Introduction to Parallel Computing

May, 2015

computecanada
regional partner

What Can Parallel Computing Do?

Pluses

- Faster time to results
- Bigger problems feasible in memory or time

Minus

- More costly in equipment and expertise

Outline

Parallel computer architectures

Programming models

MPI example

OpenMP example

Performance measurement

Terminology

Monitoring Progress

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

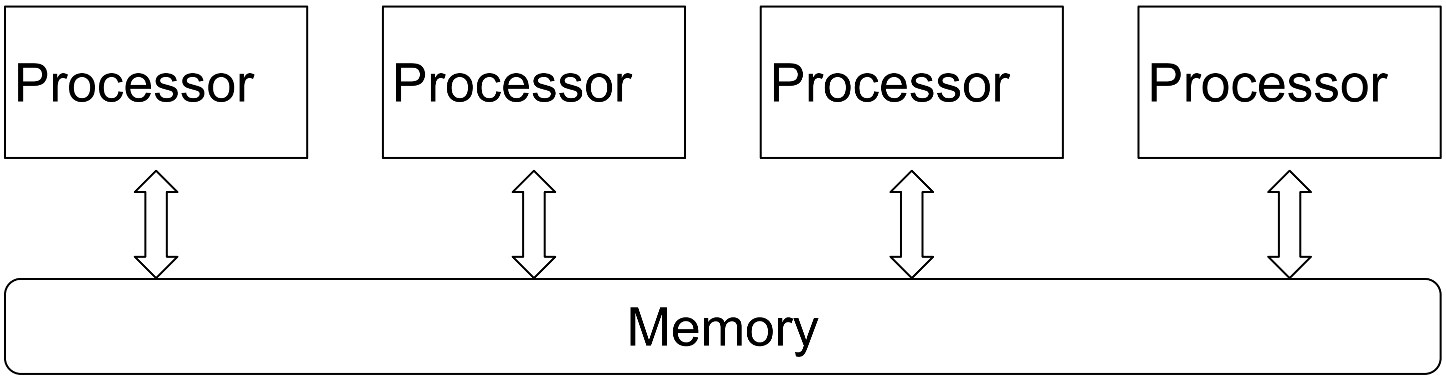
in front, behind, to the left and to the right.

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

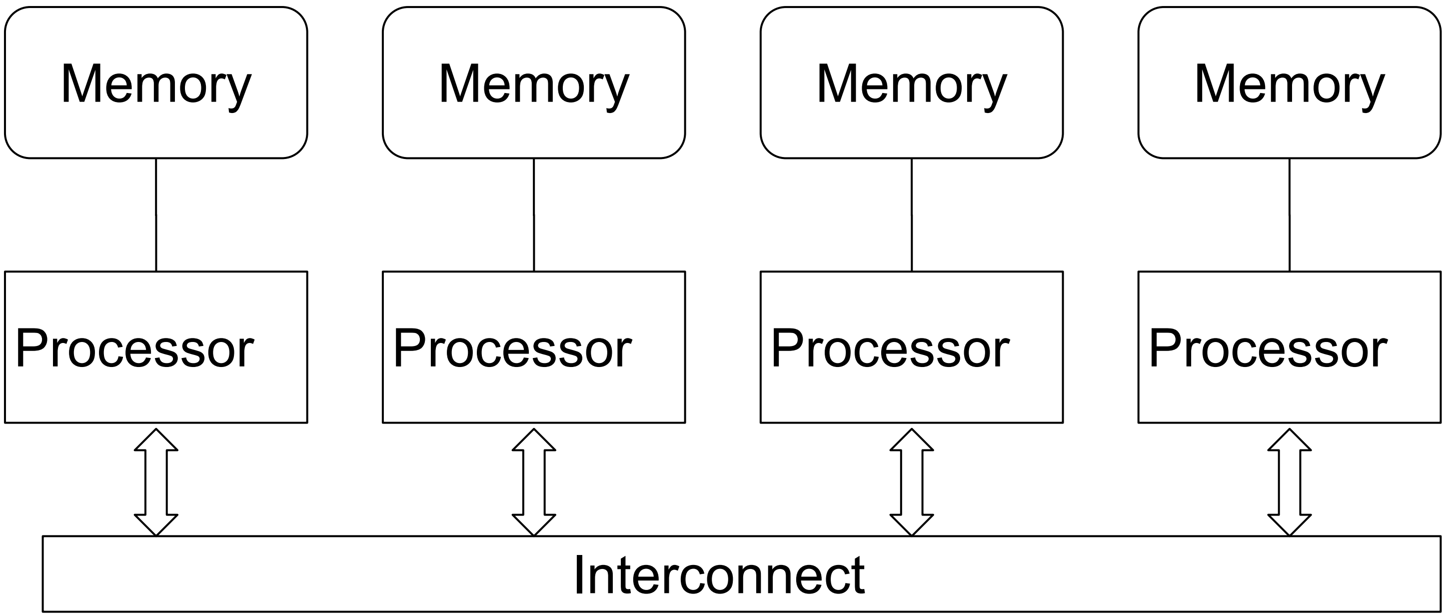
Shared Memory



- Also called “multiprocessors”, *e.g.* dual-core or quad-core CPUs or even 16-core, 22-core on modern 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

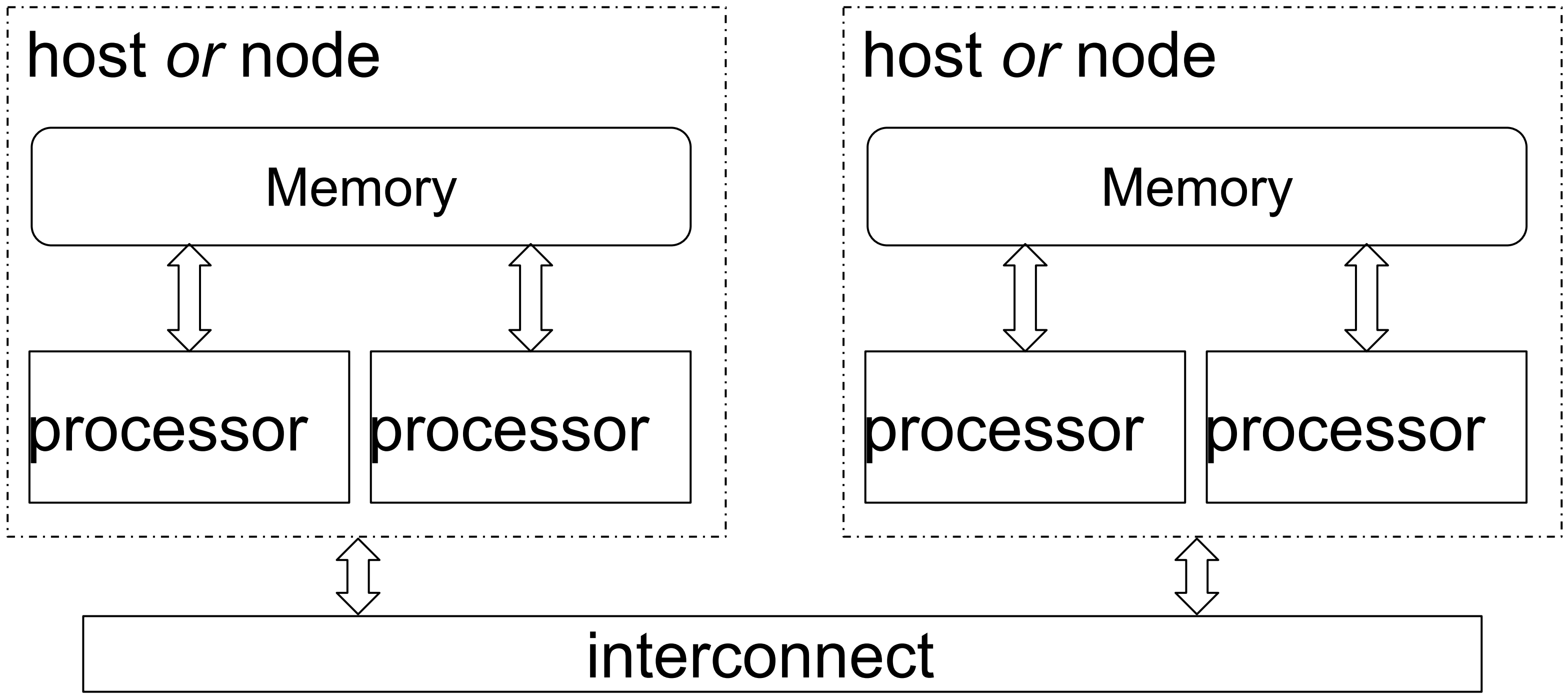


Interconnect can be Ethernet...

...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
```

- loop with known number of iterations
- each iteration is independent from the others

saxpy ≙ single precision "ax+y"

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
```

saxpy ≙ single precision "ax+y"

OpenMP Operation

```
$ cat input
1
9 8 7 6 5 4 3 2 1 0
9 8 7 6 5 4 3 2 1 0

$ 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
$
```

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.

$
```

Another Option

Auto-Parallelizing Compilers

Some compilers will try to parallelize code in OpenMP style but without the programmer supplying OpenMP directives, if told to:

- PGI: **`-Mconcur`**
- Intel: **`-parallel`**

Like OpenMP,

- works on individual loops
- **`$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:

$$\psi(n,p) \leq \sigma(n) + \varphi(n) / \sigma(n) + \varphi(n)/p + \kappa(n,p)$$

How Much Speedup Can You Get?

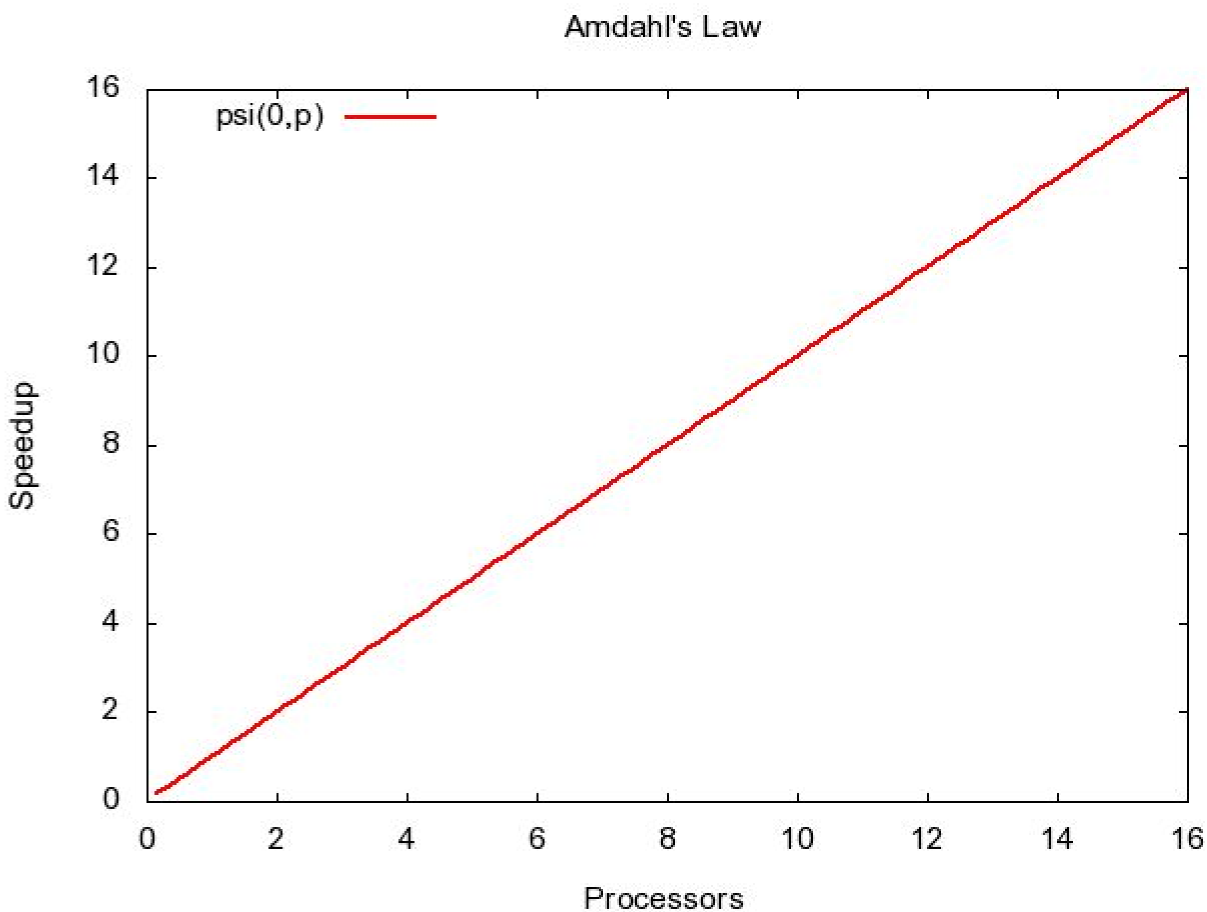
Assume parallel overhead κ is negligible (Optimist!)

Define serial fraction $f_s = \sigma/(\sigma+\varphi)$

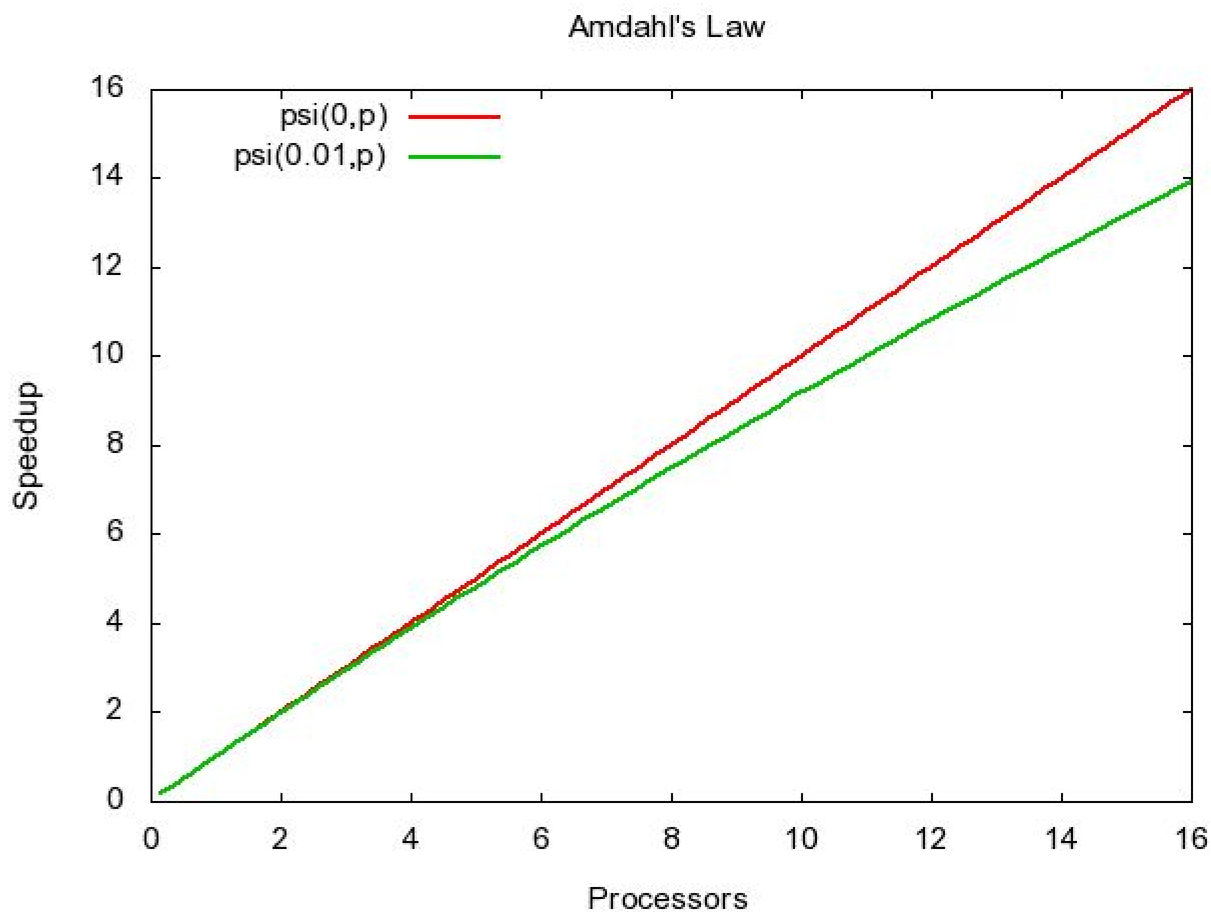
$$\psi(f_s, p) \leq 1/f_s + (1 - f_s)/p$$

Amdahl's Law

How Much Speedup Can You Get?



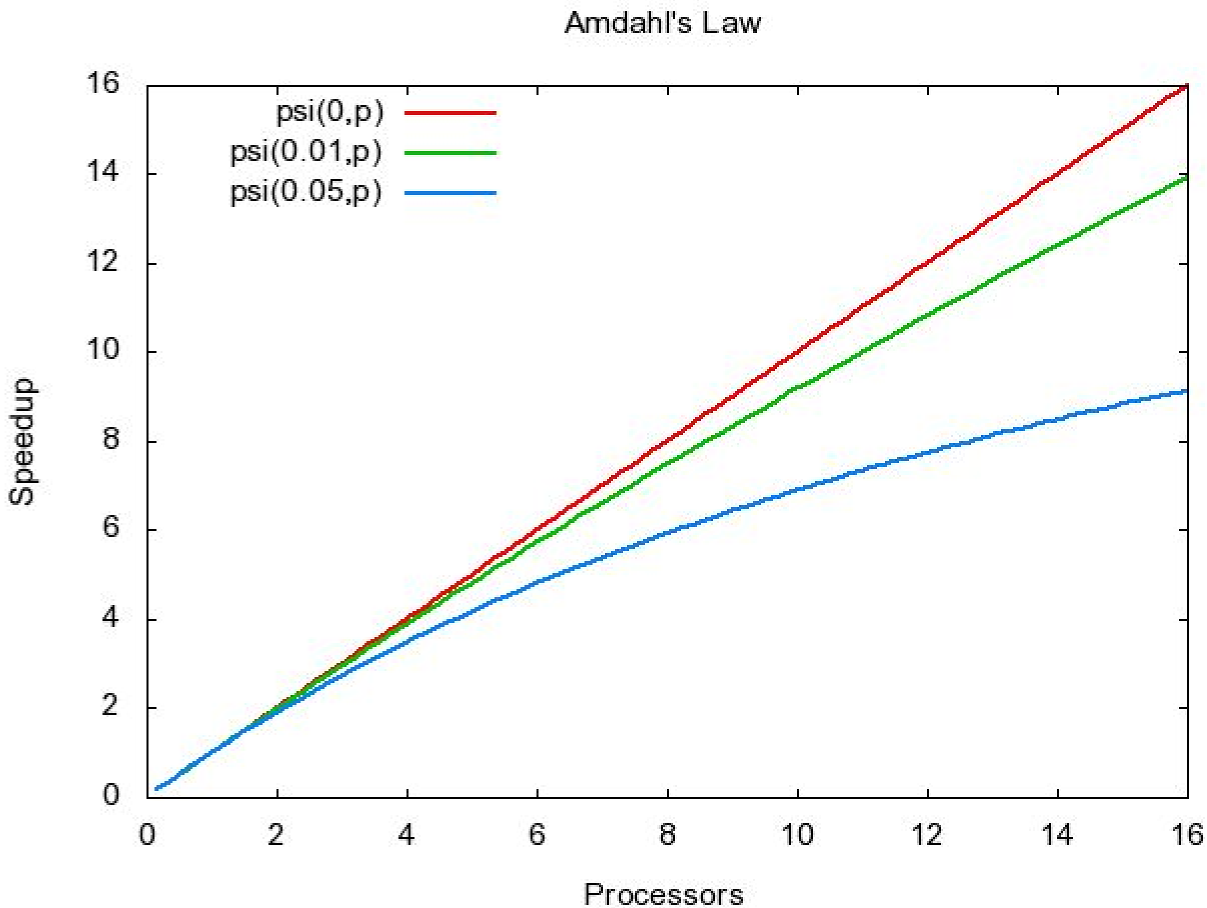
How Much Speedup Can You Get?



ACENET accelerate discovery

compute canada regional partner

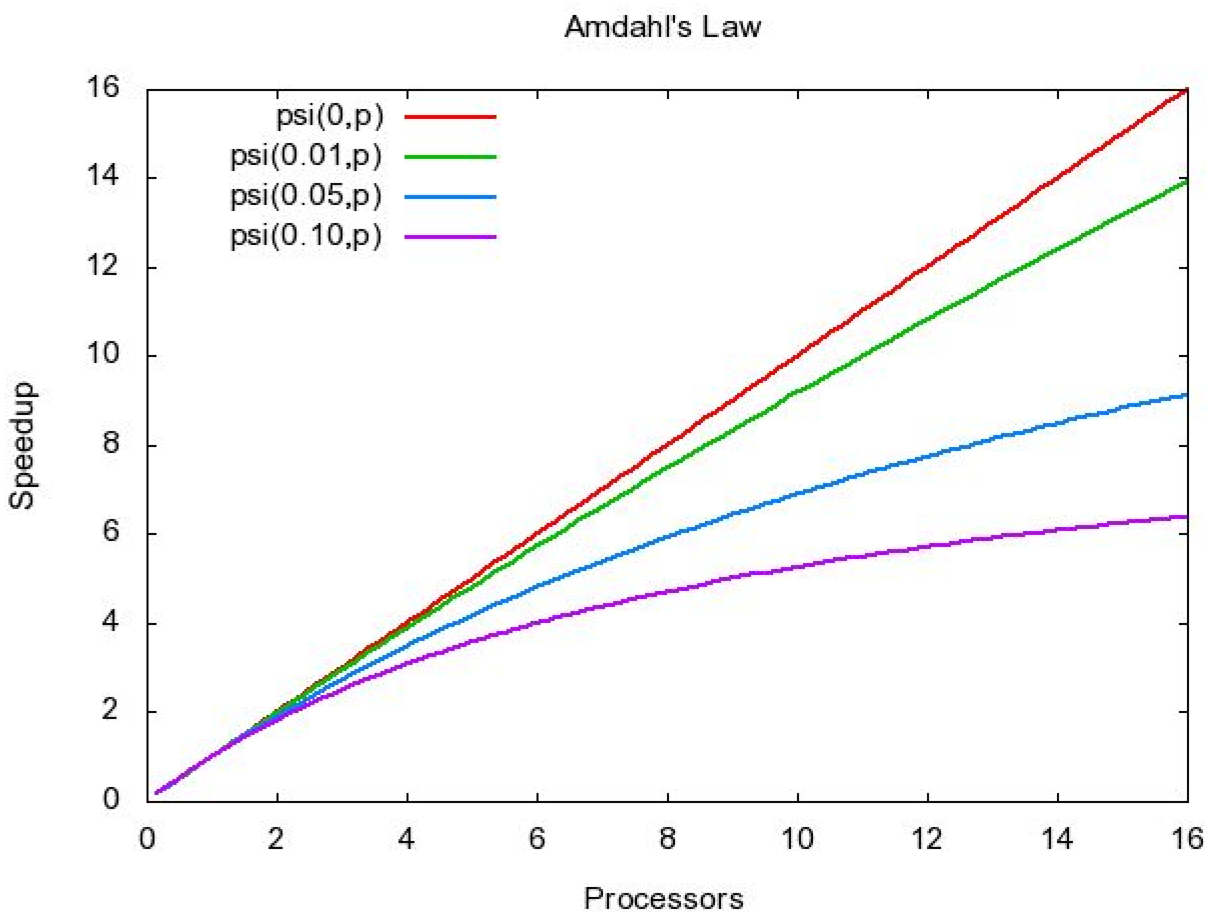
How Much Speedup Can You Get?



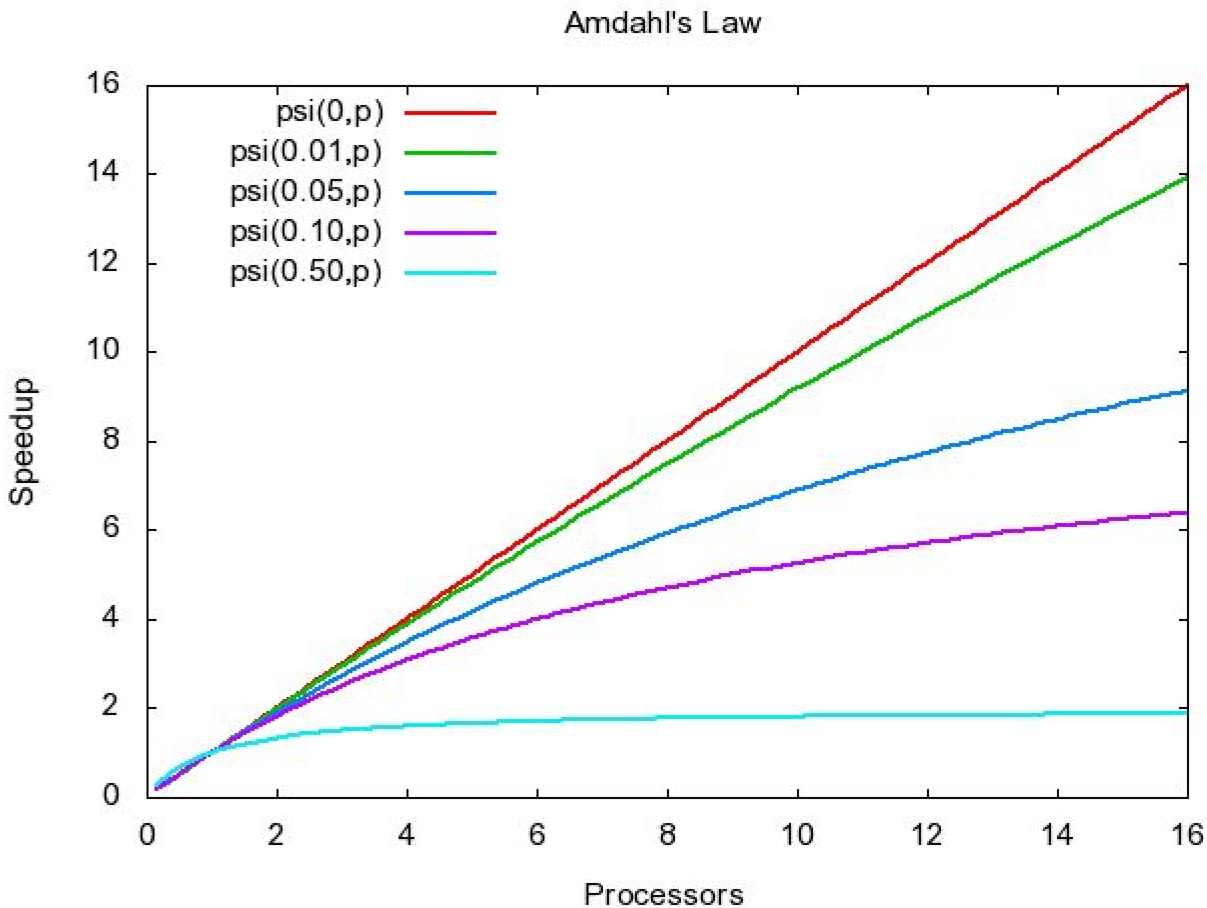
ACENET
accelerate discovery

compute canada
regional partner

How Much Speedup Can You Get?



How Much Speedup Can You Get?



ACENET
accelerate discovery

compute canada
regional partner

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

$$\psi(n,p) \leq \sigma(n) + \varphi(n) / \sigma(n) + \varphi(n)/p + \kappa(n,p)$$

The time spent in sequential work, σ, tends to stay near-constant with problem size.

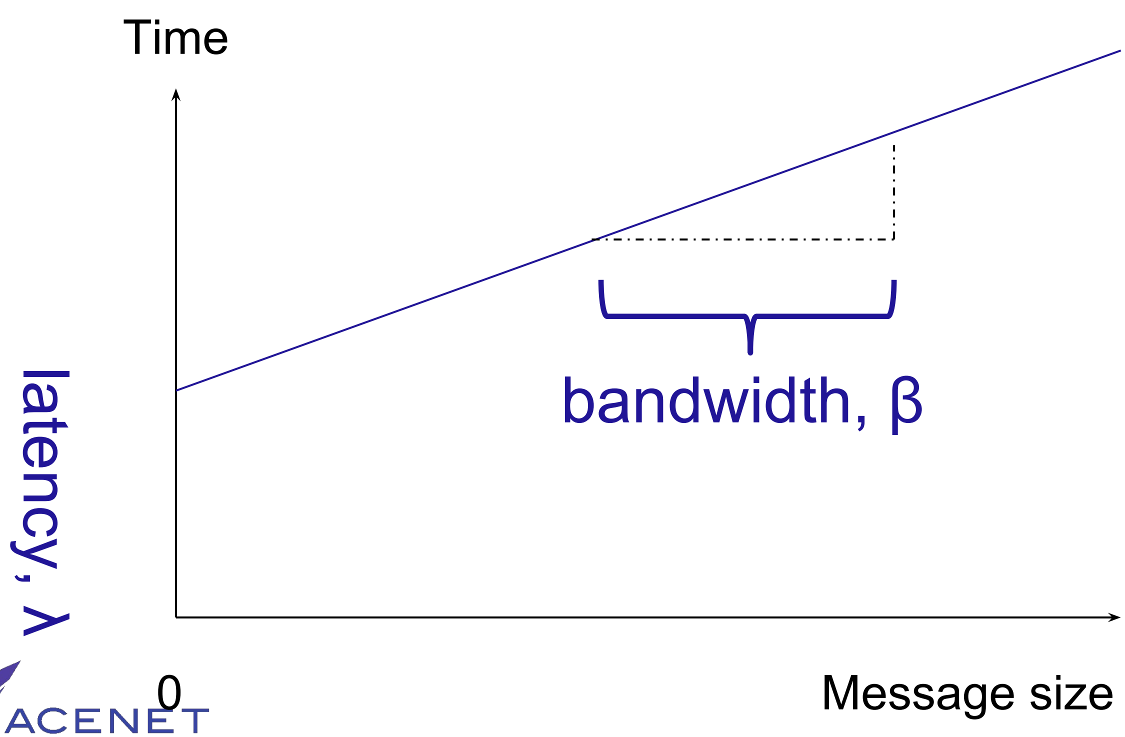
The time spent in parallelizable work, φ, usually *grows* with problem size.

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.)



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

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.acceleratediscovery.ca/wiki/ACENET

Email support@ace-net.ca