Measuring Parallelism
or
"I used MPI, what do you mean it's not efficient???"

Jon Johansson

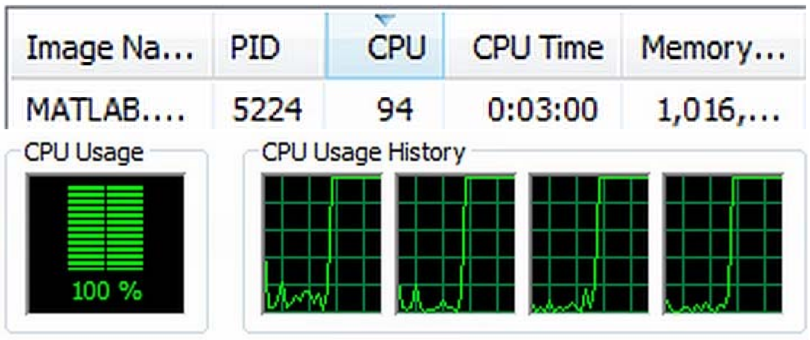
AICT, University of Alberta

Why Parallel Computing?

- we want to save time!
- we want answers now so we can move on to new questions
  - results/insight/understanding/publications in less time
- with a serial program doubling the number of calculations doubles the time it takes to get results
  - if we made a mistake when starting a job that takes 3 weeks we have to redo it
  - this wastes time (3 weeks, unless we get it wrong the second time)
    - a parallel program would waste less life-time for the same amount of computational work

Which Time?

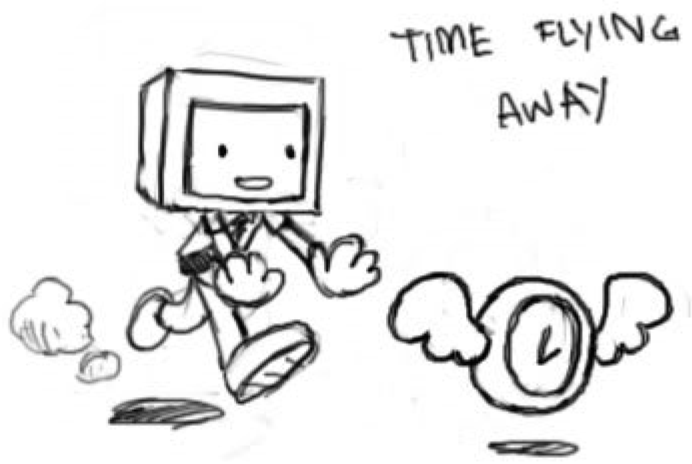
- ***Wall clock time:***
  - the time that passes on the clock watched by a human being waiting for his program to run
- ***CPU time:***
  - the time that the computer's CPU spends processing instructions for a particular task

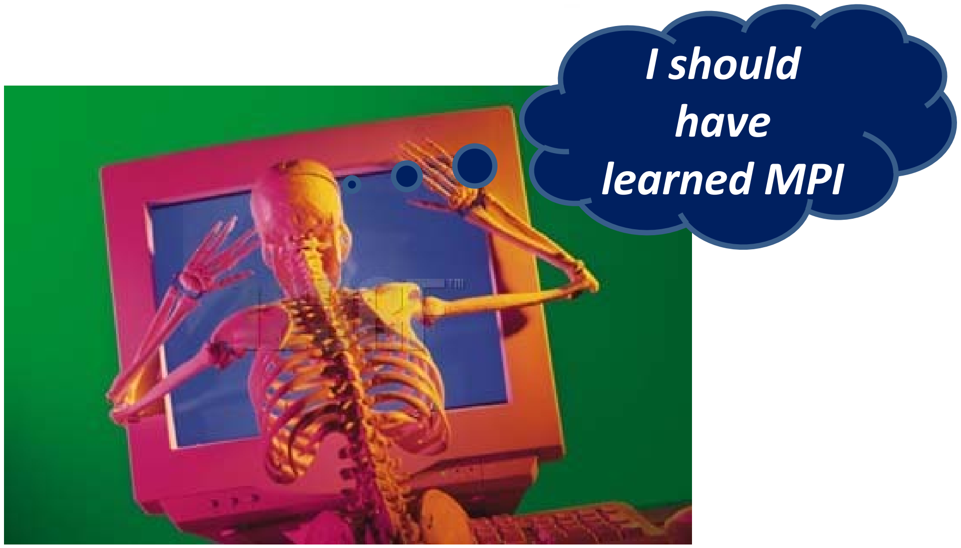


Matlab has used 3:00 min of cpu time – 45 secs wall clock time on 4 cpus

Parallel Processors Compress Time

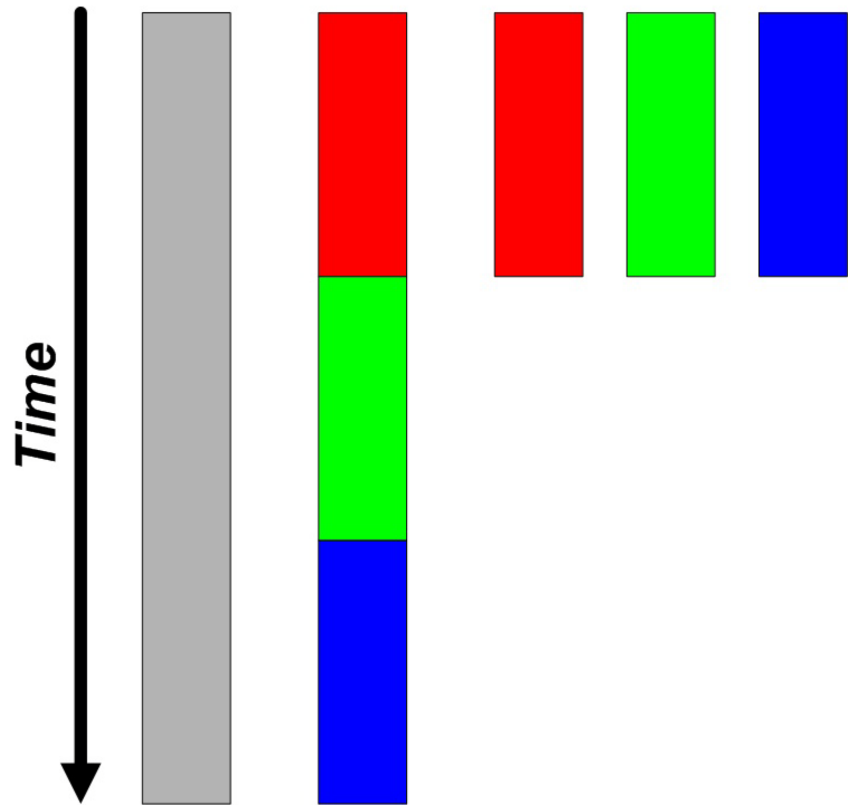
- by harnessing multiple CPUs/cores we compress a bunch of calculation time (CPU time) into a short amount of wall time (the measure of your life)
- if we have 100 processors working on our program exclusively, 24 hours/day we are effectively living one hundred times longer
  - perhaps this is a bit of an overstatement, but lets move on





Split the Work

- try to split the work into pieces that can be done by different processes
- distribute the work to the CPUS
- we can split the tasks or the data

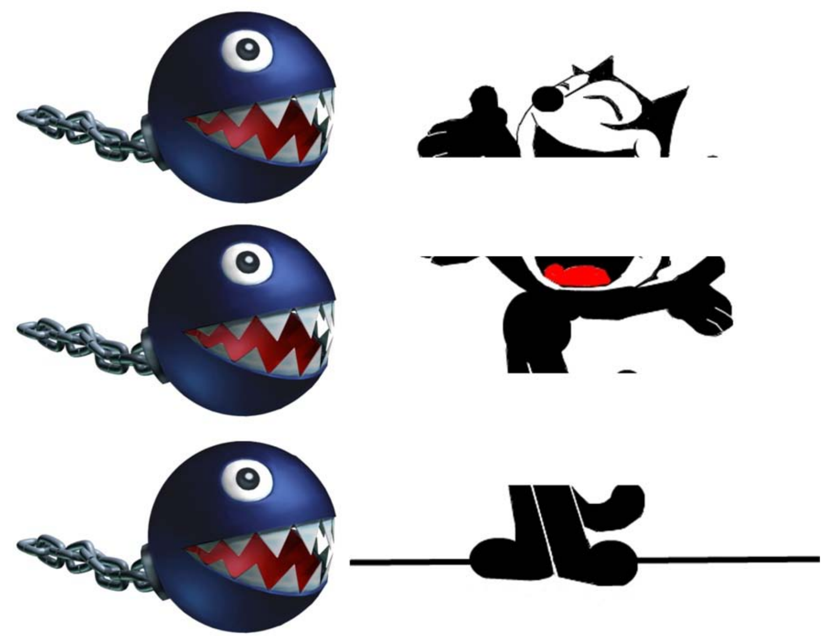


Types of Parallelism

Task parallelism

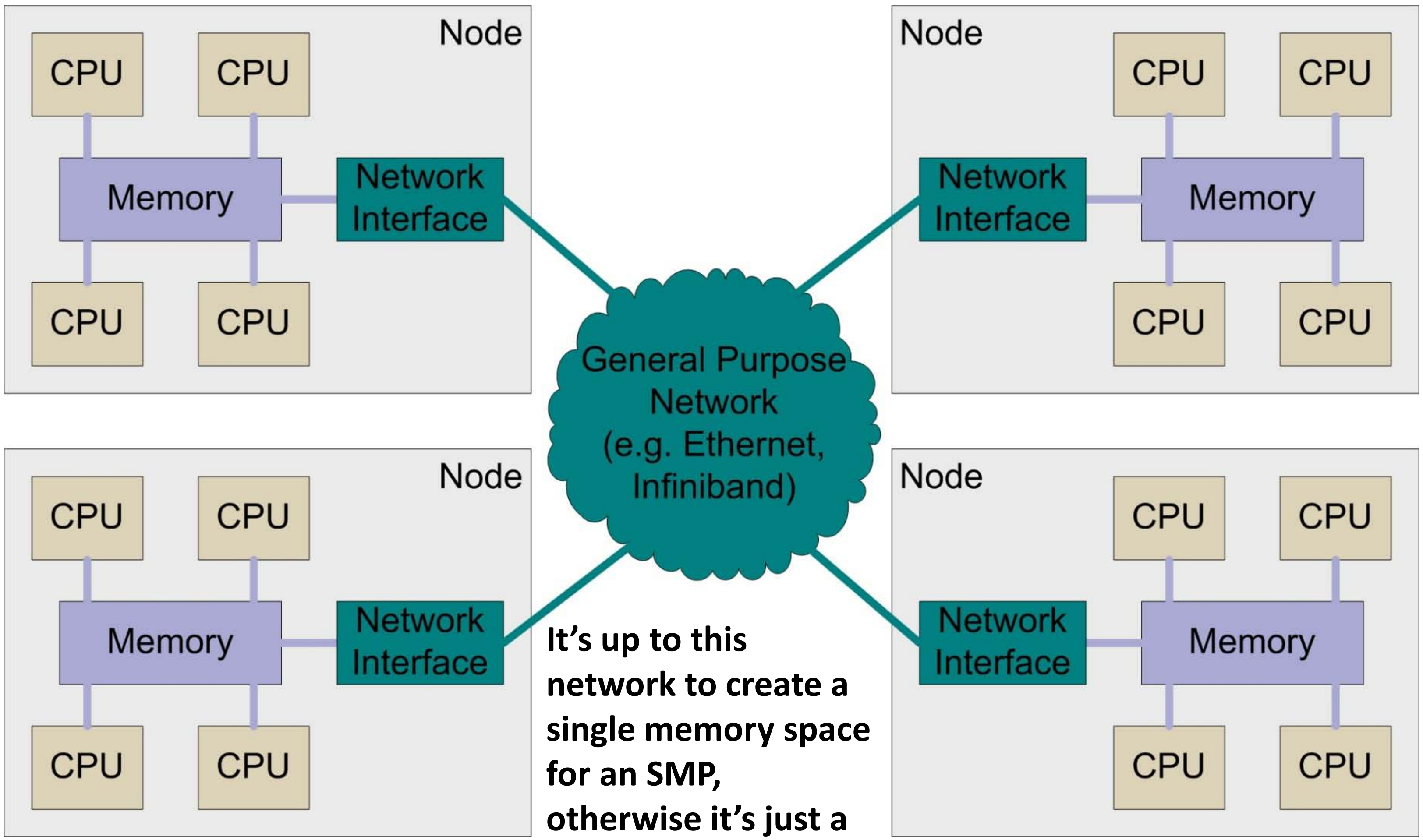
Different programs operate
on copies of the same data
- *functional decomposition*

Data parallelism



Copies of the same program
operate on different data
- *domain decomposition*

Today's Common System Architecture



It's up to this network to create a single memory space for an SMP, otherwise it's just a cluster

Speedup

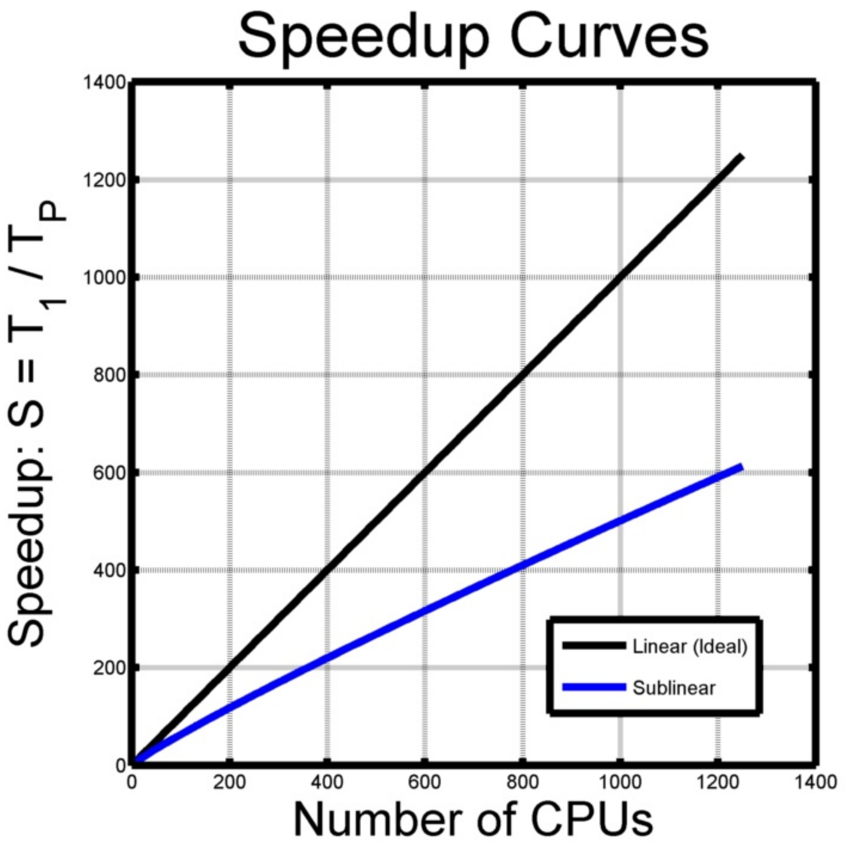
- how can we measure how much faster our program runs when using more than one processor?
- define **Speedup *S*** as:
  - the ratio of 2 program execution times
  - constant problem size

$$S = \frac{T_1}{T_P}$$

- T_1 is the execution time for the problem on a single processor
  - ***Absolute Speedup*** if this is measured with the “best” serial implementation
  - ***Relative Speedup*** if we use the parallel implementation with one CPU
    - remember that algorithms probably change when moving to parallel
- T_P is the execution time for the problem using P processors

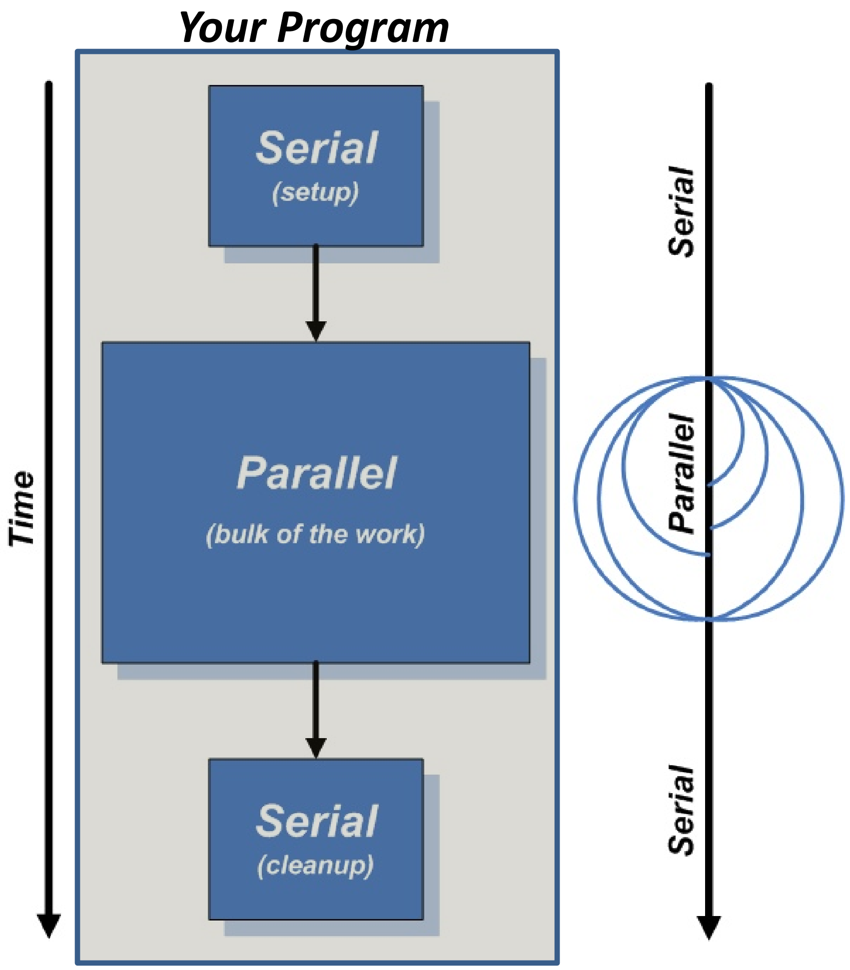
Speedup

- **Ideal (Linear) speedup**
  - the time to execute the problem decreases by the number of processors
  - if a job requires 1 week with 1 processor it will take less that 10 minutes with 1024 processors
- referred to as:
  - ***embarrassingly parallel***
  - stupidly parallel
  - perfectly parallel
- doesn’t take much effort to turn the problem into a bunch of parts that can be run in parallel:
  - parameter searches
  - rendering the frames in a computer animation
  - brute force searches in cryptography



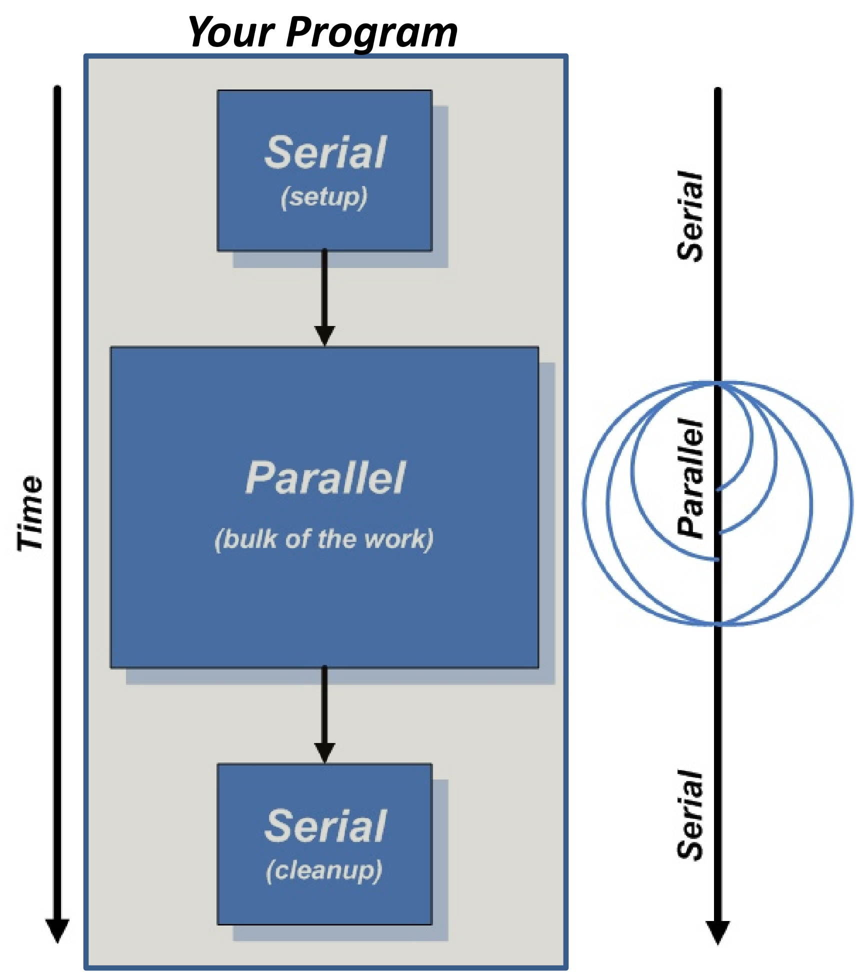
A Parallel Program

- if we can use many CPUs efficiently, we can
  - run simulations faster
  - increase problem sizes
  - run simulations at greater accuracy
- run a program on a cpu that can provide 1 gigaflop/s (10^9 flop/s)
- if you need 1 teraflop/s (10^{12} flop/s) to finish the calculation in a reasonable time you can use 1000 cpus
  - you need to use them efficiently!



A Parallel Program

- some of the program is serial, some parallel
- would like to use a number of processors at the same time to speed up calculations
  - the problem must be broken into parts that can be solved concurrently
  - each part of the problem becomes a program to run on its own processor



Example: Convolution - discrete

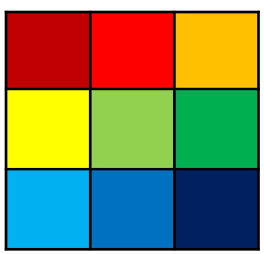
- in 2 dimensions the convolution is:

$$(f * g)_{m,n} = \sum_{i,j=-\infty}^{\infty} f_{i,j}\, g_{m-i,n-j}$$

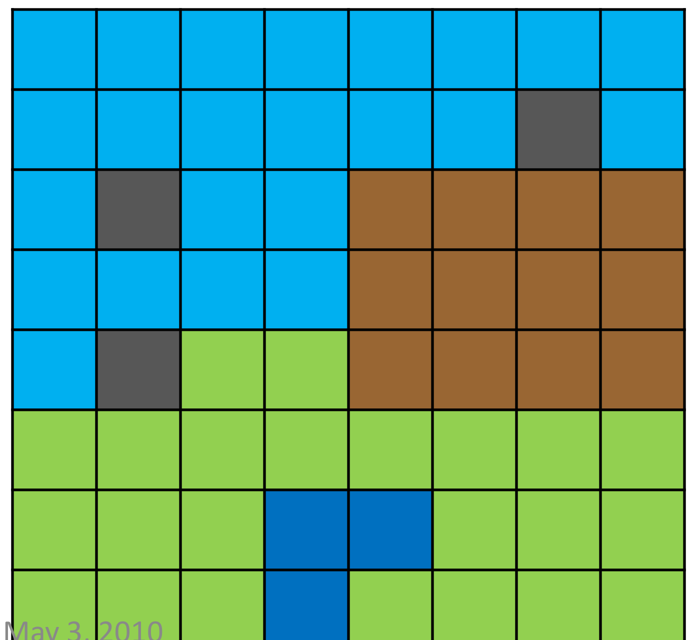
$$= \sum_{i,j=-\infty}^{\infty} f_{m-i,n-j}\, g_{i,j}$$

- apply a 3x3 filter to the image

$g \rightarrow$

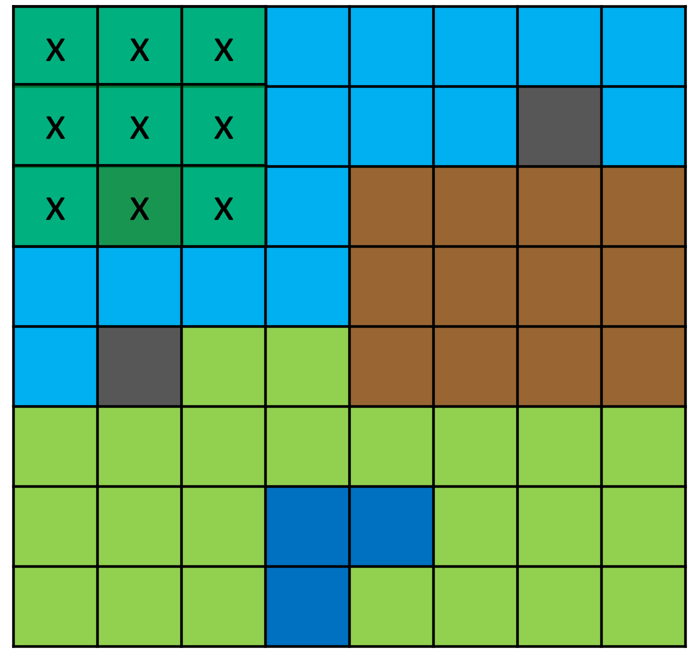


$f \rightarrow$



Convolution - discrete

- for each image point:
  - multiply the corresponding filter and image values
  - sum the result
  - multiply by a normalizing factor if necessary
- for a 3x3 filter each new image point requires 9 multiplies and 8 adds



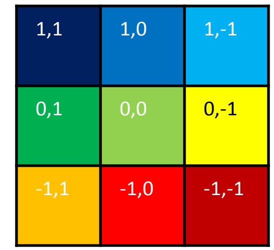
Convolution - discrete

$$(f * g)_{2,2} = \sum_{i,j=1}^{3} f_{i,j}\, g_{2-i,2-j}$$

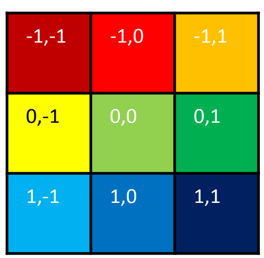
$$= f_{1,1}\, g_{1,1} + f_{1,2}\, g_{1,0} + f_{1,3}\, g_{1,-1}$$

$$+ f_{2,1}\, g_{0,1} + f_{2,2}\, g_{0,0} + f_{2,3}\, g_{0,-1}$$

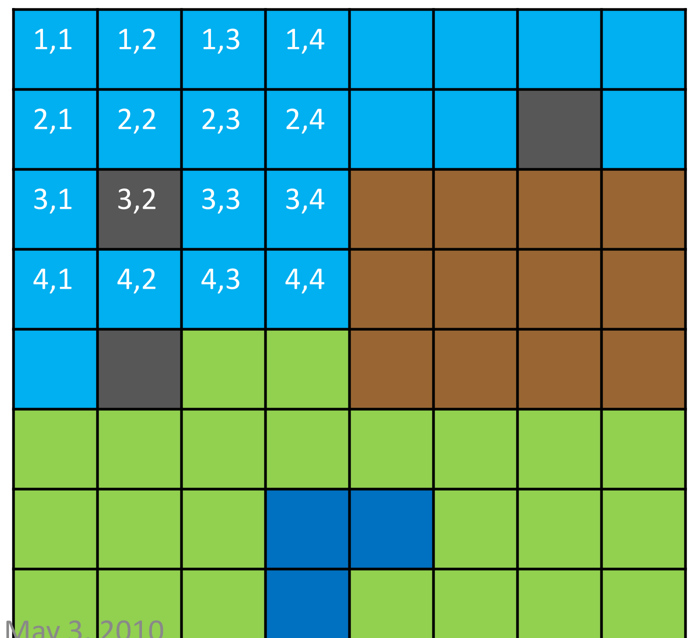
$$+ f_{3,1}\, g_{-1,1} + f_{3,2}\, g_{-1,0} + f_{3,3}\, g_{-1,-1}$$



$g \rightarrow$



$f \rightarrow$



Convolution Calculation – Serial

- the calculation involves 4 nested loops
- two outside loops move over the image
- two inside loops do multiplication and sum for new image point

- image size: 7000x7000
- filter size: 7x7

- Serial times:
  - program: 34.13 sec
  - loops: 29.7 sec
  - serial section: 4.42 sec
  - max speedup = 34.13/4.42 = 7.7

```
for(i = offset; i < nx + offset; i++) {
  for(j = offset; j < ny + offset; j++) {
    // Operate in each pixel in the
      image with the filter
    sum = 0.0;
    for(m = 0; m < nf; m++) {
      for(n = 0; n < nf; n++) {
        sum = sum + filter[m][n] *
        paddedimage[i-offset+m][j-
        offset+n];
      }
    }
    newimage[i][j] = sum;
  }
}
```

Convolution Calculation - OpenMP

- to create OpenMP threads and tell OpenMP that we are parallelizing a loop we can combine two directives:

  #pragma omp parallel
- and

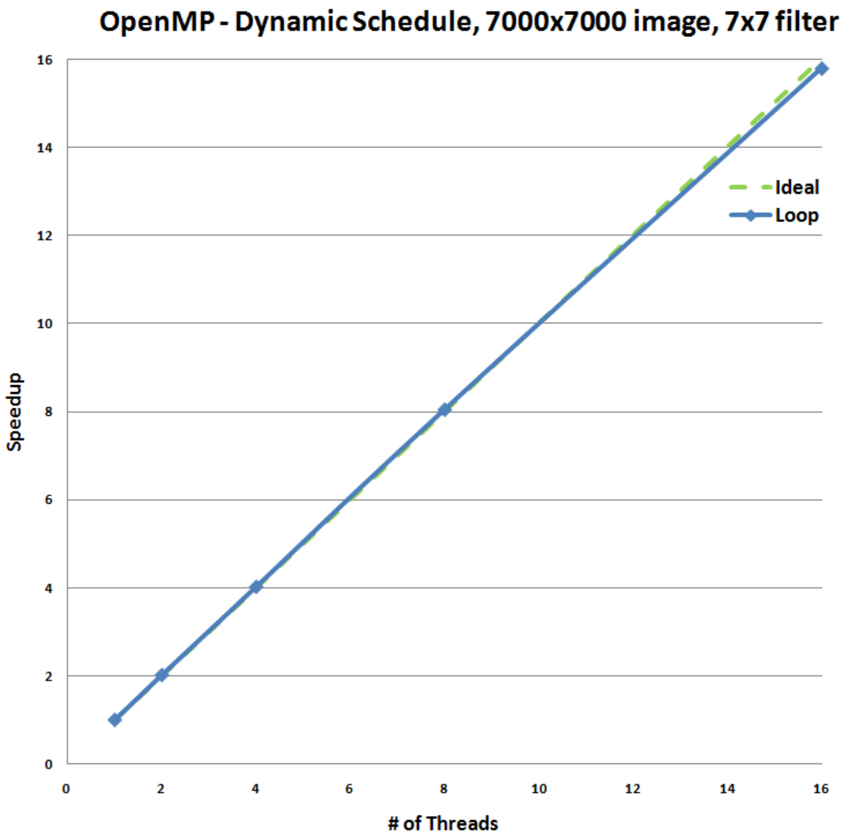
  #pragma omp for
- into the directive

  #pragma omp parallel for

```
#pragma omp parallel for private (i,j,m,n,sum)
        schedule(dynamic,1)
{
for(i = offset; i < nx + offset; i++) {
  for(j = offset; j < ny + offset; j++) {
    // Operate in each pixel in the image with the
       filter
    sum = 0.0;
    for(m = 0; m < nf; m++) {
     for(n = 0; n < nf; n++) {
       sum = sum + filter[m][n] * paddedimage[i-
        offset+m][j-offset+n];
     }
    }
    newimage[i][j] = sum;
  }
 }
}
```

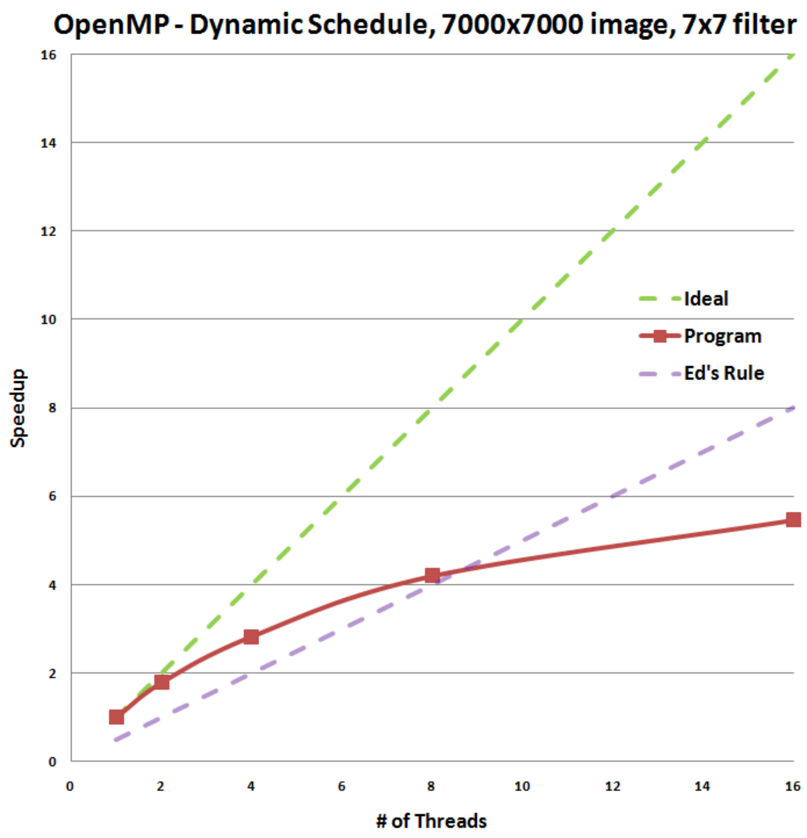
Speedup Example

- timing the parallel section shows that the loops parallel very well
- it might be fair to say that most effort to parallelize programs happens in loops
- recall that in the serial program we measured
  - program: 34.13 sec
  - loops: 29.7 sec



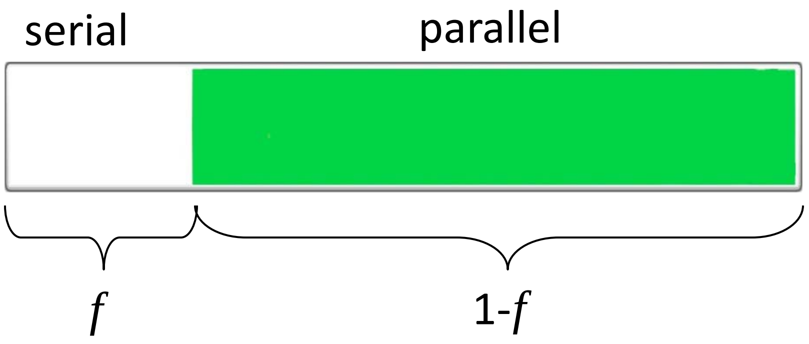
Speedup Example

- recall that in the serial program we measured
  - program: 34.13 sec
  - loops: 29.7 sec
- there is work being done outside the loops in the serial region
- time the ***whole*** program



Amdahl's Law

- Gene Amdahl: 1967
- parallelize some of the program – some must remain serial
- *f* is the fraction of the calculation that is serial
- 1-*f* is the fraction of the calculation that is parallel
- the maximum speedup that can be obtained by using P processors is:

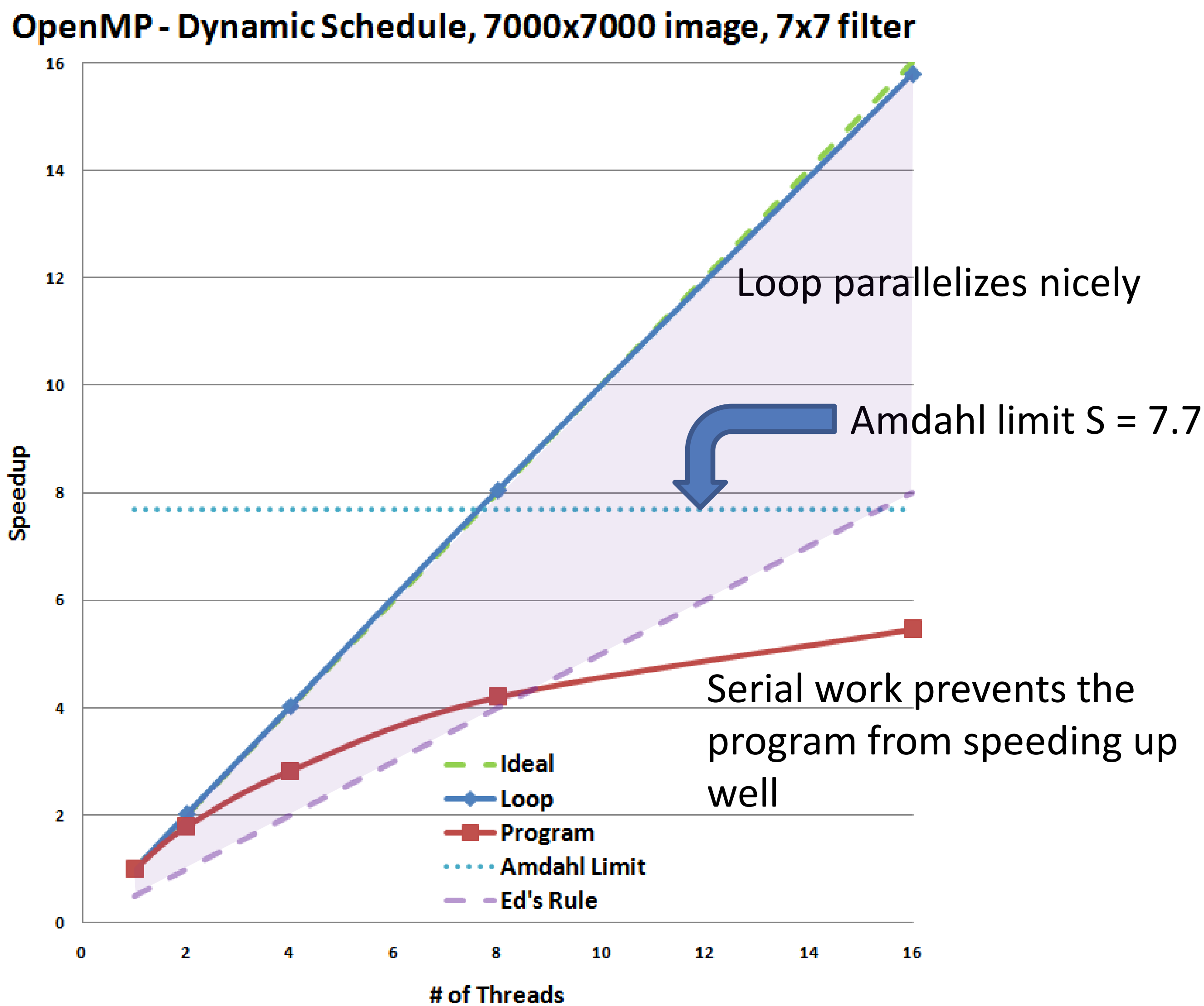


$$S_{\max} = \frac{1}{f + \frac{(1-f)}{P}}$$

Amdahl’s Law

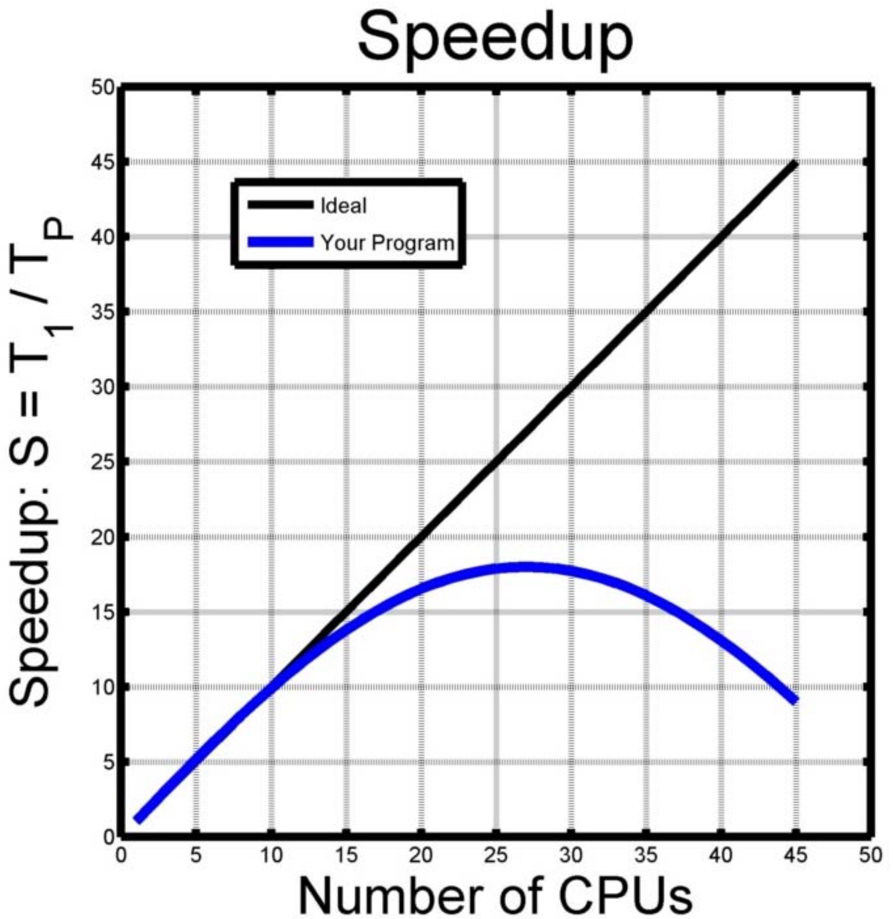
- if 25% of the calculation must remain serial the best speedup you can obtain is 4
- need to parallelize as much of the program as possible to get the best advantage from multiple processors

Speedup Example



Speedup

- why do a speedup test?
- it's hard to tell how a program will behave on many processors
- "Your Program" is actually fairly common behaviour for un-tuned code
  - in this case:
    - linear speedup to ~12 cpus
    - after 27 cpus speedup is starting to decrease
- QUESTION: how many cpus to run this program?



How to Measure Speedup?

- on Linux use the shell ***time*** function
- on Windows the C time library function ***clock()*** returns wall time
  - on Linux this gives CPU time
- Fortran: ***system_clock()*** returns wall time
- MPI: ***MPI_Wtime()***
- OpenMP: ***omp_get_wtime()***
- There Is a Way!

Summary

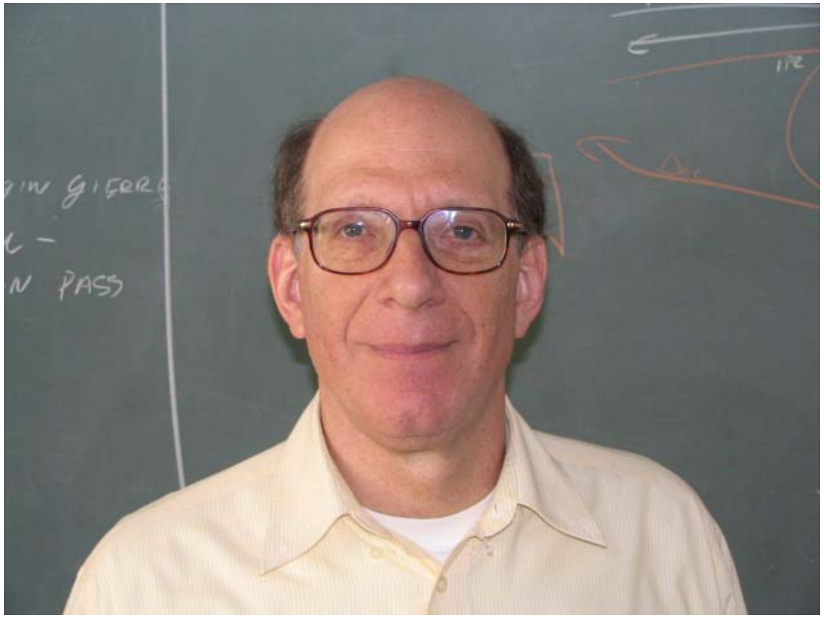
- use Linux
  - **Intro to Linux, Understanding Bash**
- write programs
  - **Linux Programming, Fortran**
- make your programs run efficiently
  - **Code Optimization**
- parallelize your programs
  - **MPI, Using a Linux Cluster**
- use other technologies
  - **Ruby, Matlab**

- you don't know how efficiently your program uses multiple processors until you do a speedup test
  - do a speedup test
- DO a speedup test!

It’s Not Easy to Make it Fast

"Sequential programming is really hard, and parallel programming is a step beyond that.“

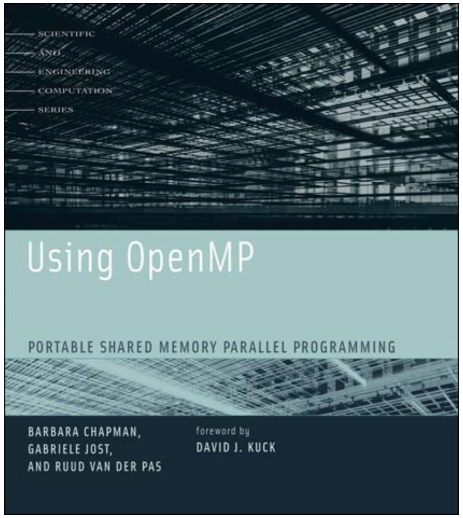
– Andrew Tanenbaum, quoted at the June 2008 Usenix conference



MINIX 3:

http://www.minix3.org/

Resources

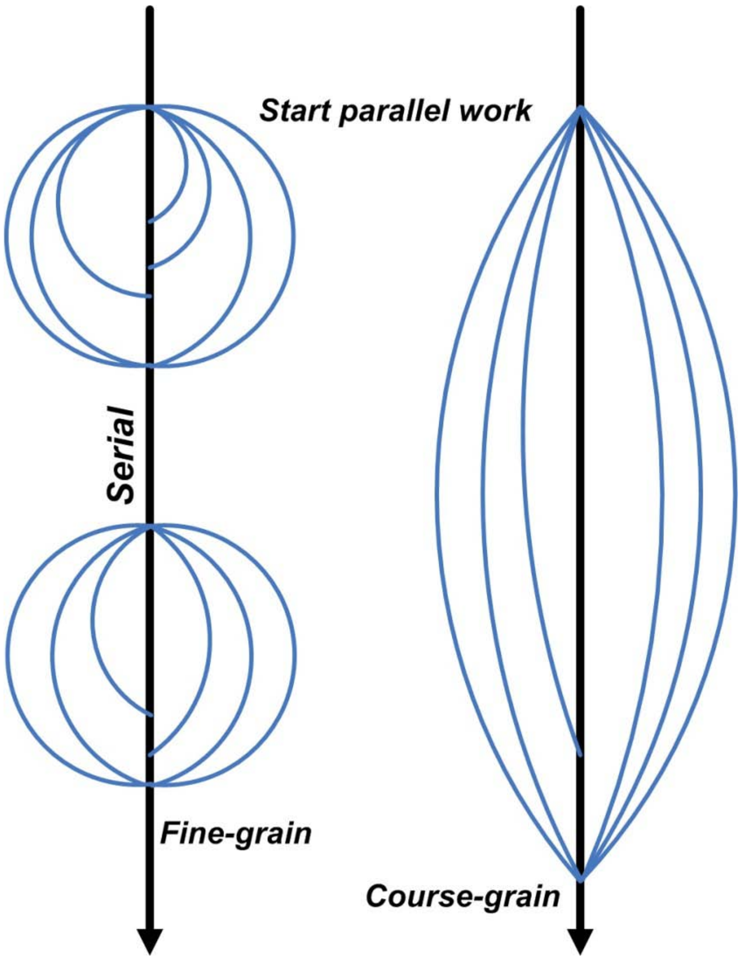


- OpenMP docs:
  http://openmp.org/wp/
- MPICH2 docs:
  http://www.mcs.anl.gov/research/projects/mpich2/
- MPI, The Complete Reference – online book
  http://www.netlib.org/utk/papers/mpi-book/mpi-book.html

Questions?

Granularity

- a qualitative measure of the ratio between computation and communication or synchronization
- fine-grain: a small amount of work is done before communication is required
- coarse-grain: a large amount of work is done before communication is required



Parallel Errors

- there are two types of errors that occur only in a parallel program:
- **Race Conditions**
  - a result depends on which thread executes a section of code first
  - this leads to unpredictable results
- **Deadlocks**
  - two threads are each waiting for a result from the other
  - no work gets done

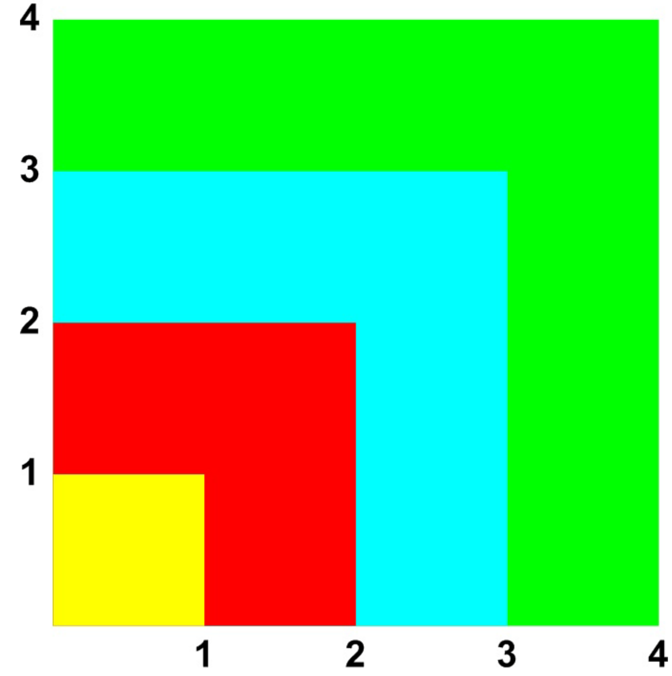
Types of Speedup

- we have been discussing **Strong Scaling**
  - *the problem size is fixed and we increase the number of processors*
    - decrease computational time (Amdahl Scaling)
  - the amount of work available to each processor decreases as the number of processors increases
  - eventually, the processors are doing more communication than number crunching and the speedup curve flattens
  - difficult to have high efficiency for large numbers of processors

- we are often interested in **Weak Scaling**
  - *double the problem size when we double the number of processors*
    - constant computational time (Gustafson scaling)
  - the amount of work for each processor stays roughly constant
  - parallel overhead is (hopefully) small compared to the real work the processor does
- weather prediction

Gustafson's Law

- keep the total time of execution fixed
- the serial part of the program is fixed
- increase the parallel work as the number of processors N increases
  - increase grid size
- work done by each thread is constant



$$S = N - f(N-1)$$

f is the serial fraction of the program
S is the speedup

Gustafson's Law