

# Parallel Programming Concepts

John Burkardt,  
Information Technology Department,  
Virginia Tech.

.....

Mathematics Department,  
Ajou University,  
Suwon, Korea,  
13 May 2009.

.....

[https://people.sc.fsu.edu/~jburkardt/presentations/  
parallel\\_2009\\_ajou.pdf](https://people.sc.fsu.edu/~jburkardt/presentations/parallel_2009_ajou.pdf)



- 1 **Sequential Computing and its Limits**
- 2 What Does Parallelism Look Like?
- 3 What Computations Can Be Parallel?
- 4 How Do We Run a Parallel Program?
- 5 Performance Measurement
- 6 Conclusion



# Sequential Computing and its Limits

For 50 years, computers have expanded (more memory) and accelerated (faster processing) and gotten cheaper.

As powerful computers have solved big problems, they have opened the door to the next problems, which are always bigger and harder.

Programming design has changed much more slowly; the same old techniques seemed faster because the hardware was faster.



# Sequential Computing and its Limits

The Grand Challenge problems such as weather prediction, protein structure, turbulent flow, chemical reactions, cannot be handled with today's computers.

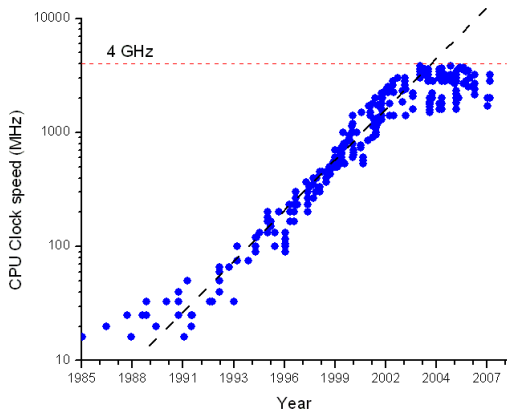
But now we can't expect these problems to become solvable simply by waiting for computing machinery to get even faster.

Here's some bad news:

**Computers have reached their maximum speed.**



# Sequential Computing and its Limits



In 2003, clock speeds stopped at 4 GigaHertz because of physical laws (size of atoms, the speed of light, excessive heat.)



# Sequential Computing and its Limits

If processor speed is fixed, the only way that computing power can grow is to combine the intelligence of several processors to cooperate on a given calculation.

And indeed, computer makers have been preparing new hardware to make this possible, including chips with multiple processors, processors with multiple cores, and ways for chips to communicate with other chips.

Software developers have invented hundreds of parallel languages. There now seem to be two common favorites, OpenMP and MPI.

And for MATLAB users, that language has developed its own parallel abilities.



# Sequential Computing and its Limits

To take advantage of these new opportunities, the programmer must enter the world of **Parallel Programming**.

While this world still contains all the familiar computational features of serial programming, there are many peculiar new issues of communication, interference, and data integrity.

We will try to outline the new paths to parallel programming that are available to you, give you a flavor of what such programs look like and how their efficiency can be evaluated, and prepare you for the extra issues that arise when designing and running a parallel program.



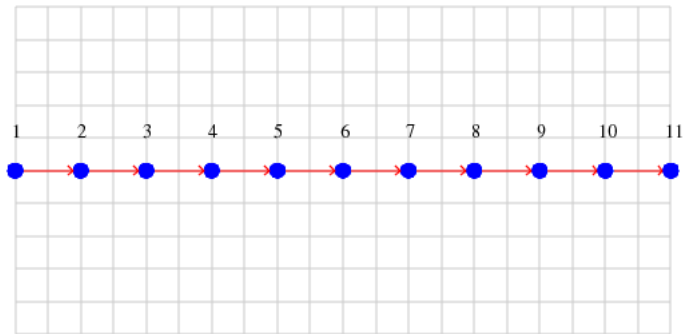
# Parallel Programming Concepts

- 1 Sequential Computing and its Limits
- 2 **What Does Parallelism Look Like?**
- 3 What Computations Can Be Parallel?
- 4 How Do We Run a Parallel Program?
- 5 Performance Measurement
- 6 Conclusion





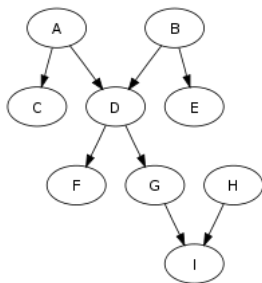
# What Does Parallelism Look Like?



A sequential program is a list of things to do, with the assumption that the tasks will be done in the given order.



# What Does Parallelism Look Like?



If we look at the logical dependence of our data, we are really dealing with a (directed acyclic) graph. We are free to work simultaneously on all calculations whose input is ready.



# What Does Parallelism Look Like?

Computational biologist Peter Beerli has a program named **MIGRATE** which infers population genetic parameters from genetic data using maximum likelihood by generating and analyzing random genealogies.

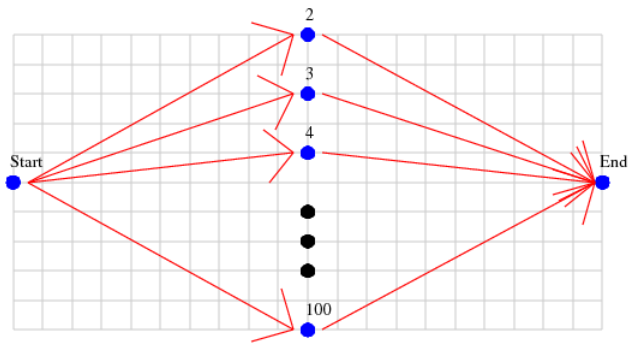
His computation involves:

- 1 an input task
- 2 *thousands* of genealogy generation tasks.
- 3 an averaging and output task



# What Does Parallelism Look Like?

In an **embarrassingly parallel** calculation, there's a tiny amount of startup and wrapup, and in between, complete independence.



## What Does Parallelism Look Like?

A more typical situation occurs in Gauss elimination of a matrix. Essentially, the number of tasks we have to carry out is equal to the number of entries of the matrix on the diagonal and below the diagonal.

A *diagonal task* seeks the largest element on or below the diagonal.

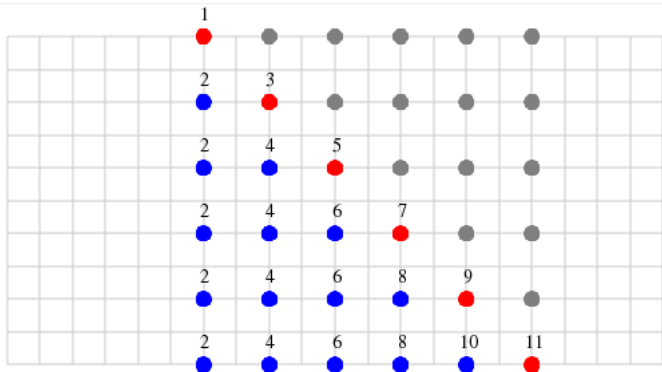
A *subdiagonal task* adds a multiple of the diagonal row that zeroes out the subdiagonal entry.

Tasks are ordered by column. For a given column, the diagonal task comes first. All the subdiagonal tasks are independent.



# What Does Parallelism Look Like?

In Gauss elimination, the number of independent tasks available varies from step to step.



# What Does Parallelism Look Like?

A natural place to look for parallelism is in loops.

The iterations must be independent, so that they could be performed in any order.

We need to check whether different iterations simultaneously try to read or update the same variables.

We will now look at some specific simple examples of computations and bits of associated code.



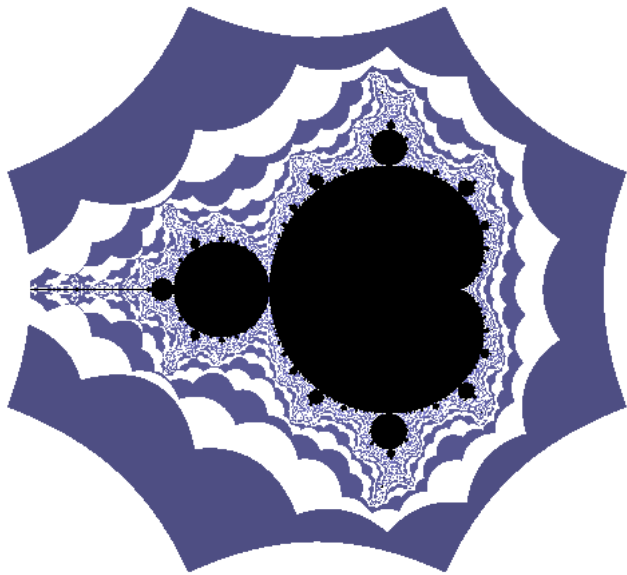
# Parallel Programming Concepts

- 1 Sequential Computing and its Limits
- 2 What Does Parallelism Look Like?
- 3 **What Computations Can Be Parallel?**
- 4 How Do We Run a Parallel Program?
- 5 Performance Measurement
- 6 Conclusion





# What Computations Can Be Parallel?



# What Computations Can Be Parallel?

```
do i = 1, n
  do j = 1, n

    x = ( ( n - j ) * xmin + ( j - 1 ) * xmax ) / ( n - 1 )
    y = ( ( n - i ) * ymin + ( i - 1 ) * ymax ) / ( n - 1 )
    pixel(i,j) = 0

    x1 = x
    y1 = y

    do k = 1, 1000

      x2 = x1 * x1 - y1 * y1 + x
      y2 = 2 * x1 * y1 + y

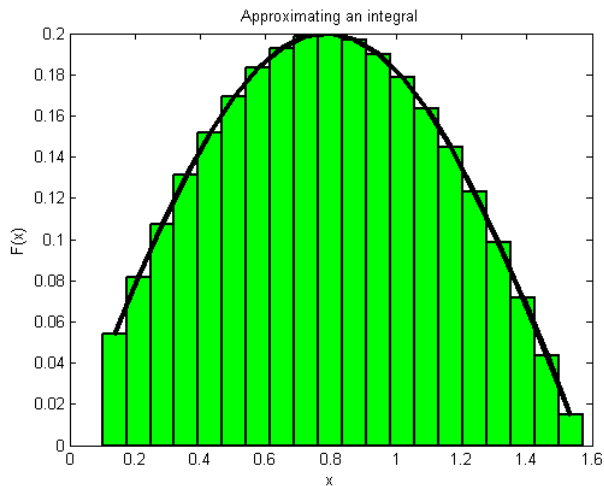
      if ( x2 < -2.0 .or. 2.0 < x2 .or. y2 < -2.0 .or. 2.0 < y2 ) then
        pixel(i,j) = 1
        exit
      end if

      x1 = x2
      y1 = y2

    end do
  end do
end do
```



# What Computations Can Be Parallel?



# What Computations Can Be Parallel?

```
a = 0.0
b = 1.6
h = ( b - a ) / n
quad = 0.0

do i = 1, n

  x = ( ( n - i ) * a + ( i - 1 ) * b ) / ( n - 1 )

  sump = ((((( &
    pp(1) &
    * x + pp(2) ) &
    * x + pp(3) ) &
    * x + pp(4) ) &
    * x + pp(5) ) &
    * x + pp(6) ) &
    * x + pp(7)

  sumq = ((((( &
    x + qq(1) ) &
    * x + qq(2) ) &
    * x + qq(3) ) &
    * x + qq(4) ) &
    * x + qq(5) ) &
    * x + qq(6)

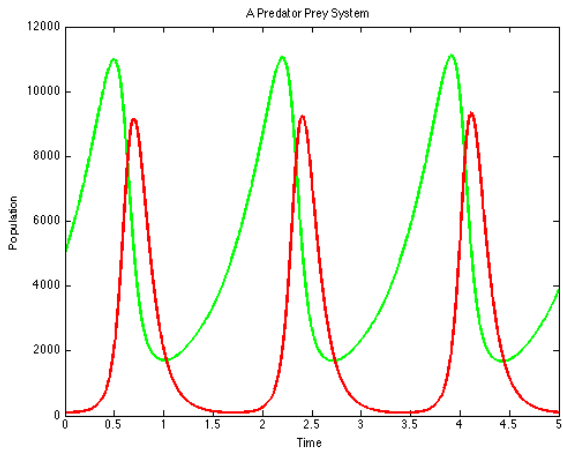
  fx = sump / sumq

  quad = quad + h * fx

end do
```



# What Computations Can Be Parallel?



# What Computations Can Be Parallel?

```
t_start = 0;
t_stop = 5;
dt = ( t_stop - t_start ) / step_num;

t = zeros(step_num+1,1);
pred = zeros(step_num+1,1);
prey = zeros(step_num+1,1);

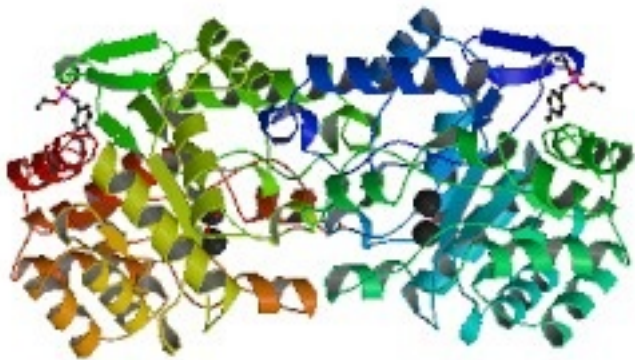
t(1) = t_start;
prey(1) = 5000;
pred(1) = 100;

for i = 1 : step_num
    t(i+1) = t(i) + dt;
    prey(i+1) = prey(i) + dt * ( 2 * prey(i) - 0.001 * prey(i) * pred(i) );
    pred(i+1) = pred(i) + dt * ( - 10 * pred(i) + 0.002 * prey(i) * pred(i) );
end

plot ( t, prey, 'g-', t, pred, 'r-' )
```



# What Computations Can Be Parallel?



# What Computations Can Be Parallel?

Align a sequence of M letters to another of N letters.

The matching must be done consecutively, but we can refuse to match some or all of the letters.

There are 5 possible alignments of a sequence of M=1 to a sequence of N=2:

a1 *	* a1	* a1 *	* * a1	a1 * *
b1 b2	b1 b2	b1 * b2	b1 b2 *	* b1 b2





# What Computations Can Be Parallel?

To figure out how many alignments can be made between sequences of length  $M$  and length  $N$ , we must fill in a table of  $M+1$  rows and  $N+1$  columns.

Row 0 and Column 0 are filled with 1's.

Fill in the rest of the table using the formula:

$$A(I,J) = A(I-1,J) + A(I-1,J-1) + A(I,J-1)$$

```
+-----+-----+
| A(I-1,J-1) | A(I-1,J) |
|-----|-----|
| A(I, J-1) | A(I, J) |
+-----+-----+
```



# What Computations Can Be Parallel?

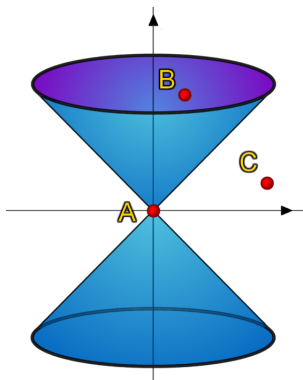
How much of this computation could be done in parallel?

M/N	0	1	2	3	4	5
	+-----					
0	: 1	1	1	1	1	1
1	: 1	3	5	7	9	11
2	: 1	5	13	25	41	61
3	: 1	7	25	63	129	231
4	: 1	9	41	129	321	681

```
do j = 1, n
  do i = 1, m
    A(I,J) = A(I-1,J) + A(I-1,J-1) + A(I,J-1)
  end do
end do
```



# What Does Parallelism Look Like?



We can't compute data A until its "past" is computed.  
We can't compute data B ("future") until A is done.  
Data C can be computed simultaneously with A.



# Parallel Programming Concepts

- 1 Sequential Computing and its Limits
- 2 What Does Parallelism Look Like?
- 3 What Computations Can Be Parallel?
- 4 **How Do We Run a Parallel Program?**
- 5 Performance Measurement
- 6 Conclusion



# How Do We Run a Parallel Program?

Our first example of parallel programming involves MATLAB.

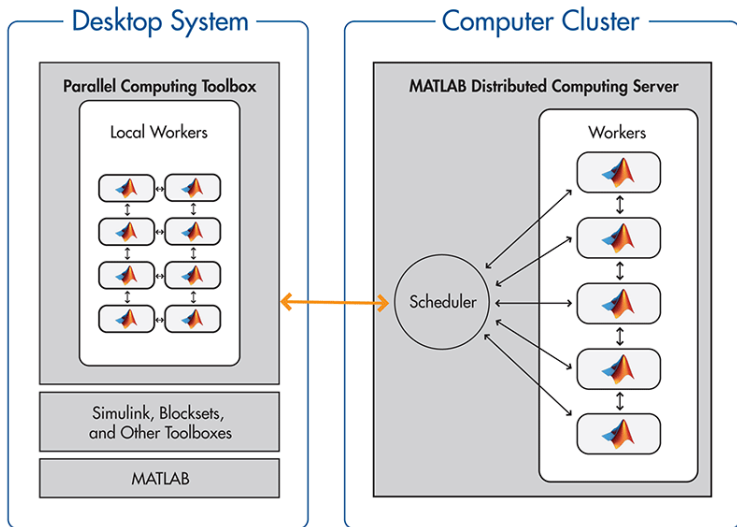
MATLAB has developed a *Parallel Computing Toolbox* and a *Distributed Computing Server* or **DCS**.

The Toolbox, by itself, allows a user to run a job in parallel on a desktop machine, using up to 4 "workers" (additional copies of MATLAB) to assist the main copy.

With the **DCS**, the user can start a job on the desktop that gets assistance from workers on a remote cluster, or (more efficiently) submit a batch MATLAB job to run in parallel on the cluster.



# How Do We Run a Parallel Program?



# How Do We Run a Parallel Program?

MATLAB also includes a **batch** command that allows you to write a script to run a job (parallel or not, remote or local) as a separate process.

MATLAB also includes the **spmd** command, (single program, multiple data) which allows it to distribute arrays across multiple processors, allowing you to work on problems too large for one machine.

If desired, you can exert much more control over the parallel execution using message passing functions based on the MPI standard.



# How Do We Run a Parallel Program?

Our second example of parallel programming involves Graphics Processing Units (GPU's).

Essentially, in order to produce realistic 3D animation for computer games, manufacturers have figured out how to exceed the speed of light (that is, the speed of sequential computing).

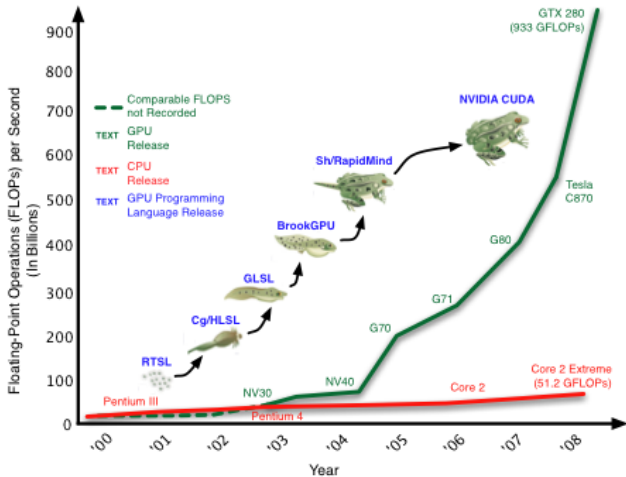
They've done this by realizing that each pixel represents a separate independent computation.

By providing multiple processors specialized for simple graphics tasks, and a programming language that organizes the computations, GPU manufacturers can offer what amounts to a 200-fold speedup over CPU's.





# How Do We Run a Parallel Program?



# How Do We Run a Parallel Program?

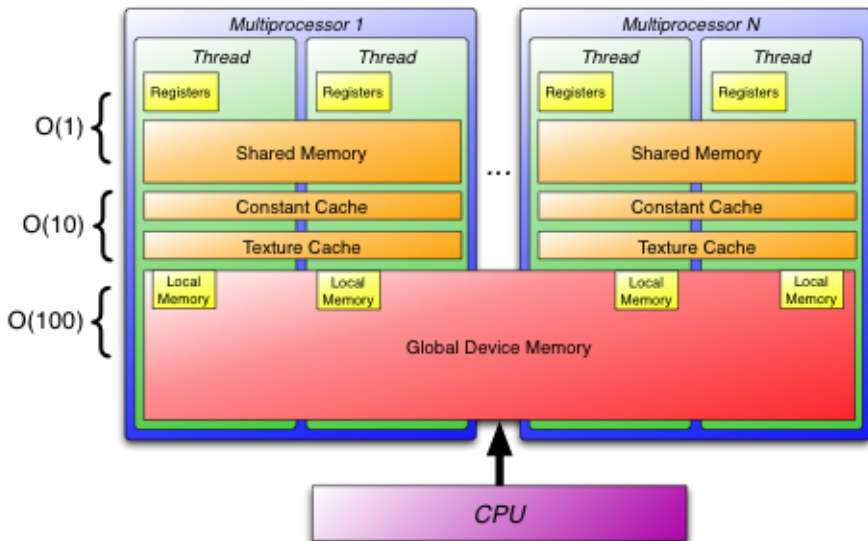
The GPU's also have the advantage that the graphics tasks they perform can be arranged in such a way that each thread of execution uses very local memory (registers, cache) as much as possible, avoiding expensive calls to remote data.

A single GPU contains multiple processors, each of which can run multiple threads of execution simultaneously.

Data is shared "instantaneously" between threads on a processor. Communication between processors is more costly, and communication that involves the CPU is the slowest.



# How Do We Run a Parallel Program?



# How Do We Run a Parallel Program?

OpenMP enables parallel programming on shared memory systems.

Instead of a single processor running one command at a time, we imagine several **threads** which cooperate.

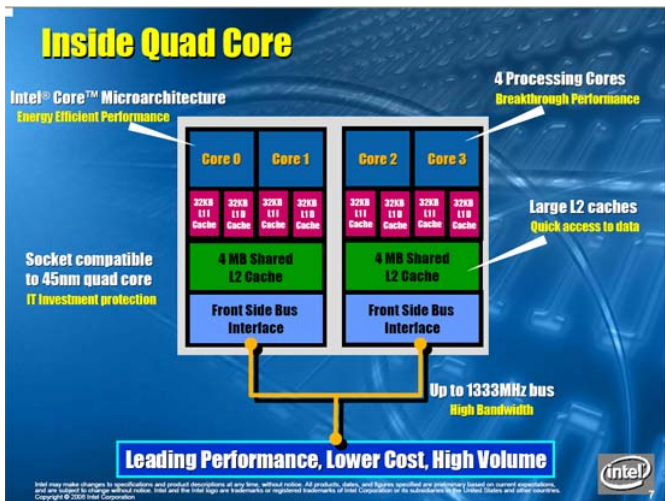
Each thread sees all the data, and can change any quantity.

Issues include:

- 1 **scheduling** the threads so they all have enough to do,
- 2 avoiding **ambiguity** and **interference**, where the value of a data item depends on "who gets to it" first.



# How Do We Run a Parallel Program?



# How Do We Run a Parallel Program?

OpenMP can run on a multicore laptop.

(Typically, this limits you to 2, 4 or perhaps 8 threads!)

The user simply needs a compiler with the OpenMP extensions.

Then the user inserts directives into a program to indicate where and how the parallel threads should execute.

OpenMP can run on a cluster, using special hardware and software to create a giant shared memory space, and many (32, 64, 128...) threads.



# How Do We Run a Parallel Program?

MPI enables parallel programming on distributed memory systems.

Instead of one program running on one processor, we imagine many copies of the program, which can communicate.

Each "process" only sees its local data. Communication with other processes is done by exchanging **messages**.

Issues include:

- 1 **communication** is slow, and must be choreographed,
- 2 The user must **modify the program** so that the data and work can be split up, and then combined at the end.



# How Do We Run a Parallel Program?





# How Do We Run a Parallel Program?

MPI runs naturally on clusters of tens, hundreds, or thousands of machines.

If a program can be rewritten to use MPI, then it requires the insertion of calls to send messages back and forth.

In addition, a program originally intended for sequential execution usually requires some significant modification and revisions to the data structures and the algorithm in order to work well with MPI.



# Parallel Programming Concepts

- 1 Sequential Computing and its Limits
- 2 What Does Parallelism Look Like?
- 3 What Computations Can Be Parallel?
- 4 How Do We Run a Parallel Program?
- 5 **Performance Measurement**
- 6 Conclusion



# Performance: Sequential Programs

The natural measurement for a computer program is some kind of computational rate, measured in floating point operations per second.

This involves two steps:

- counting the floating point operations
- measuring the time

For some special simple computations, the floating point operation count is easy to estimate.

The time is generally taken as the elapsed CPU time.



## Performance: Sequential Programs

Multiplication of two  $N$  by  $N$  matrices is easily estimated at  $2 * N^3$  operations, so an estimate of the MegaFLOPS rate might be done this way:

```
ctime = cputime ( );  
matrix_multiply ( n, n, n, a, b, c );  
ctime = cputime ( ) - ctime;  
mflops = 2 * n * n * n / 1000000.0 / ctime;
```

My Apple G5 PowerPC often runs between 100 and 200 MegaFLOPS, for instance.



# Performance: Parallel Programs

Measuring performance of a parallel program is done differently.

CPU time is not the appropriate thing to measure, since a correct value of CPU time would be summed over all the processors, and hence would be the same or more!

We are running the program in parallel, so we expect the answers to come out faster. The correct thing to measure is **elapsed wallclock time**.



## Performance: Depends on Number of Processors

Measuring wall clock time requires calling some function before and after you carry out an operation. The function's name varies, but the task looks like this in **OpenMP**:

```
seconds = omp_get_wtime ( )  
things to time  
seconds = omp_get_wtime ( ) - seconds
```

In **MPI**, you call **MPI\_Wtime()** and in **MATLAB** you call **tic** before, and **toc** afterwards.



## Performance: Depends on Number of Processors

Measuring wallclock time has the advantage that it captures both costs of parallel programming: **computation** and **communication**.

There are many hidden costs to parallel programming which mean that we cannot predict that using two processors will give us a code that runs twice as fast!

Practical experience suggests that we forget about trying to measure floating point operations, and simply record what happens to our timings as we increase the number of processors.



## Performance: Depends on Number of Processors

A disappointing fact to discover is that the parallel speedup will usually depend strongly on the number of processors  $P$ .

To examine the speedup, we can simply run the same program with  $P = 1, 2, 4, 8$  processors, recording the time as  $T(P)$ .

The speedup in going from 1 to  $P$  processors is then the ratio

$$\text{Speedup}(P) = \frac{T(1)}{T(P)}$$





# Performance: BLAST: $P = 1, 2, 4, 6, 8$



## Performance: Number of Processors

There is a nice interpretation of the scale in these plots.

Note that when we use 6 processors, our speedup is about 4.5.

This can be understood to say that when we use 6 processors, the program is getting the full benefit of 4.5 of them.

You can already see the curve flattening out for increasing **P**

At least for this problem, adding more processors helps, but a little less each time.



## Performance: Problem Size $N$

We know that performance can depend on the problem size  $N$  as well as the number of processors  $P$ . To get a feeling for the interplay of both parameters, it is useful to plot timings in which you solve problems of increasing size over a range of processors.

For a “healthy” parallel program on a “decent” size problem, you may expect that the runs of bigger problems stay closer to the ideal speedup curve.

The problem size  $N$  for the BLAST program is the length of the protein sequence.

Let's do timings for sequences of 5 different sizes  $N$ .



# Performance: BLAST with 5 Problem Sizes



Now we see a bigger picture!

For any particular problem size  $N$ , we will indeed gradually stray away from the perfect speedup curve.

But the effect is worst for small problems. As problem size increases, parallelism is efficient.

And if we had to choose, that's exactly what we would want:

**Parallel programming is most powerful for our biggest jobs, precisely the ones we cannot solve sequentially!**



# Parallel Programming Concepts

- 1 Sequential Computing and its Limits
- 2 What Does Parallelism Look Like?
- 3 What Computations Can Be Parallel?
- 4 How Do We Run a Parallel Program?
- 5 Performance Measurement
- 6 **Conclusion**



To gain the benefits of parallel programming, you have to actually write a parallel program.

Future lectures discuss parallel programming in specific languages:

- Parallel Matlab  
(available in 1 minute!)
- Parallel Programming with OpenMP  
(available as **ajou\_2009\_openmp.pdf**)
- Parallel Programming with MPI  
(available as **ajou\_2009\_mpi.pdf**)

