, plotting in blue the first time, and red the second.

(And it uses the **clear** command at the beginning, so we have a clean start!)

Perhaps if we plot the data together we can understand why the shape of the plot changed.

SPEED: First and Second Runs of ARRAY1_ONCE



SPEED: Compare First and Second Runs

When we look at the two graphs together, it becomes clear that something is wrong, or out of proportion. In both computations, on the last step, we compute 1,000,000 matrix entries. But somehow, something else is taking up 10 times as much time as the numerical calculation on the last step (2.6 seconds the first time and 0.18 the second.)

I mentioned that MATLAB is an interpreted language, and has to do some setup and interpretation of your statements. But here is a case where the setup is severely expensive...and easily fixed!

In MATLAB, you don't need to declare the size of an array. You can simply start using it. That means that MATLAB has to "guess" the space you need based on your commands, and it may have to adjust this guess as you issue new commands.

SPEED: MATLAB's Array Allocation

If your first statement is

```
x(100) = 5;
```

then MATLAB "realizes" you want an array **x** of size at least 100. Suppose your next statement is

```
x(200) = 5;
```

Then MATLAB realizes it didn't set aside enough space so it finds space for an array of size 200, copies the old version of **x** into the first 100 entries, and assigns x200.

This means that an array assignment can "accidentally" also be a request for more space.

SPEED: MATLAB's Array Allocation

Now we see the flaw in our program. Every time the **i** or **j** loop index increases by 1, MATLAB realizes we need one more row or one more column to hold the matrix, so it pauses in the calculation, finds spare memory to make a new array, copies the old data into the new array, and then moves to the next loop iteration.

Unless we use the **clear** command, MATLAB keeps this information around; so the second time we run the same program, all that space is already available and so the program runs 10 times faster.

So, is the key to running faster to run the program twice?
(Wrong conclusion!)

SPEED: Preallocate Arrays

MATLAB allows you to request space for an array before you use it. In particular, the **zeros** command is what we need!

Try this now:
Change the **array1_twice** program so that the line

```
for logn = 0 : 10
```

now reads:

```
for logn = 0 : 10
  a = zeros ( m, n );   <=   allocate space for the array!
```

Run the program. Do the two calculations take about the same time now?

SPEED: MATLAB's Editor Can Help

MATLAB's editor can spot and warn you about some inefficiencies like this.

If you use the editor to view a program, on the right hand margin of the window you will see a small red, orange or green box at the top, and possible orange or red tick marks further down, opposite lines of the program.

If you examine **array1.m** this way, you might see an orange box and an orange tick mark. Putting the mouse on the tick mark brings the following message:

Note:

The variable 'a' appears to change size on every loop iteration (within a script). Consider preallocating for speed.

This is the change we just made to **array1_twice.m**.

SPEED: Another Flaw

The second flaw in our program is again a MATLAB issue.

MATLAB allows you to program in the “scalar” style in which each command works with numbers in the ordinary way. But you are encouraged to use vector and array commands instead.

These commands can be simpler (fewer statements are needed), but they can also result in much faster execution.

The speedup can be significant enough that it’s even worthwhile to rewrite your program in a way that makes it look more complicated.

SPEED: MATLAB’s Vector Notation

Examples include:

```
for i = 1 : n          ==> a(1:n) = 7
  a(i) = 7
end

for i = 2 : 93         ==> b(2:93) = (2:93) / 17
  b(i) = i / 17
end

for i = 1 : m          ==>  c = a * x’
  c(i) = 0
  for j = 1 : n
    c(i) = c(i) + a(i,j) * x(j)
  end
end
```

SPEED: Using Vectors

Suppose we had vectors **s** and **c** containing the values

```
s(1:m) = sin ( pi * (1:m) / m )
c(1:n) = cos ( pi * (1:n) / n )
```

Then the entry a(i,j) is the ith entry of s times the jth entry of c

```
for i = 1 : m
  for j = 1 : n
    a(i,j) = s(i) * c(j)
  end
end
```

This is an **outer product**. It can be written in vector notation as

```
a = s' * c;  (if s and c are row vectors)
a = s  * c'; (if s and c are column vectors)
```

SPEED: Same Calculation, Different Forms

Try this now:

Modify **array1_twice** so that the first half of the program computes **a** the old way, but the second half computes vectors **s** and **c** first, and then evaluates **a** as an outer product.

```
s = sin ( ( 1 : m ) * pi / m ); <= (1xM) row vector
c = cos ( ( 1 : n ) * pi / n ); <= (1xN) row vector
a = s' * c;                     <= (MxN) array!
```

Now run the program: How do the plots compare?

SPEED: Same Calculation, Different Forms

We looked at three ways to compute the entries of the matrix A.

When the matrix is of size 1000 by 1000, the three ways have very different performance:

```
Method 1:  3.80 seconds
Method 2:  0.37 seconds  (allocate array)
Method 3:  0.02 seconds  (use MATLAB vectors)
```

so the first improvement produced a factor of 10 speedup, and the second a factor of about 15.

SPEED: Summary

Performance tells you how your computer and algorithm are handling a given task.

Computer time is part of this measurement;

If we can also measure "work", then it's possible to say more about the computational rate.

The "work" in computing includes not just the numerical computations in our formulas, but also "support work" like setup, interpretation, array resizing. Try to minimize support work, and look for optimized ways of carrying out certain operations.

Fast Programs for Big Problems

- Introduction
- Speed = Work / Time
- **MD: Performance of a Molecular Dynamics Program**
- NEIGHBORS: Performance of a Neighbor Program
- Complexity: How Calculations Grow
- Some Sample Calculations
- Conclusion

MD: Performance Profiles

If a program is small and the calculation is simple, it is easy to estimate how much work is involved, and to make experiments to see if a change to the code will make it run faster.

But very often, you start with a big, complicated code which performs several tasks, including reading input, interacting with the user, performing the calculation, writing output and making graphs.

You may not even have written the program yourself!

But you are told that it's important to measure its performance on a typical problem, and to identify parts of the code that take up a lot of time and that might be speeded up.

MD: How the Program Works

Our first example program, called **MD**, ("molecular dynamics") simulates the movement of a collection of particles which have a weak attraction to each other.

The program is given

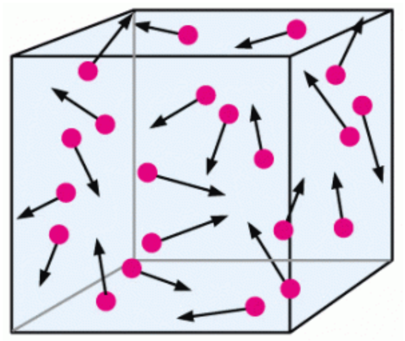
- **ND**, the number of space dimensions;
- **NP**, the number of particles;
- **STEP_NUM**, the number of time steps.

The program starts the particles at random positions, and then estimates their motion over the given number of time steps.

Depending on the problem size, the program might take a few seconds or minutes to run. We have been asked to locate the parts of the program that take up the most time.

The **MD** program has about 500 lines of code;
we need help to make an intelligent report!

MD: A Molecular Dynamics Simulation



Compute positions and velocities of **N** particles over time.
The particles exert a weak attractive force on each other.

MD: Run the Program

Since we don't know much about the program, let's keep things small, staying in 2D, with 100 particles and 20 time steps.

Try this now:

```
>> md ( 2, 100, 20 )
```

MD: Run the Program

```
>> md ( 2, 100, 20 )

Step      Potential          Kinetic          (P+K-E0)/E0
          Energy             Energy           Energy Error

  0      4843.049881          0.000000      0.000000e+00
  2      4843.049881          0.532737      1.100004e-04
  4      4829.529611         13.541667      4.417896e-06
...      ...                ...             ...
 18      4736.761673        262.217688      3.219655e-02
 20      4748.749293        295.544894      4.155322e-02

Main computation:
  Wall clock time = 21.216702 seconds.
```

MD: Bigger Problems Need Better Code

We'd like to go to 3 dimensions, use many more particles, and take more time steps. Can we solve bigger problems?

To estimate the cost of increasing a parameter requires knowing the **complexity** of the calculation, which we'll come back to later.

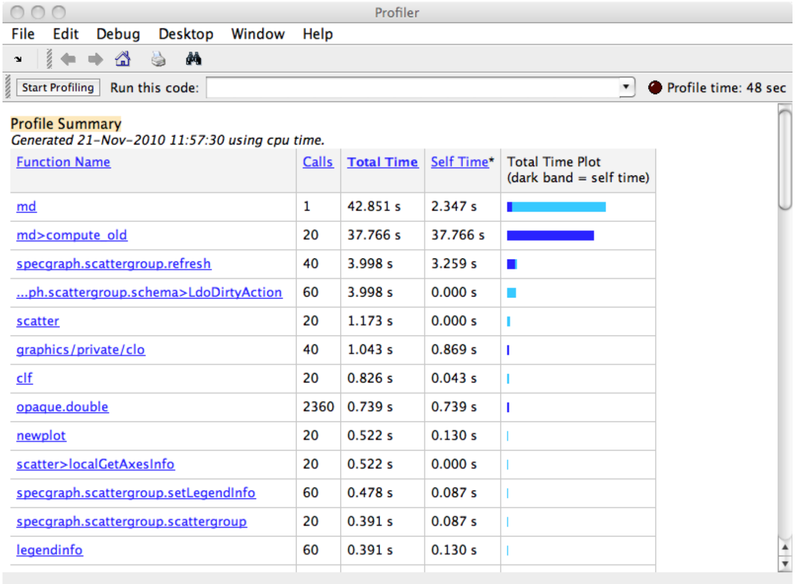
For now, we want to see where the code took the most time.

MATLAB has a **profile** command that can "watch" a program run and tell us a lot about it afterwards.

Do this now:

```
>> profile on
>> md ( 2, 100, 20 )
>> profile viewer
```

MD: Where is Execution Time Spent?



Profile time: 48 sec

Profile Summary

Generated 21-Nov-2010 11:57:30 using cpu time.

Function Name	Calls	Total Time	Self Time*	Total Time Plot (dark band = self time)
md	1	42.851 s	2.347 s	
md>compute_old	20	37.766 s	37.766 s	
specgraph.scattergroup.refresh	40	3.998 s	3.259 s	
...ph.scattergroup.schema>LdoDirtyAction	60	3.998 s	0.000 s	
scatter	20	1.173 s	0.000 s	
graphics/private/clo	40	1.043 s	0.869 s	
clf	20	0.826 s	0.043 s	
opaque.double	2360	0.739 s	0.739 s	
newplot	20	0.522 s	0.130 s	
scatter>localGetAxesInfo	20	0.522 s	0.000 s	
specgraph.scattergroup.setLegendInfo	60	0.478 s	0.087 s	
specgraph.scattergroup.scattergroup	20	0.391 s	0.087 s	
legendinfo	60	0.391 s	0.130 s	

MD: Where is Execution Time Spent?

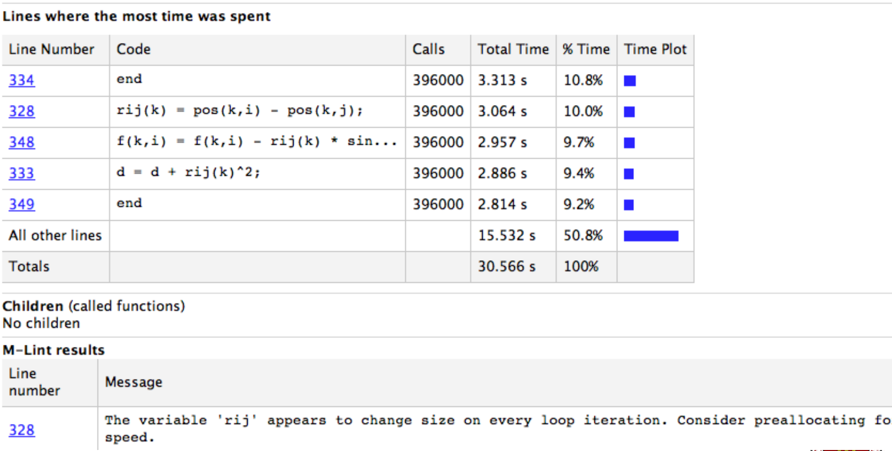
If you interpret the profile report, you can see that the **MD** function calls **COMPUTE_OLD** 20 times, and that about 38 seconds of the 43 second run time was spent inside of **COMPUTE_OLD**.

(Notice also that, because of adding the profiler, the wall clock time increased. The profiling work has been added to our total computation - but that will go away once we are done our analysis.)

By the way, use **profile off** when you are done, otherwise the profiler will slow down all your commands!

If we click on the line **md>compute_old** we can even see more detail about what went on in that function.

MD: Focus on the COMPUTE_OLD function



Lines where the most time was spent

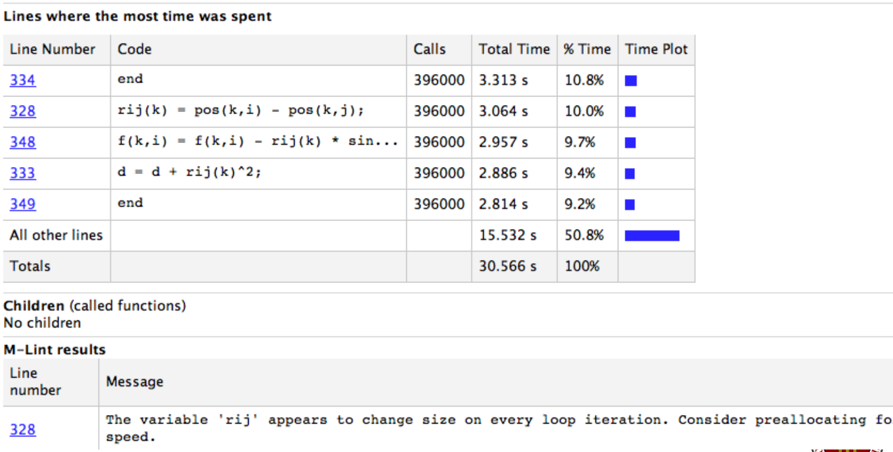
Line Number	Code	Calls	Total Time	% Time	Time Plot
334	`end`	396000	3.313 s	10.8%	■
328	`rij(k) = pos(k,i) - pos(k,j);`	396000	3.064 s	10.0%	■
348	`f(k,i) = f(k,i) - rij(k) * sin...`	396000	2.957 s	9.7%	■
333	`d = d + rij(k)^2;`	396000	2.886 s	9.4%	■
349	`end`	396000	2.814 s	9.2%	■
All other lines			15.532 s	50.8%	▬
Totals			30.566 s	100%	

Children (called functions)
No children

M-Lint results

Line number	Message
328	The variable 'rij' appears to change size on every loop iteration. Consider preallocating fo speed.

MD: Focus on the COMPUTE_OLD function



Lines where the most time was spent

Line Number	Code	Calls	Total Time	% Time	Time Plot
334	`end`	396000	3.313 s	10.8%	▇
328	`rij(k) = pos(k,i) - pos(k,j);`	396000	3.064 s	10.0%	▇
348	`f(k,i) = f(k,i) - rij(k) * sin...`	396000	2.957 s	9.7%	▇
333	`d = d + rij(k)^2;`	396000	2.886 s	9.4%	▇
349	`end`	396000	2.814 s	9.2%	▇
All other lines			15.532 s	50.8%	▇▇▇▇▇
Totals			30.566 s	100%	

Children (called functions)
No children

M-Lint results

Line number	Message
328	`The variable 'rij' appears to change size on every loop iteration. Consider preallocating fo` `speed.`

MD: Assignment

Try to make the MD program run faster!

1. First run the program, without making any changes, using the command **md(2,1000,100)**. How long does this take?
2. Turn off the plotting option by changing the line **show_plots=1** to **show_plots=0**. Now what is the run time?
3. Leave plotting off. Try to replace some **for** loops in **compute_old** with vector operations. What is your best run time now?

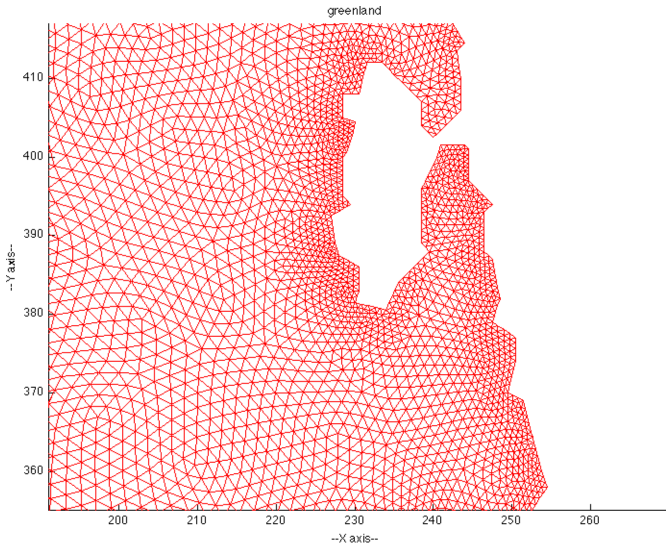
When you change **compute_old**, what is a simple check to make sure the program is probably still correct?

An acceptable report will list the three run times, and include a printout of your modifications to **compute_old**.

Fast Programs for Big Problems

- Introduction
- Speed = Work / Time
- MD: Performance of a Molecular Dynamics Program
- **NEIGHBORS: Performance of a Neighbor Program**
- Complexity: How Calculations Grow
- Some Sample Calculations
- Conclusion

NEIGHBORS: A Tiny Bit of Greenland



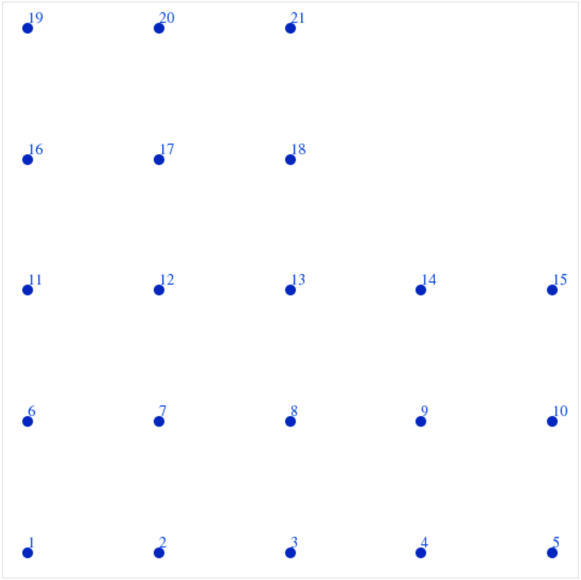
NEIGHBORS: The Triangle Neighbor Problem

Triangulation is a way of breaking up a large complicated region into simple triangles. We can think of the triangles as convenient local maps of the region. To keep track of things, we just need to know which triangle to examine.

But to keep track of *moving* things (such as the ice sheets of Greenland, or deer herds in Canada, or currents in the ocean), we need to know how to go from one triangle to any of its three neighbors.

If we assume everything possible is numbered, then our task is to identify the neighbor triangles in a list.

NEIGHBORS: The Nodes of a Simple Region

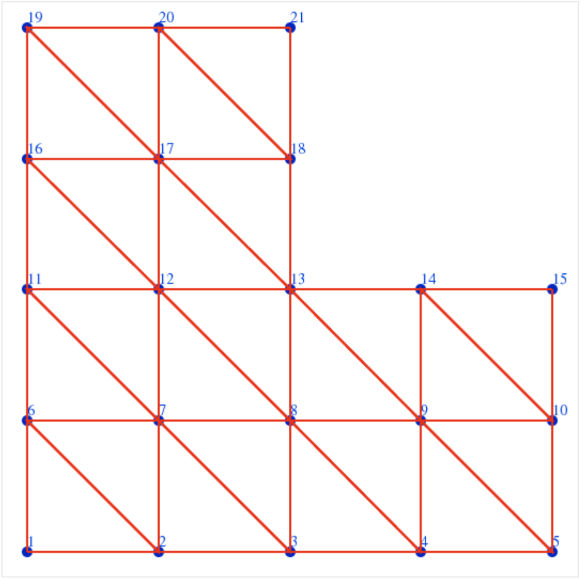


NEIGHBORS: The Nodes are Numbered

The node coordinates are supplied in a file that looks like this:

line:	X	Y
1:	0.0	0.0
2:	1.0	0.0
3:	2.0	0.0
4:	3.0	0.0
5:	4.0	0.0
6:	0.0	1.0
7:	1.0	1.0
	...	...
21:	2.0	4.0

NEIGHBORS: Triangles formed from Nodes

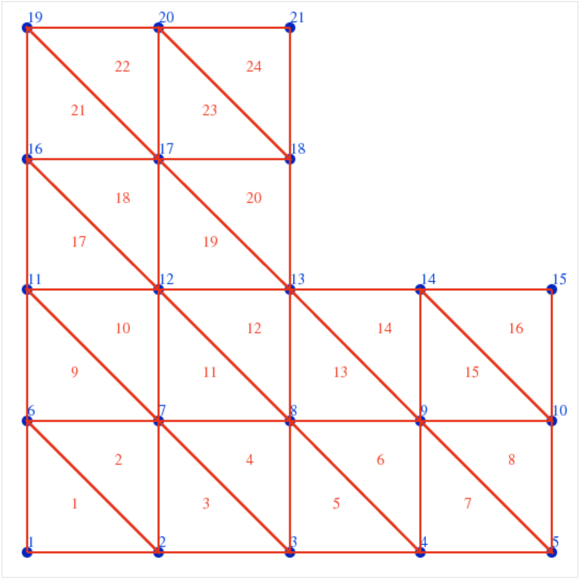


NEIGHBORS: The Triangles are Defined and Numbered

Each triangle is defined by three nodes (A,B,C):

Triangle:	**A**	**B**	**C**
1:	1	2	6
2:	7	6	2
3:	2	3	7
4:	8	7	3
5:	3	4	8
6:	9	8	4
7:	4	5	9
	...	...	...
24:	21	20	18

NEIGHBORS: Triangles Labeled in Red



NEIGHBORS: A List of Neighbors

We see the neighbors of triangle 2 are triangles 9, 1, and 3; triangle 21 has neighbors 18 and 22 (plus a missing neighbor).

The computer has no eyes: it *computes* neighbors **n1, n2, n3**:

Triangle:	**n1**	**n2**	**n3**
1:	2	*	*
2:	1	3	9
3:	4	2	*
4:	3	5	11
...	...	...	...
21:	22	*	18
22:	21	23	*
23:	24	22	20
24:	23	*	*

Neighbor **n1** is opposite node **A**, **n2** opposite **B**, **n3** opposite **C**.

NEIGHBORS: Triangle Neighbors

The computer has to "think" about this problem without a picture. Triangles 6 and 13 are neighbors. Triangle 6 uses nodes 4, 9, and 8. Triangle 13 uses nodes 8, 9 and 13. The fact that they are neighbors is hidden in the way that triangle 6 has a side using nodes 9 and 8, while triangle 13 has a side using nodes 8 and 9.

Perhaps we can detect neighbors by making a list of all the edges associated with each triangle, and looking for matches.

To make it easier to match, each edge will list the smaller node first. And each edge will have a third item, indicating the triangle it came from.

NEIGHBORS: The List of Edges

Here's how the data starts:

```
                Node1 Node2 Triangle
Triangle  1 ->  1     2      (1)
                2     6      (1)   <====+
                1     6      (1)        |
Triangle  2 ->  2     7      (2)      1 and 2 are neighbors!
                6     7      (2)        |
                2     6      (2)   <====+
Triangle  3 ->  2     3      (3)
                3     7      (3)
                2     7      (3)

               ..    ..     ....
Triangle 24 -> 18    21     (24)
               20    21     (24)
               18    20     (24)
```

Can we make this search efficient?

NEIGHBORS: Look for the right kind of SORT

Do this now:

The file **ell3_edges.txt** contains the list of edges for the figure we have been discussing. Read this data into MATLAB using a command like

```
edges = load ( 'ell3_edges.txt' );
```

We want to modify the file by putting, one after the other, the pairs of lines that correspond to matching triangle edges. In other words, the new file should have these lines one after the other:

```
2 6 1
2 6 2
```

We can do this with one MATLAB command...but it's not **sort***!*

NEIGHBORS: SORT Options

The MATLAB **sort** command has an optional argument to specify which dimension is to be sorted. And there's a **sortrows** command, too.

Do this now:
Try these commands and see which one does what we want:

```
edges2 = sort ( edges );
edges3 = sort ( edges, 1 );
edges4 = sort ( edges, 2 );
edges5 = sortrows ( edges );
```

You are hoping to find this pair of consecutive lines:

```
 2 6 1
 2 6 2
```

NEIGHBORS: Example Triangulations

Here are four examples of triangulations:

Name	Nodes	Triangles	Time
box3	20	24	0.06 seconds
lake_coarse	621	974	....
big_cavity	8,185	4,000	....
greenland	33,343	64,125	....

To see a picture of any of these triangulations, you can use the **triangulation_display** program:

Do this now:

```
  triangulation_display ( 'box3', 0, 1 )
```

NEIGHBORS: Running the Program on Small Data

To compute the neighbor information for the **box3** triangulation, we need the **triangulation_triangle_neighbors** program:

Do this now:

```
  triangulation_triangle_neighbors 'box3'
```

This will compute the neighbor information, store it to a list called **box3_element_neighbors.txt**, and print the time it took to compute the neighbors.

It only took me 0.06 seconds for the **box3** data.

NEIGHBORS: Using Bigger Data

We can see that the program runs quickly for the small **box3** problem. But we need to run the program on the other three sets of data as well.

Run the program on **lake_coarse** and record the time. It's about 30 times as much data. Does it take 30 times longer?

Can you guess how long **big_cavity** will take to run?

It should be clear that before we try to run **greenland**, it would be worthwhile to have the profile program take a look and see where the time is being spent.

Do this now:

```
profile on
triangulation_triangle_neighbors 'big_cavity'
profile viewer
```

NEIGHBORS: Identifying The Expensive Calculation

The profile viewer points to **i4col_sort_a** as using the time.

That function simply sorts the edges, as we did earlier with MATLAB's **sortrows** command.

Do this now:
In the **triangulation_triangle_neighbors** program, replace the line

```
  col = i4col_sort_a ( 4, 3*triangle_num, col );
```

by the line

```
  col = ( sortrows ( col' ) )';
```

(Be careful! We need those two "transposes" exactly as they appear here because MATLAB doesn't have a **sortcols** function.)

Rerun **big_cavity**. Did the execution time change significantly?

NEIGHBORS: Summary

Your experiences with MATLAB's performance profiler should give you a little bit of confidence that even if you didn't write the program, and it's a big program with lots of pieces, you can still get some idea of where the computational time is being spent.

If a lot if time is spent on a small amount of code, then we may be able to understand what that code is doing, and try to speed it up, without actually spending time measuring and understanding the entire program.

Fast Programs for Big Problems

- Introduction
- Speed = Work / Time
- MD: Performance of a Molecular Dynamics Program
- NEIGHBORS: Performance of a Neighbor Program
- **Complexity: How Calculations Grow**
- Some Sample Calculations

COMPLEXITY:

A computer calculation or algorithm is often thought of as an abstract machine or procedure that accepts input and produces output.

Sometimes, the value of one particular input quantity is a measure of how hard the calculation is going to be. Often this quantity is an integer, perhaps **N**, which might measure the length of an input vector, the number of iterative steps to take, or some other quantity that affects the amount of work.

It is sometimes possible to estimate the work **W**, the number of arithmetic operations performed, as a function of an input parameter such as **N**. Suppose we worked out an exact formula for the work and it came out to be:

$$W(N) = 23N^2 + 38N + 1447.$$

COMPLEXITY:

The details of the complexity formula are not important. The important thing is, if **W** can be written as a polynomial in **N**, then what is the highest power of **N** that shows up?

The reason this is important is that we want to know what happens to the work as N grows.

For our example formula,

$$W(N) = 23N^2 + 38N + 1447,$$

the highest power of **N** is 2, so we say that

- W has quadratic growth with N or
- W is order 2 in N or
- W is order N^2 or
- W is $O(N^2)$.

COMPLEXITY:

Having quadratic growth tells us something about how hard the problem is. If we want to solve a problem twice as big, there will be (about) 4 times the work. A problem 10 times as big has 100 times the work, and so on.

Knowing the complexity of an algorithm is important, because the same algorithm is likely to be used for a wide range of problem sizes. An algorithm whose work grows quadratically will hit the "computational ceiling" very quickly.

COMPLEXITY: Example Algorithms

- **binary search of sorted list**: $O(\log N)$
- **search an unsorted list**: $O(N)$
- **vector dot product: s = U'*V**: $O(N)$
- **Fast Fourier Transform**: $O(N \log N)$
- **Heap sort N numbers**: $O(N \log N)$
- **Diagonally Dominant Symmetric Iteration**: $O(N(\log N)^2)$
- **Gauss backsolve**: $O(N^2)$
- **Bubble sort N numbers**: $O(N^2)$
- **Gauss elimination**: $O(N^3)$
- **Matrix multiply A=B*C**: $O(N^3)$
- **Factor N digit number**: $O(2^N)$
- **Shortest round trip through N cities**: $O(N!)$

(It took two years and hundreds of computers to factor a single 232 digit number)

COMPLEXITY:

Let's suppose we're willing to do W = 1,000,000,000 operations. What size problem can we do?

- $O(\log N)$ - no limit on N
- $O(N)$: N = 1,000,000,000
- $O(N \log N)$ N = 50,000,000
- $O(N(\log N)^2)$ N = 4,000,000
- $O(N^2)$ N = 30,000
- $O(N^3)$ N = 1,000
- $O(2^N)$ N = 30
- $O(N!)$ N = 13

In other words, a high order algorithm will very quickly use up whatever work or time limit we allow.

COMPLEXITY: Time and Complexity

It's important to be able to estimate complexity, because program A may be faster than program B for a low value of N, but the advantage may change as N increases.

Since we don't want to figure out a formula for the work as a function of N, it is reasonable to try to get a feeling for the behavior of a function by timing it for a sequence of values of N over a representative range.

We should not be surprised if the results are meaningless for very small and very large values of N.

COMPLEXITY: Dot Product

Given two (column) vectors **U** and **V**, the scalar dot product is defined by:

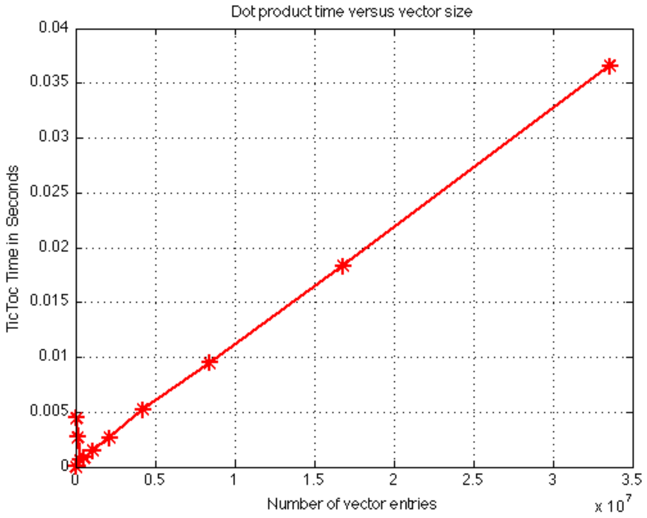
$$s = U'V = \sum_{i=1}^{N} u_i v_i$$

and can be computed in “about” **N** operations:

- 1 initialization
- 2***N** “fetches” from memory
- **N** multiplies
- **N** adds
- 1 write to memory

Counting only the **N+N** computational operations, we have an **O(N)** algorithm.

COMPLEXITY: Dot Product Timings



COMPLEXITY: Absolute performance

So far, we've only looked at relative performance; we compare the times as we double the problem size.

The computer I am using has a clock speed of 2.8 GigaHertz. Very roughly speaking, this means 2.8 billion things can "happen" in one second. A "thing" might be a numerical operation, for instance.

Let's compare that to the the work we did (2 * 33 million operations) and the time it took, about 0.037 seconds:

rate = 2 * 33,000,000 ops / 0.037 seconds = 1.8 billion (ops/sec)

This suggests that our operation count and timing are meaningful, and indicate that the computer is operating at about 65% of its maximum possible speed on this calculation.

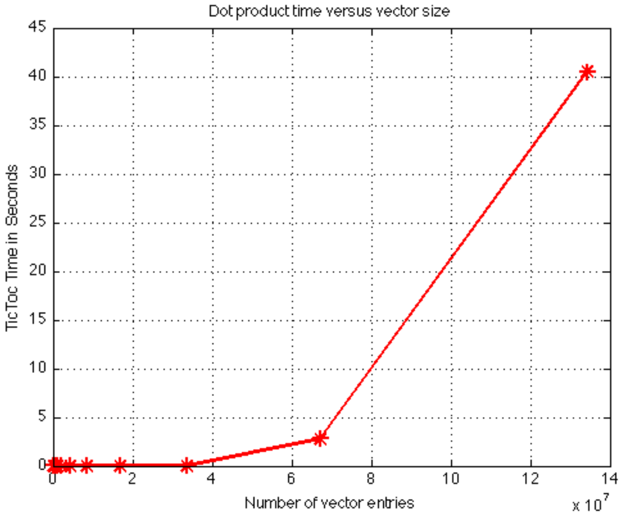
COMPLEXITY: Absolute performance

Try this now:

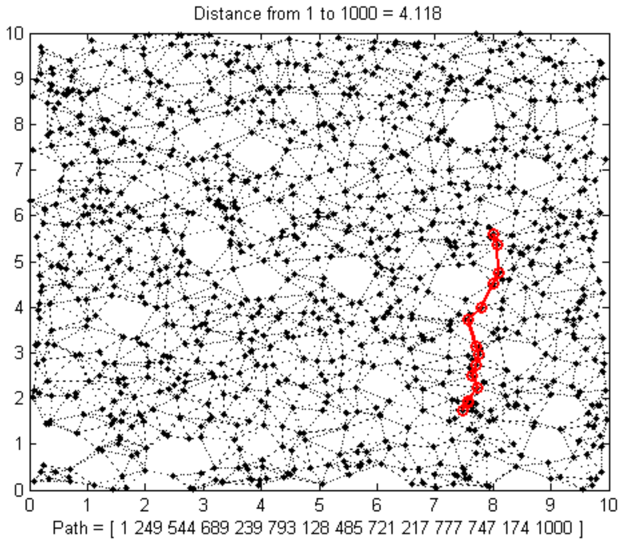
- Find the clock speed on the computer you are using.
- Run the dot product calculation, doubling from N=1 to N=2^{25}.
- Is your plot linear?
- Use the results of your calculation to determine the rate, in terms of (ops/sec).
- How does this compare to the clock speed?

COMPLEXITY: Dot Product Timings

What has gone wrong now? (Same plot as before except now we've added N $= 2^{26}$ and N $=2^{27}$)



COMPLEXITY:Floyd's Algorithm



COMPLEXITY: Floyd's Algorithm

Suppose we have **N** cities, and we are interested in determining the shortest time ST(I,J) to drive from any city I to any city J.

We have to assume that we start with a table that gives the driving time DT(I,J) for a direct trip from city I to each city J. Since most cities don't have a direct link, many of these values will be ∞.

But if a direct link doesn't exist, we can usually find many ways to get from I to J, and we want to find the shortest of all possible routes.

Between city I and city J there are N-2 other cities, so theoretically there are (N-2)! routes to check. Since there a total of N * (N-1) values of T(I,J) to compute, this seem like an O(N!) problem, also known as "impossible"!

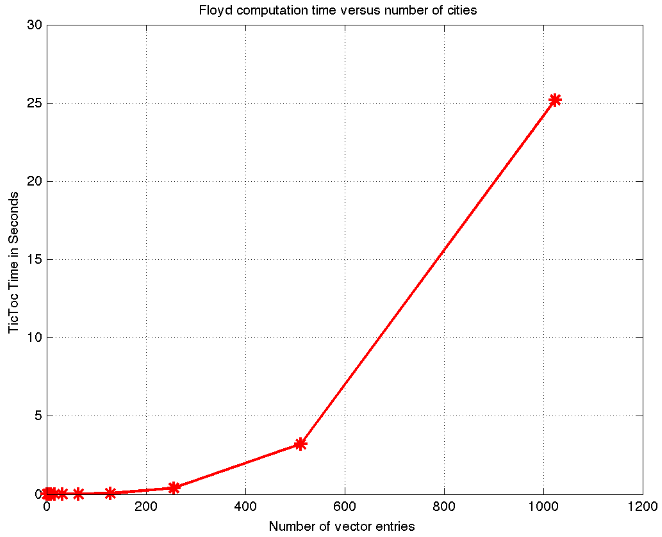
COMPLEXITY: Floyd's Algorithm

Instead of being impossible, Floyd's algorithm shows a simple way to compute the entire table in just a few lines of code:

```
st = dt
for k = 1 : n
  for j = 1 : n
    for i = 1 : n
      st(i,j) = min ( st(i,j), st(i,k) + st(k,j) )
    end
  end
end
```

Tell me right now:
What is the complexity or order of this algorithm?
What size is a "big" problem for Floyd's algorithm?

COMPLEXITY: Floyd's Algorithm Timed



COMPLEXITY: Floyd's Algorithm Timed

It's easy to be tricked by plots. Since we expect a cubic, we are prepared to accept this image. Is it really a cubic? It could be a quadratic. It could be that the last two data values are "wild", as we saw with the dot product results.

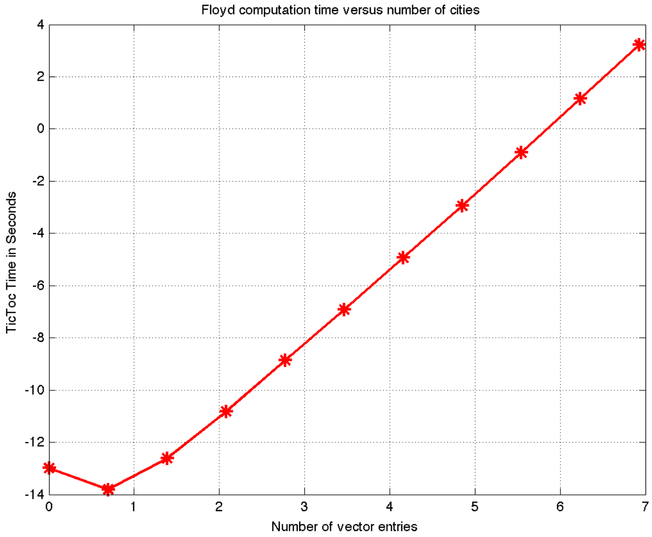
The human eye is **terrible** at detecting a quadratic or cubic curve. On the other hand, straight lines are a cinch!

If it's really true that $T \propto Work = O(N^3)$ so that roughly $T = c * N^3$ what happens if we take logarithms?

$$\log(T) = \log(c * N^3) = \log(c) + \log(N^3) = \log(c) + 3 * \log(N)$$

So plotting logarithms should reveal a straight line;
moreover, that line should have slope 3!

COMPLEXITY: Floyd's Algorithm Timed, Logarithmically



COMPLEXITY: Floyd's Algorithm Timed, Logarithmically

Rather than estimate the slope, let us print the values of

$$\frac{\log(time(k+1)) - \log(time(k))}{\log(n(k+1)) - \log(n(k))}$$

1	-2.769370
2	1.571803
3	2.515884
4	2.870909
5	2.789927
6	2.883388
7	2.898314
8	2.927246
9	2.960317
10	2.984813

COMPLEXITY: Conclusions

A program may have many lines of code, and yet, as the problem size increases, the behavior of the program becomes more and more like the highest order term in the complexity function.

Knowing the program complexity tells you how much it will cost you to solve a problem that is twice as big.

Knowing that one algorithm has a lower complexity than another, for example, $N \log N$ versus N^2, helps when looking for a better algorithm for large problems.

Plotting the run time over a range of N values can help to estimate the complexity, but you should always test your complexity formula in a way that would result in a straight line graph or a table of logarithms.

Fast Programs for Big Problems

SAMPLES: Suppose You Find this Interesting?

If you find the topics we have discussed to be of special interest to you, then the best thing you can do is try to explore some simple cases, make some intelligent attempts to solve the problems, keep track of your results, and hope that the patterns you find give you some ideas and insights.

The following examples all involve problems that can be made to be of any size **N**. As the problem size increases, we expect the work to increase. This depends, of course, on the algorithm you choose to solve the problem. Often the structure of your algorithm will tell you the complexity. Other times, you may need to estimate the complexity by doing some experiments.

Even those these examples are very simple, the problem of estimating complexity and choosing a good algorithm for large problems is something most scientific programmers must consider in all their work.

Example 1: The Largest Number in a List

Suppose we have a vector of **N** numbers **V**, and we wish to compute the largest value.

1) We can compute this value using a **for** loop and the **max** function, checking one element of **V** on each step.

2) We could apply the **max** function to the whole array in one step.

Presumably, the second approach is faster.

- Graph the times of the two operations as **N** increases;
- How do you expect work (and time) to grow with **N**?
- Is there some value of **N**, after which the second approach is usually 5 times faster than the first?

Example 2: Cheapest Bridge System

We have **N** islands, and we want to connect them. A bridge can be built to connect any pair of islands; (we don't allow two bridges to meet in mid-air and make extra connections!)

What is the smallest number of bridges that can be used so that every island is connected to all the others?

Short bridges are cheaper than long ones. *Dijkstra's algorithm* tells us how to decide which islands to using the shortest total bridge mileage.

- paint all islands red;
- start at island 1 and paint it blue.
- find the red island that is closest to some blue island, build a bridge to it, and paint it blue.
- repeat the previous step until all islands are blue.

Example 2: Cheapest Bridge System (continued)

This algorithm is a more "realistic" example of the kinds of problems that occur in computing. And for that reason, it is somewhat harder to say how much time and work is involved in the computation, that is, the order of the work as a function of the number of islands, **N**.

I won't ask you to program this problem. You can get the files you need for a program to solve this problem by going to http://people.sc.fsu.edu/∼jburkardt/m_src/dijkstra/dijkstra.html

This program takes **N** as input, choosing island distances randomly.

Your job is to run the program for an increasing sequence of values of **N**, record and graph the times, and try to estimate the order.

Example 3: Multiplying Matrices

We are given an **N**x**N** linear system **A*x=b**, with **x** unknown.

One way to solve the system is by Gauss elimination. MATLAB will do this for you automatically using the command

```
x1 = A \ b;
```

You could instead multiply **b** by the inverse of **A**:

```
x2 = inv ( A ) * b;
```

- estimate the orders of algorithms 1 and 2 using a sequence of increasing values of **N**
- let **e1 = norm(A * x1 - b)** and define **e2** similarly. Plot **e1** and **e2** together as **N** grows. Are the algorithms equally accurate?

Example 4: Traveling Salesman

A traveler begins at city 1, and wishes to visit N-1 other cities exactly once and then return home. The traveler wishes to minimize the total distance traveled.

The simplest approach to this problem generates every possible trip, adds up its distance, and “remembers” the shortest one.

How many trips are there if we have 6 cities? What about N cities?

Use trial and error to search for **Nbig**, the largest value of **N** for which the problem can be solved in under one minute. Then run the problem for each value of **N** from 1 up to **Nbig**, and make a plot. Compute the slopes. What seems to be happening?

Example 5: The Hailstone Problem

1. Pick any positive integer N.
2. If N is equal 1, stop.
3. If N is even, divide it by 2 and go back to step 1.
4. If N is odd, multiply it by 3 and add 1, then go to step 1.

Instead of trying to count floating point operations, simply count the number of steps it takes you to reach the value 1.

This problem is much harder to analyze, and you will have to think about how to summarize your data, since the results will not follow a smooth graph. Can you find a linear or quadratic function of N that is always above or below the running times?

Fast Programs for Big Problems

- Introduction
- Speed = Work / Time
- MD: Performance of a Molecular Dynamics Program
- NEIGHBORS: Performance of a Neighbor Program
- Complexity: How Calculations Grow
- Some Sample Calculations
- **Conclusion**

Conclusion

In your classes, you may be used to solving “toy” problems; a 10x10 matrix, a set of 3 ODE’s, a pair of nonlinear equations. These examples teach you one part of scientific computing, namely, the design and use of numerical algorithms.

However, if you need to to scientific computing for research or work, it is likely that you will be dealing with much bigger problems. The same algorithms may work correctly on big problems, but perhaps not always efficiently.

I hope I have told you how to estimate computer performance, look for trouble spots in your program, compare two ways of solving the same problem, and the many ways that problems become harder as they get bigger.

Conclusion: Thanks!

Thanks to my host, Professor Hyung-Chun Lee,
and to his students for their kind attention!