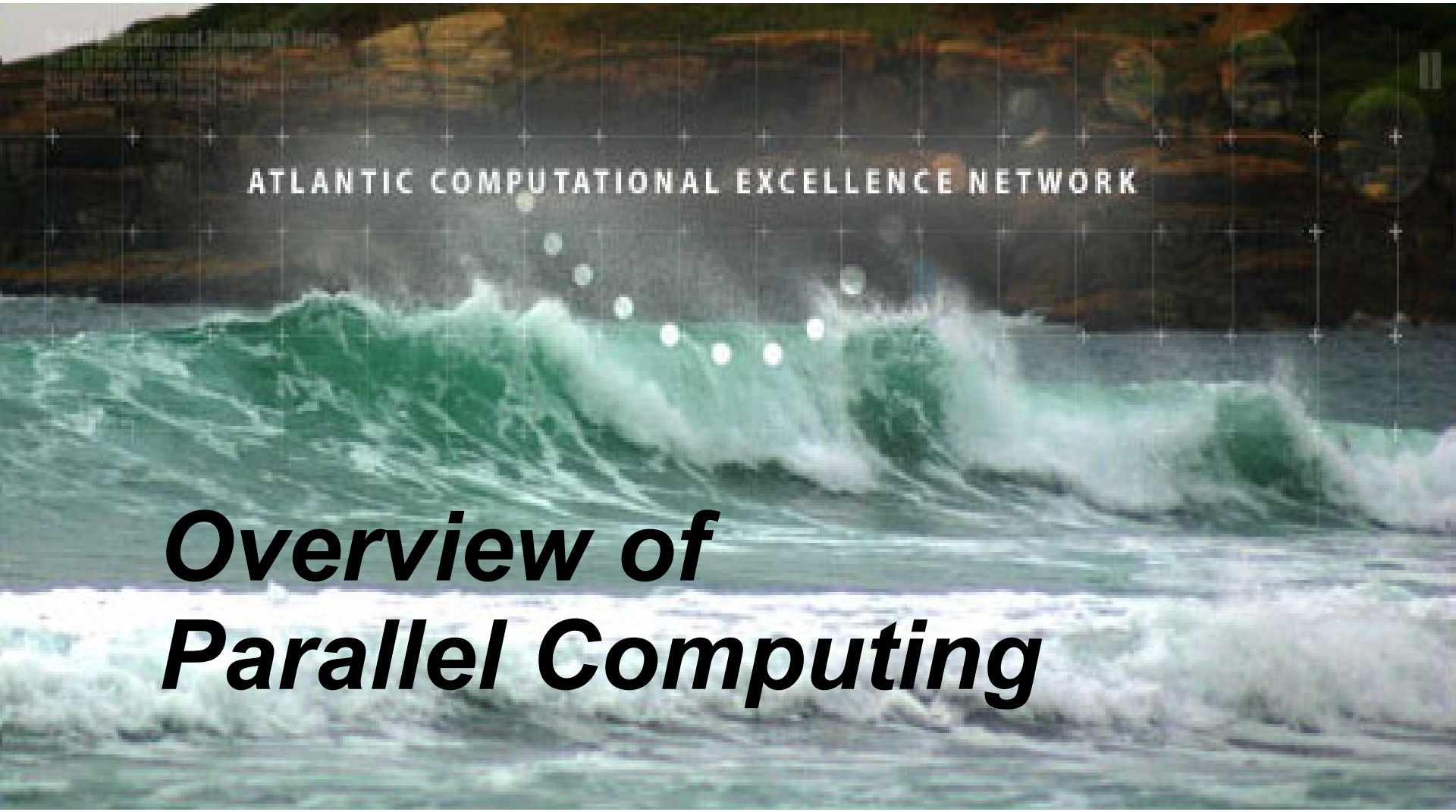


Overview of Parallel Computing

Dr. Ross Dickson, Computational Research Consultant

What can parallel computing do for me?

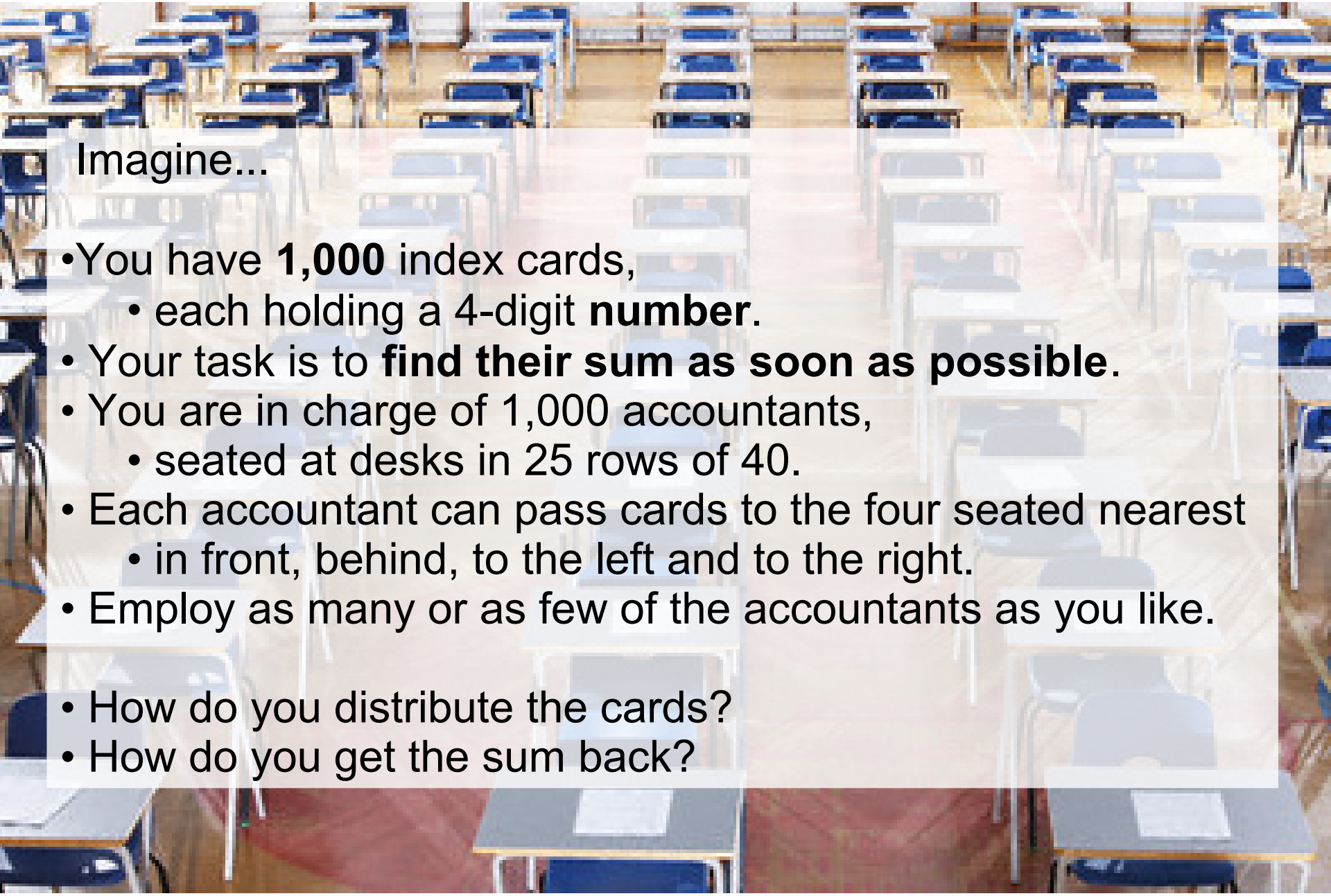
+ Faster time-to-results

+ Bigger problems feasible
  in memory
  or time

– More costly
  in equipment
  and expertise

Outline

- Parallel computer architectures
- Programming models
- MPI example
- OpenMP example
- Performance measurement
  - terminology

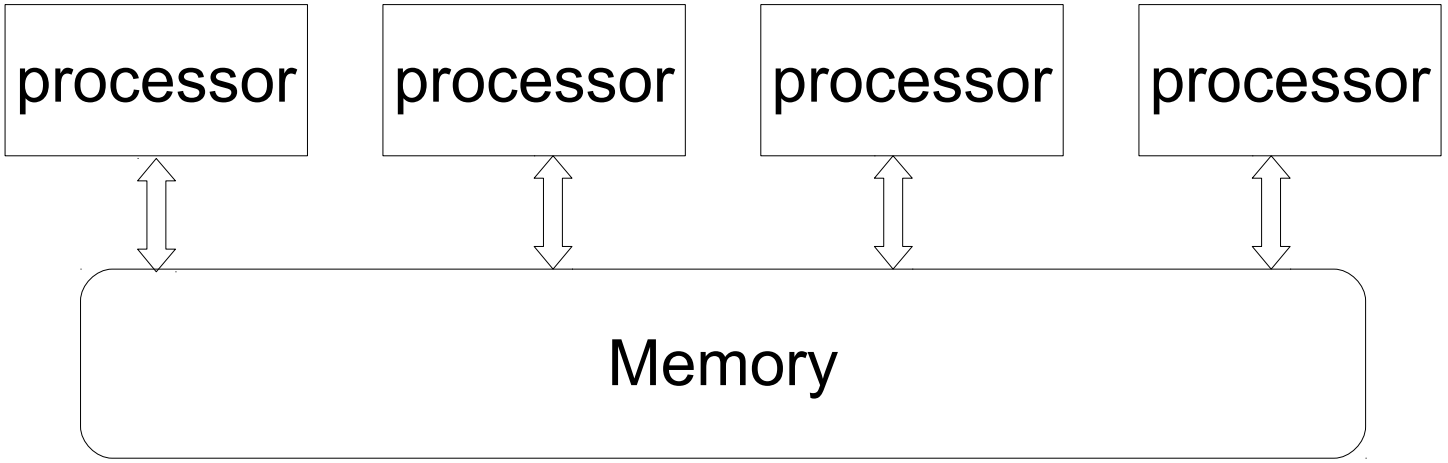


Imagine...

- You have **1,000** index cards,
  - each holding a 4-digit **number**.
- Your task is to **find their sum as soon as possible**.
- You are in charge of 1,000 accountants,
  - seated at desks in 25 rows of 40.
- Each accountant can pass cards to the four seated nearest
  - in front, behind, to the left and to the right.
- Employ as many or as few of the accountants as you like.

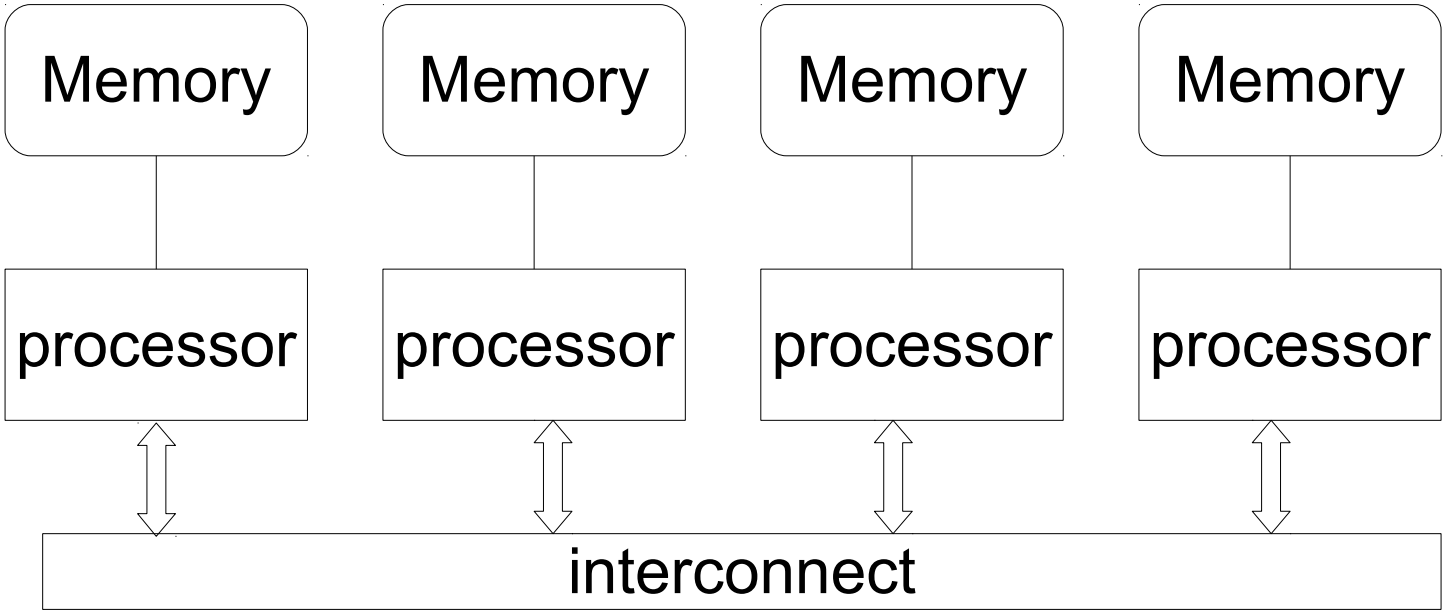
- How do you distribute the cards?
- How do you get the sum back?

Parallel architecture: Shared-memory



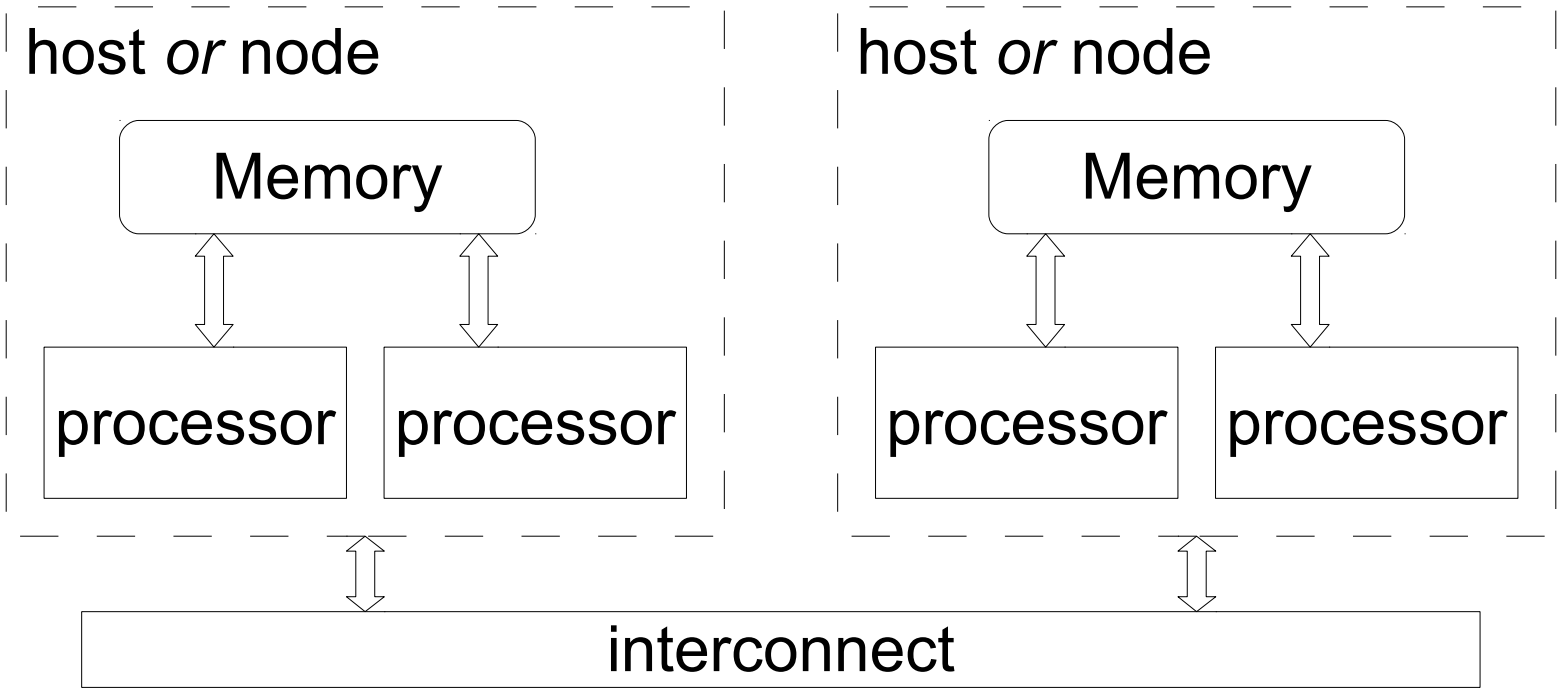
- also called “multiprocessors”
- *e.g.* modern dual-core or quad-core CPUs
- “cc-NUMA” a common implementation
  - *cache-consistent Non-Uniform Memory Access*
- “SMP” = “symmetric multiprocessor”
  - *often used loosely for “shared memory processor”*

Parallel architecture: Distributed-memory



- interconnect can be Ethernet...
- ...or something faster and more expensive like Infiniband or Myrinet

Parallel architecture: Hybrid



- Contemporary clusters usually built on this model

Programming Model 1: Independent tasks

- Near 100% efficiency
- Very simple to implement
  - handled at job-manager level
  - no changes to serial application code
- "Embarrassingly parallel"
  - so-called because it is of no theoretical interest
  - better term: "perfectly parallel"
- Similar terms:
  - "parametric parallelism"
  - "High-throughput computing"

Programming Model 2: Shared-memory

- Each process has
  - read/write access to a **shared** memory area
  - a **private** memory area (stack)
- Maps well to shared-memory computers
  - practical limit: size of a single machine
- Communication between processes is via shared memory
  - potential for **race conditions**
- Processes sometimes called “threads” or LWPs (light-weight processes)
- OpenMP and Posix Threads two popular implementations

OpenMP example – saxpy.f90


```
program saxpy
integer, parameter :: n=10
real a,x(n),y(n)
integer i
read *,a,x(1:n),y(1:n)



do i=1,n
   y(i) = a*x(i) + y(i)
end do



print *,y(1:n)
end program
```

OpenMP example – saxpy.f90

```
program saxpy
integer, parameter :: n=10
real a,x(n),y(n)
integer i
read *,a,x(1:n),y(1:n)

!$OMP PARALLEL PRIVATE(i) SHARED(a,x,y)
!$OMP DO
do i=1,n
   y(i) = a*x(i) + y(i)
end do
!$OMP END DO
!$OMP END PARALLEL

print *,y(1:n)
end program
```

OpenMP operation

```
$ pgf90 -mp saxpy.f90 -o saxpy
$ export OMP_NUM_THREADS=2
$ ./saxpy <input
  18.0000   16.0000  14.0000   12.0000
  10.0000    8.0000   6.0000    4.0000
   2.0000    0.0000
$
```

More about OpenMP programming from Joey Bernard here on Thursday May 29.

Programming Model 3: Message-passing

- Each process has its own independent memory
  - Maps well to distributed-memory computers
- Communication between processes is explicit
  - Application has to be reprogrammed...
  - ...maybe even redesigned
- Analogous to how teams of people work
  - *"You and I don't share memory, we pass messages" – N. Ostlund*
- Scales to arbitrary number of processors
  - ...though perhaps not efficiently!
- MPI most popular standard for message-passing programming

MPI example: send-recv.c

```
#include <mpi.h>
#include <stdio.h>
int main( int argc, char *argv[] )
{  int myRank, msg=12345, source=0, destination=1, tag=99;
   MPI_Status status;

   MPI_Init(&argc,&argv);
   MPI_Comm_rank(MPI_COMM_WORLD,&myRank);

   if (myRank == source) {
         MPI_Send(&msg, 1, MPI_INT, destination, tag, MPI_COMM_WORLD);
         printf ("Process %d sent %d to process %d.\n", myRank, msg, destination);

   } else if (myRank == destination) {
         MPI_Recv(&msg, 1, MPI_INT, source, tag, MPI_COMM_WORLD, &status);
         printf ("Process %d received %d from proc %d.\n", myRank, msg, source);
   }

   MPI_Finalize();
}
```

MPI operation

```
$ mpicc send-recv.c -o send-recv
$ mpirun -np 2 send-recv
Process 0 sent 12345 to process 1.
Process 1 received 12345 from proc 0.
$
```

More about MPI programming from Ross Dickson here on Tuesday May 27.

Another option:
Autoparallelizing compilers

- Some compilers will attempt to parallelize code in OpenMP style but without OpenMP directives, if proper flag supplied
  - SunStudio: **`-autopar`**
  - PGI: **`-Mconcur`**
  - Intel: **`-parallel`**
- Works on individual loops
  - like OpenMP
  - **`$OMP_NUM_THREADS`** controls thread count

Performance: Speedup

$$\text{Speedup} = \frac{\text{Sequential execution time}}{\text{Parallel execution time}}$$

Example:

- 6 hours to run on 1 processor (sequential)
- 1 hour to run on *p* processors (parallel)
- → 6h/1h = speedup of 6

Michael J. Quinn, *Parallel Programming in C with MPI and OpenMP* (McGraw-Hill, 2004). ISBN 0-07-282256-2

Performance: Speedup

$$\psi \equiv \text{Speedup} = \frac{\text{Sequential execution time}}{\text{Parallel execution time}}$$

Consider

- the time spent in sequential work, σ
- the time spent in parallelizable work, φ
- parallel overhead costs, κ

...for a problem of size n running on p processors:

$$\psi(n, p) \leq \frac{\sigma(n) + \phi(n)}{\sigma(n) + \phi(n)/p + \kappa(n, p)}$$

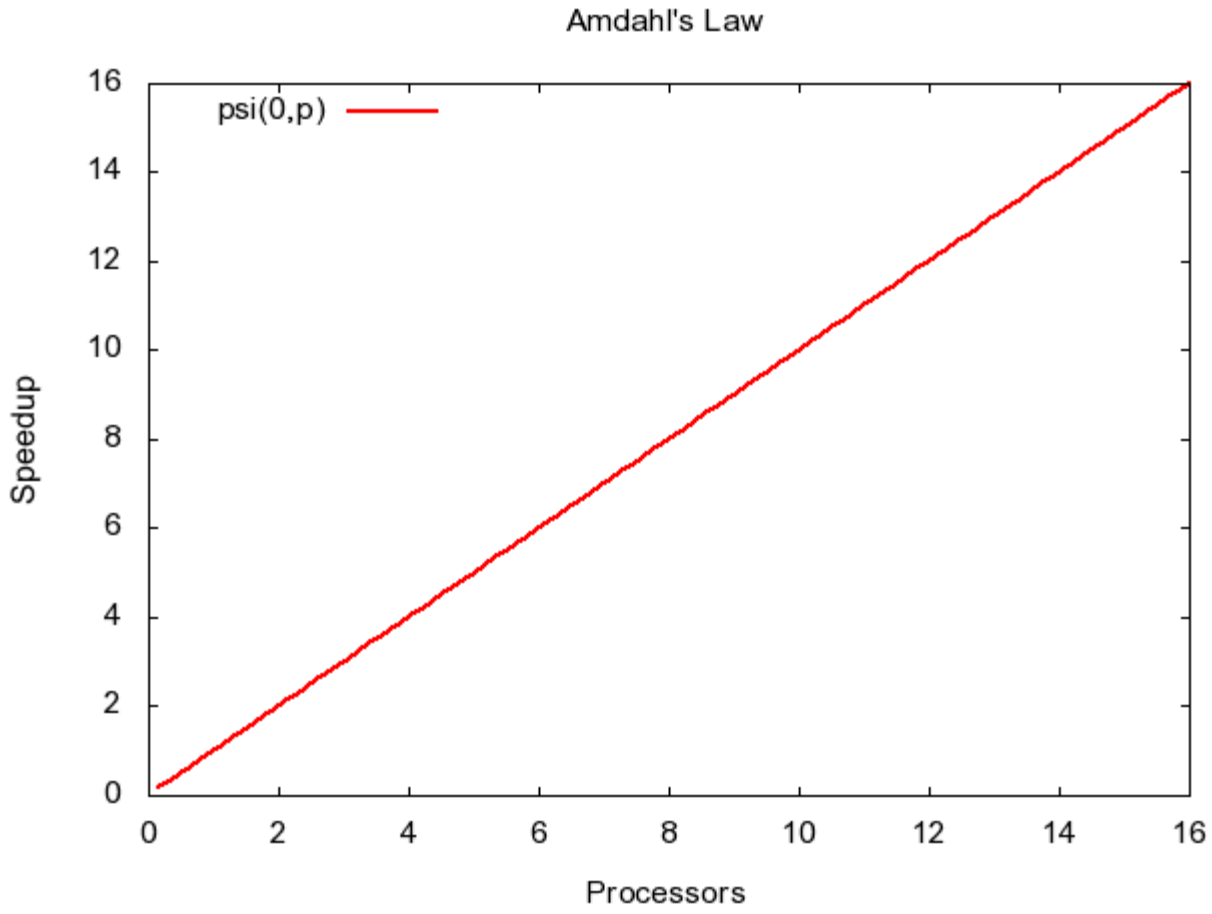
How much speedup can you get?

- Assume parallel overhead κ is negligible (Optimist!)
  - Define serial fraction $f_s = \sigma/(\sigma+\varphi)$

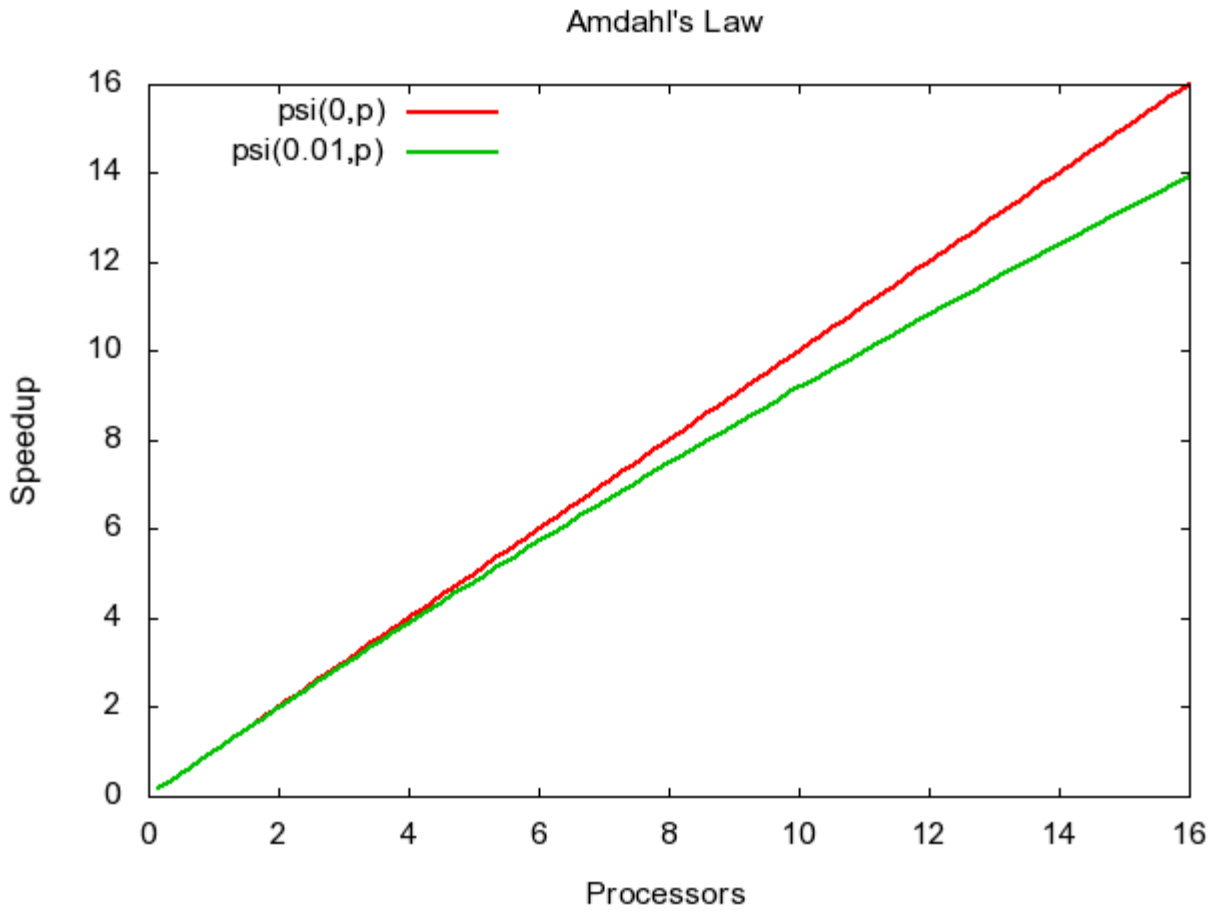
$$\psi(f_s, p) \leq \frac{1}{f_s + (1 - f_s)/p}$$

Amdahl's Law

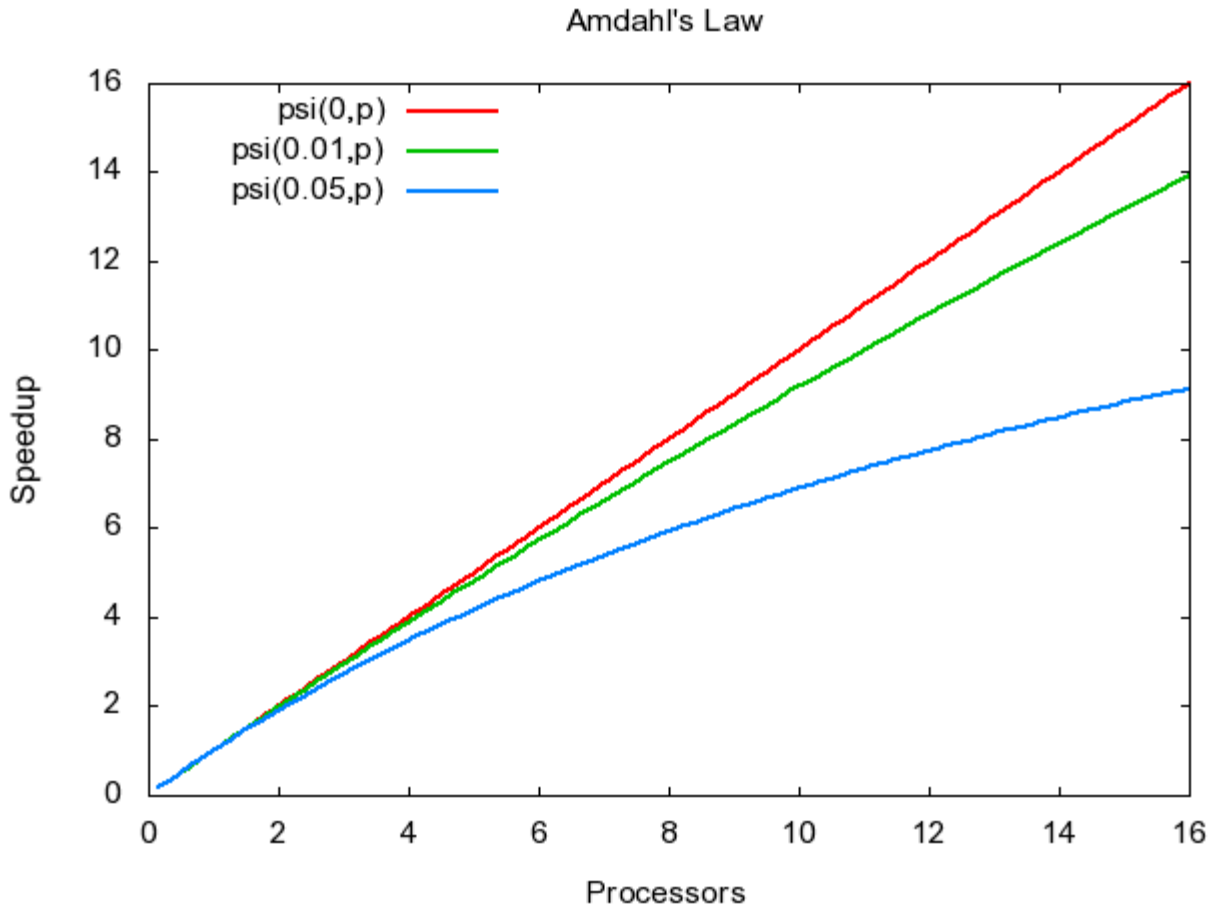
How much speedup can you get?



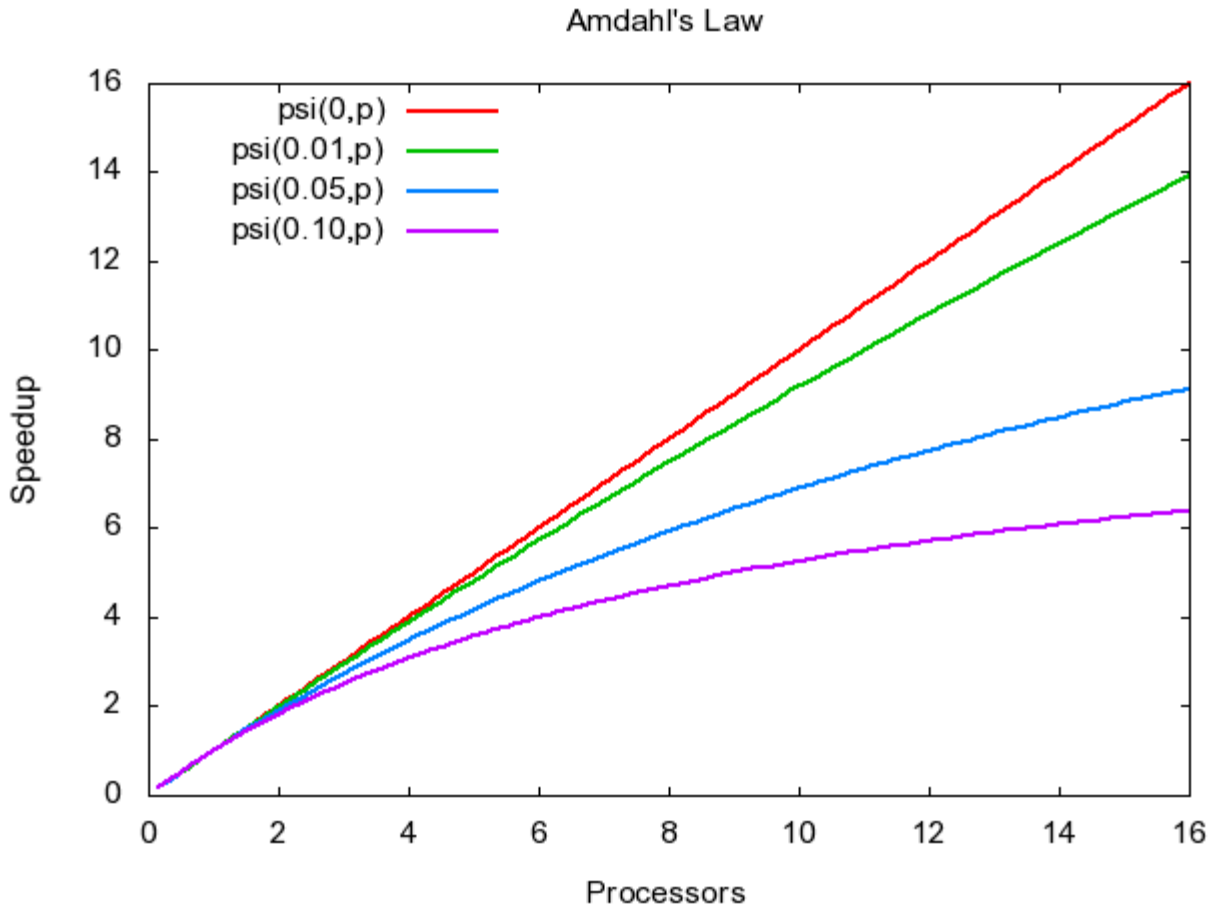
How much speedup can you get?



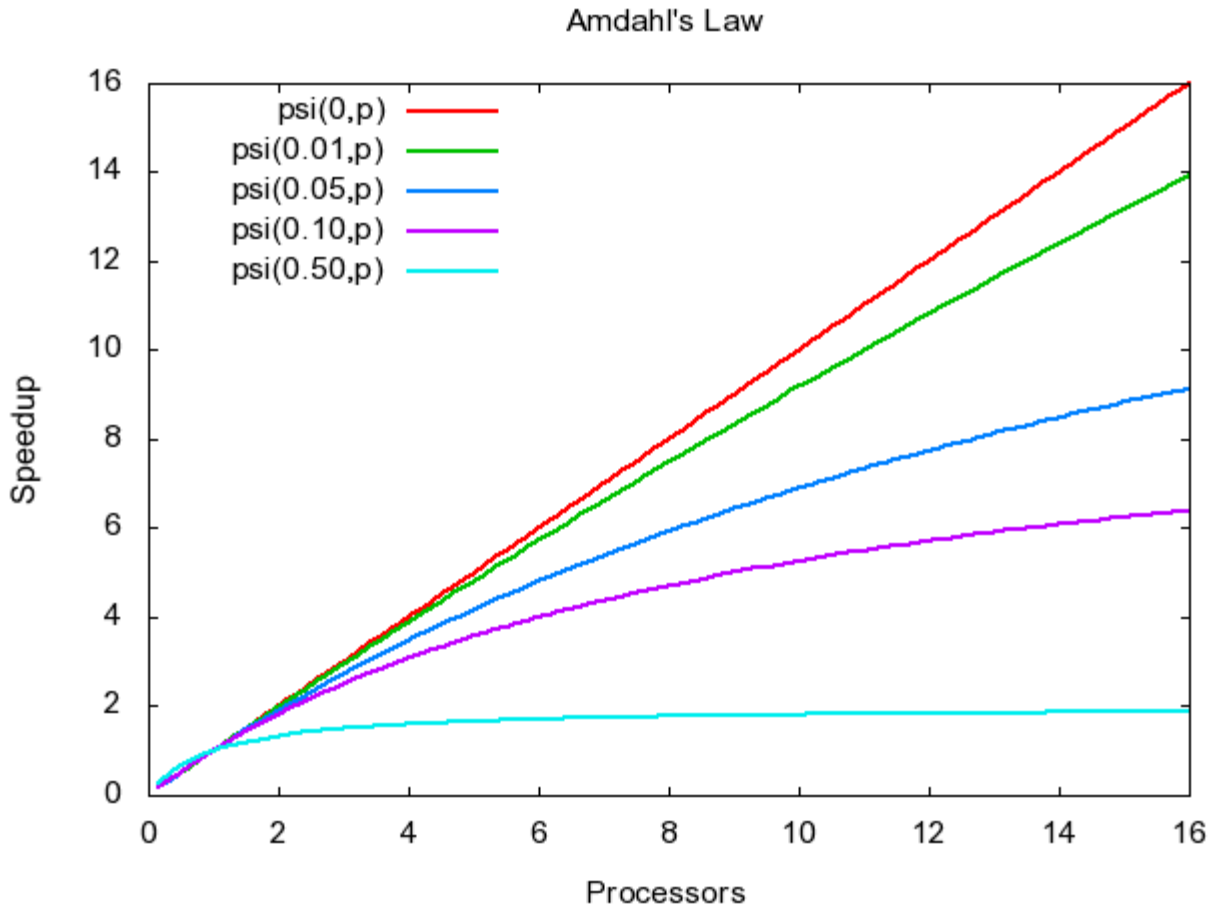
How much speedup can you get?



How much speedup can you get?



How much speedup can you get?



Efficiency

$$\text{Efficiency} = \frac{\text{Speedup}}{\text{Processors}}$$

$$= \frac{\text{Sequential time}}{\text{Processors} \times \text{Parallel time}}$$

- a fraction between 0 and 100%
- depends on number of processors!
- varies with size of problem

Scalability

- A measure of the ability to increase performance as the number of processors increases

- As the problem size increases,
  - so (usually) does the parallel fraction,
  - and hence the speedup,
  - and hence efficiency.

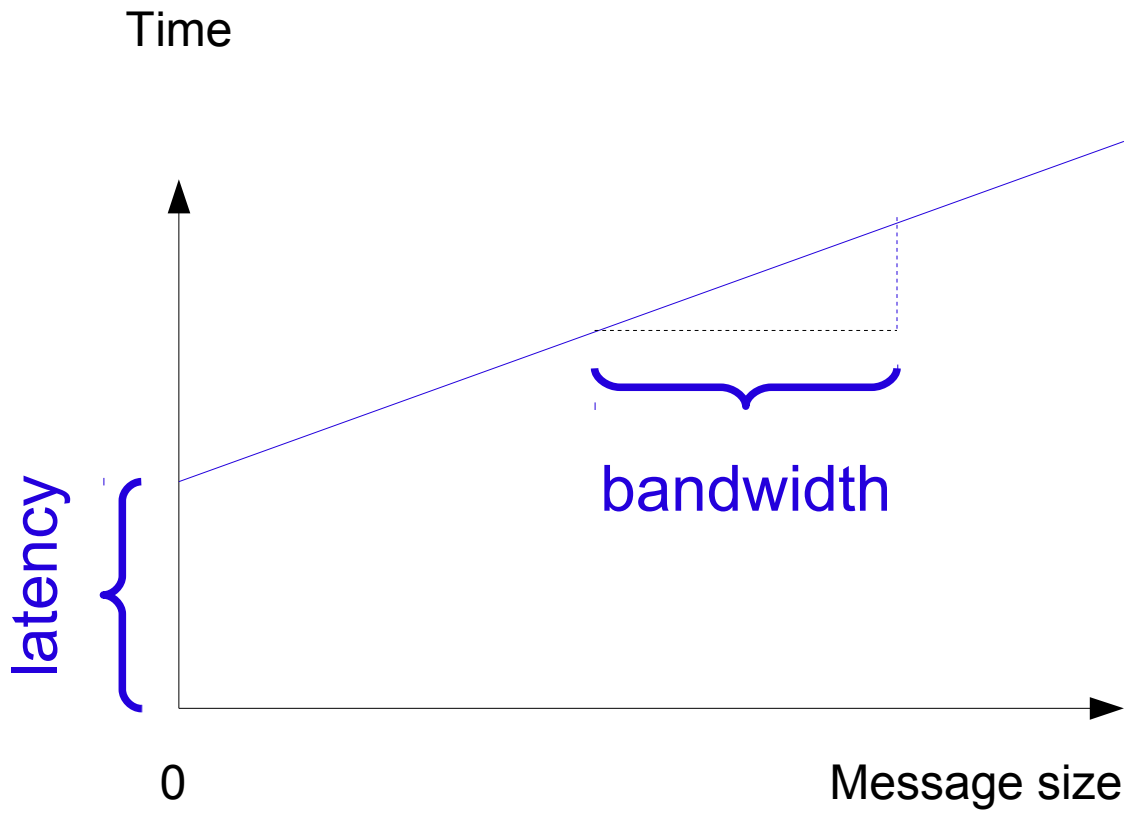
Performance: Speedup

$$\psi(n, p) \leq \frac{\sigma(n) + \phi(n)}{\sigma(n) + \phi(n)/p + \kappa(n, p)}$$

- The time spent in sequential work, σ, tends to stay near-constant with problem size
- The time spent in parallelizable work, φ, usually *grows* with problem size
- Parallel overhead cost, κ,
  - also tends to grow with problem size
  - depends critically on the algorithm
  - and on communication speed (the interconnect)

Communication costs

- Passing messages takes finite time
  - So does coordinating the use of shared memory



$$t(n) = \lambda + n/\beta$$

Other resources

- Books
  - Michael J Quinn, *Parallel Programming in C with MPI and OpenMP* (ISBN 0-07-282256-2)
  - ...many others !...
- Websites

  http://www.mcs.anl.gov/research/projects/mpi/
  http://openmp.org/wp/
- ACEnet User Wiki
  - http://www.ace-net.ca/wiki/ACEnet
  - email: support@ace-net.ca