A Software View

- Parallel Programming Languages and Environments
- Parallel Programming and Problem Solving

©2004 P.Y. Wang

Parallel Programming Languages and Environments

- What do they they look like? How should they look?
- How much work can a compiler do?
- Should languages/programs be portable?
- Should languages be extensions of existing serial languages?
- What is the role of the underlying parallel architecture?
- What kind of software environments exist?

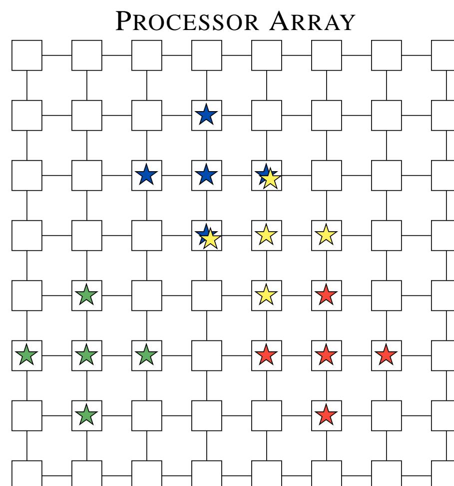
©2004 P.Y. Wang

Some Sample Parallel Programs

- Data-parallel neighborhood averaging programs
  - MPP Pascal
  - CM *Lisp
- MIMD Master-worker PVM code

Neighborhood Averaging Computation

PROCESSOR ARRAY



$$\frac{1}{5}(x_{i,j} + x_{i-1,j} + x_{i+1,j} + x_{i,j-1} + x_{i,j+1})$$

MPP Pascal

```
program average(input,output);
type mx = parallel array [1 . . . .128, 1 . . . 128] of real;
type bx = parallel array [1 . . . 128, 1 . . . 128] of boolean;
var M : mx;
    INNER : bx;
begin
      (* set INNER to be true for non-border data *)
      INNER := true;
      INNER[1, ] := false;    INNER[128, ] := false;
      INNER[ ,1] := false;    INNER[ ,128] := false;
      (* set M as the average of its neighbors if not on the border of the matrix *)
      where INNER do
             M := 0.2 * ( M + shift(M,0,1) + shift(M,0,-1)
                                    + shift(M,1,0) + shift(M,-1,0) )
```


Connection Machine 200 CM *Lisp

```
(*cold-boot :initial-dimemnsions '(128 128))

(*defvar M)

;;; define a parallel function that returns true for
;;; non-border elements in the grid and false otherwise

(*defun inner!! ()
  (if!!   (or!!    ( zerop!!    (self-address-grid!! (!! 0)) )
                   ( =!!        (self-address-grid!! (!! 0)) (!! 127))
                   ( zerop!!    (self-address-grid!! (!! 1)) )
                   ( =!!        (self-address-grid!! (!! 1)) (!! 127)) )
          nil!!
          t!!))
```

```
;;; define M as the average of its neighbors for non-border elements

(*defun average!! ()
    (*when (inner!!)
        (*set M (*!! (!! 0.2)
                (+!! M
                    (pref-grid-relative!! M (!! 0) (!! 1)  :border-pvar (!! 0)))
                    (pref-grid-relative!! M (!! 0) (!! -1) :border-pvar (!! 0)))
                    (pref-grid-relative!! M (!! 1) (!! 0)  :border-pvar (!! 0)))
                    (pref-grid-relative!! M (!! -1) (!! 0) :border-pvar (!! 0)) )))))
```

©2004 P.Y. Wang

MIMD Master-Worker in PVM

The **master.c** Program

```
#include <stdio.h>
#include "pvm3.h"
#define WORKER "workers"

main() {
   int mytid;             /* my task id */
   int tids[32];          /* worker task ids */
   int n, nproc, numt, i, who, msgtype, nhost, narch;
   float data[100], result[32];
   struct pvmhostinfo *hostp[32];

   /* enroll in pvm */
      mytid = pvm_mytid();

   /* Set number of workers */
   /* Note: cannot do stdin from a spawned task */
      if ( pvm_parent() == PvmNoParent ) {
          puts ("How many workers (1-32)?");
          scanf ("%d", &nproc);
      }
```

©2004 P.Y. Wang

```
/* Start up worker tasks */

   numt = pvm_spawn(WORKER, (char**)0, 0, "", nproc, tids);

   if (numt < nproc ) {
      printf ("Trouble spawning workers. Aborting.\n"");
      printf ("Error codes are:\n"");
      for (i=numt; i< nproc; i++) printf ("TID %d %d\n",i,tids[i]);
      for (i=0; i< numt; i++) pvm_kill(tids[i]);
      pvm_exit();
      exit();
   }
```



```
/* Begin user program - initialize data array*/

   n=100;
   for (i=0; i<n; i++) data[i]=1;

/* Broadcast initial data array to all workers */

   pvm_initsend(PvmDataDefault);
   pvm_pkint(&nproc,1,1);
   pvm_pkint(tids,nproc,1);
   pvm_pkint(&n,1,1);
   pvm_pkfloat(data,n,1);
   pvm_mcast(tids,nproc,0);
```

```
/* Wait for results from workers */
   msgtype = 5;
   pvm_recv(-1,msgtype);
   pvm_upkint(&who,1,1);
   pvm_upkfloat(&result[who],1,1);
   printf ("I got %f from worker %d\n",result[who],who);

/* Program finished; exit gracefully */
   pvm_exit;
```


The **workers.c** Program

```
# include <stdio.h>
# include "pvm.h"

main () {
   int mytid;       /* my task id */
   int tids[32];    /* all task ids */
   int n, me, i, nproc, master, msgtype;
   float data[100], result;
   float work();

   /* enroll in PVM */
      mytid = pvm_mytid();

   /* Receive data from master */
      msgtype=0;
      pvm_recv(-1,msgtype);
      pvm_unpkint(&nproc,1,1);
      pvm_unpkint(tids,nproc,1);
      pvm_unpkint(&n,1,1);
      pvm_unpkfloat(data,n,1);
```

```
   /* Determine which worker I am */
      for (i=0;i<nproc;i++)
         if (mytid == tids[i]) {me = i; break;}

   /* Call my subroutine to do calculations */
      result = work (me,n,data,tids,nproc);

   /* Send my result to master */
      pvm_initsend(PvmDataDefault);
      pvm_pkint(&me,1,1);
      pvm_pkfloat(&result,1,1);
      msgtype = 5;
      master=pvm_parent();
      pvm_send(master,,msgtype);

   /* End gracefully */
      pvm_exit();
}
```



```
float work(me,n,data,tids,nproc)
   int me, n *tids,nproc;
   float *data;
{
   int i, dest;
   float psum, sum;
   psum=0.0; sum = me*1.0;

   /* Swap data with left worker (wrapping) */
      pvm_initsend(PvmDataDefault);
      pvm_pkfloat(&sum,1,1);
      dest = me+1;
      if (dest == nproc) dest = 0;
      pvm_send(tids[dest],22);
      pvm_recv(-1,22);
      pvm_upkfloat(&psum,1,1);

   return (sum+psum);
}
```

Parallel Programming and Problem Solving

- How do we write parallel programs/algorithms?
- How do we implement data structures in parallel?
- How do we measure the performance of parallel algorithms?

©2004 P.Y. Wang

Approaches for Designing Parallel Algorithms

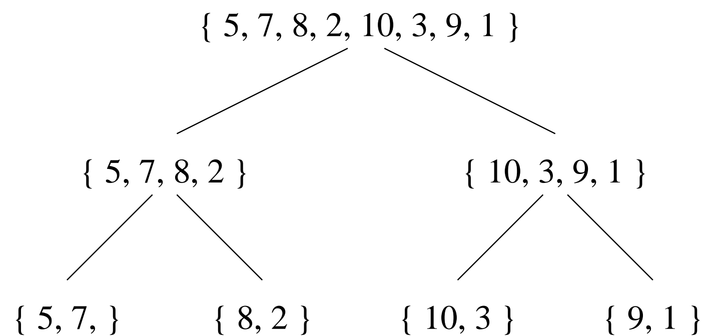
- Take a sequential algorithm and parallelize it: may work well for algorithms based on divide-and-conquer schemes
- Create a new algorithm targeted for solving the problem on a specific parallel architecture
- Design an algorithm using a theoretical model

©2004 P.Y. Wang

Problem: Finding the Maximum(S)

- The divide and conquer paradigm can be applied:
  - Split S into two subsets
  - Find the maximum of each of the two subsets
  - Compute the maximum of the two maximums
- Use recursion to find the maximum of the two subsets

©2004 P.Y. Wang



©2004 P.Y. Wang

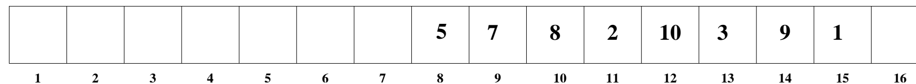
Using a PRAM

Put the $n = 2^m$ items in the global memory of a PRAM at locations A[n], A[n+1], ... A[2n-1]

for $k \leftarrow m - 1$ downto 0 do

forall $P_j, 2^k \leq j < 2^{k+1}$ do in parallel

$A[j] \leftarrow \max\{A[2j], A[2j+1]\}$



Time complexity = $\Theta(\lg n)$

Alternative Approach

forall P_i do in parallel

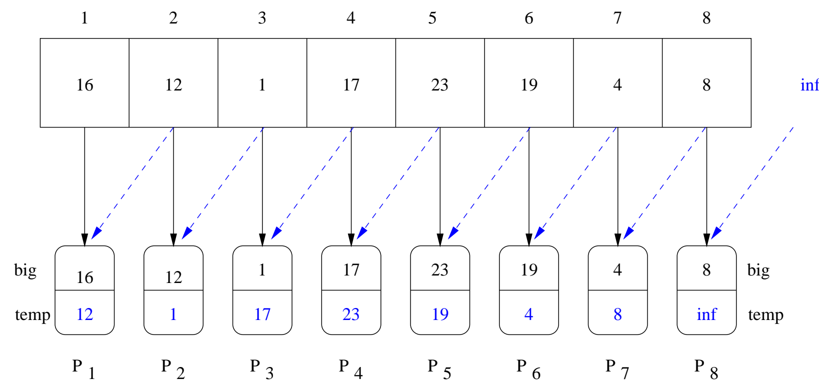
big ← A[i] /*Each processor sets its big value */

incr ← 1

for step ← 1 to $\lg n$ do

temp ← A[i+incr] /* Each processor sets it temp value */
big ← max {big, temp} /* and does a local comparison */
incr ← 2 * incr
A[i] ← big /* storing bigger value into global memory */

Example



Finding the Maximum on a Hypercube

Distribute the $n = 2^m$ items onto an $m-$dimensional hypercube, one item per node

for $k \leftarrow 0$ to $m - 1$ do

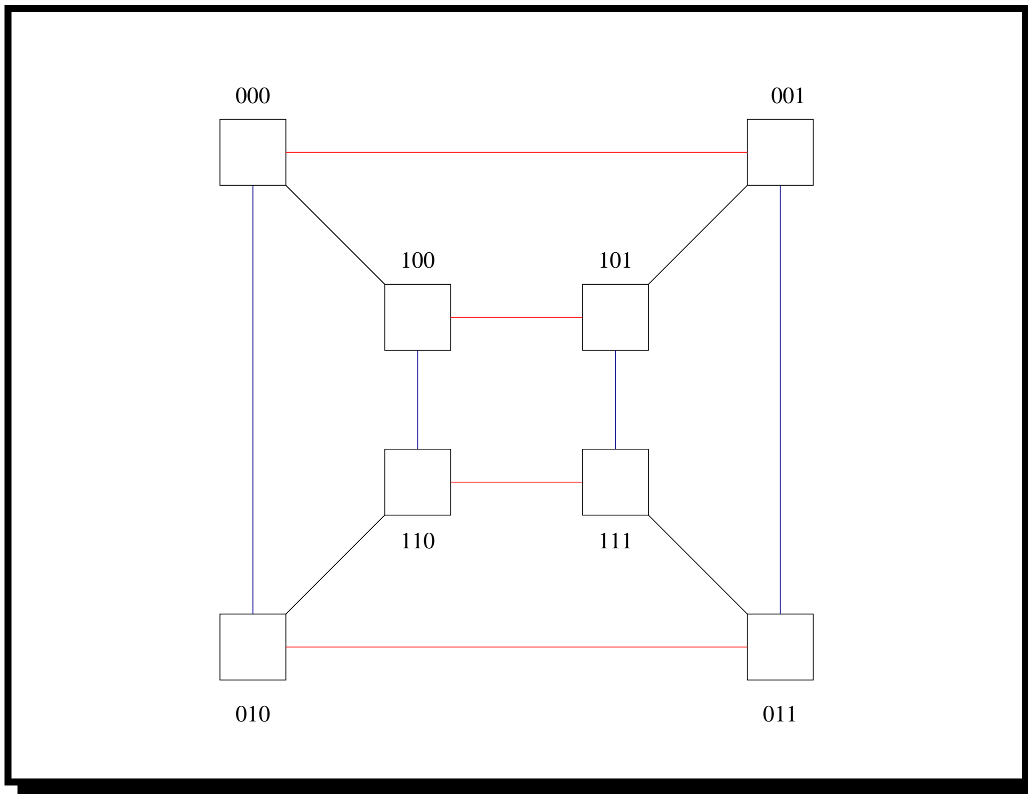
Use the $cube_k$ function to exchange values between nodes

Store the larger value as the maximum

endfor

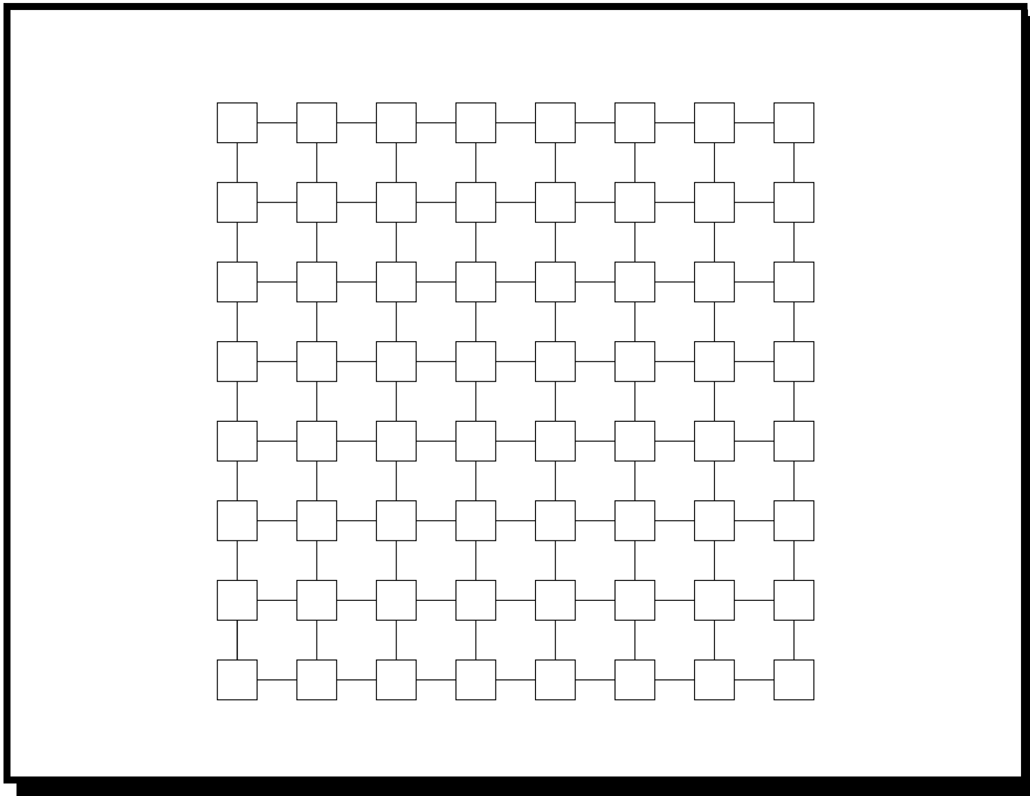
Time complexity =

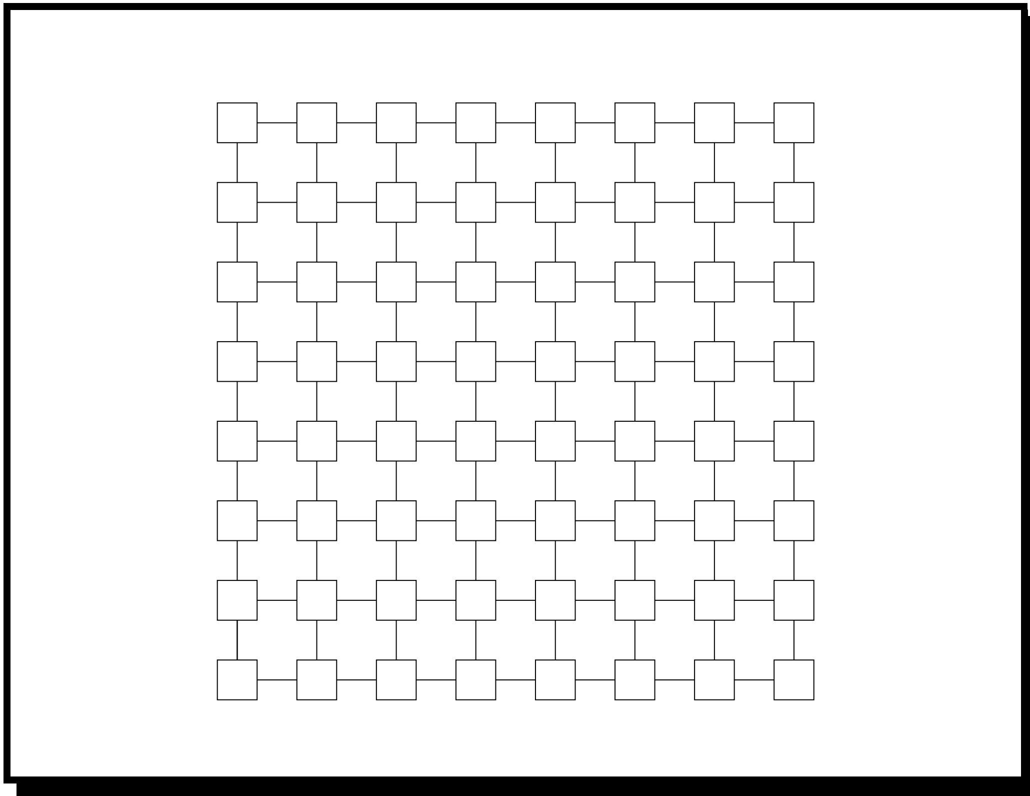
$\Theta(\lg n \cdot \{\text{Time to route data + time to compute local max }\})$



Problem: Replicate the First Row of an $n \times n$ Array

- Approach I:
  - for $k \leftarrow 0$ to $n - 1$ do: move the row k south
- Approach II:
  - Use recursive doubling





Sequential Complexity Theory

- Terminology
  - **P** is the class of problems solvable in polynomial time
  - **NP** is the class of problems solvable in nondeterministic polynomial time
  - **NP-Complete** is the class of problems for which any one being solvable in polynomial time implies they are all solvable in polynomial time
- Question: **Is P = NP?**

Parallel Complexity Theory

- Parallel Computation Thesis

  The class of problems solvable in polynomial time on a PRAM is P-SPACE

  where **P-SPACE** are the problems solvable by a sequential machine in polynomial space
- Many problems in **P** (like sorting) can be solve on a PRAM in polylog parallel time: $(\log n)^{O(1)}$ where n is the problem size

- Some problems in **P** appear unlikely to admit a solution in polylog parallel time
- They belong to a class called **log-space complete for P**
  - **NC** is the class of problems solvable in polylog parallel time on a polynomial number of processors
  - **SC** is the class of problems solvable in polynomial sequential time and polylog space
- Question: **Is NC = SC?**

©2004 P.Y. Wang

Performance Measures and Algorithm Analysis

- Performance Metrics
- Granularity of the Problem
- Scalability
- Sources of Overhead
- Minimizing Execution Time
- Other Measures

©2004 P.Y. Wang

Common Performance Metrics

Suppose we want to measure the performance of a p processor parallel architecure for solving a particular problem. We define

$$\text{Speedup} \doteq S = \frac{\text{Time of the best serial algorithm}}{T_p = \text{Time of parallel algorithm}}$$

$$\text{Efficiency} \doteq E = \frac{S}{p}$$

$$\text{Work or Cost} = T_p \times p$$

Ideally, we want speedup $S = p$ and Efficiency $E = 1$, but this rarely happens.

©2004 P.Y. Wang

An Example

Compute the sum of n numbers on a n node hypercube

$$\begin{aligned} T_p(n) &= c \lg n = \Theta(\lg n) \\ T_s(n) &= d(n-1) = \Theta(n) \\ S &= \Theta(\frac{n}{\lg n}) \\ E &= \Theta(\frac{1}{\lg n}) \\ Cost &= \Theta(n \lg n) \end{aligned}$$

This is not cost-optimal: The $Cost$ exceeds T_s !!

©2004 P.Y. Wang

Granularity

- How do we modify the approach to be cost-optimal?
- Reduce the number of processors
- But how??

©2004 P.Y. Wang

Mapping n Items Onto p Processors Where $n >> p$

Suppose each processor now handles n/p data items

- First approach:
  - repeat the original algorithm n/p times
  - that is, virtualize the system
  - Time = $\Theta((n/p) \lg p)$
- Second approach:
  - Each processor first adds up its n/p items; then execute the original algorithm
  - Time = $\Theta((n/p) + \lg p)$
- Note the second approach is cost-optimal if $n = \Omega(p \lg p)$

©2004 P.Y. Wang

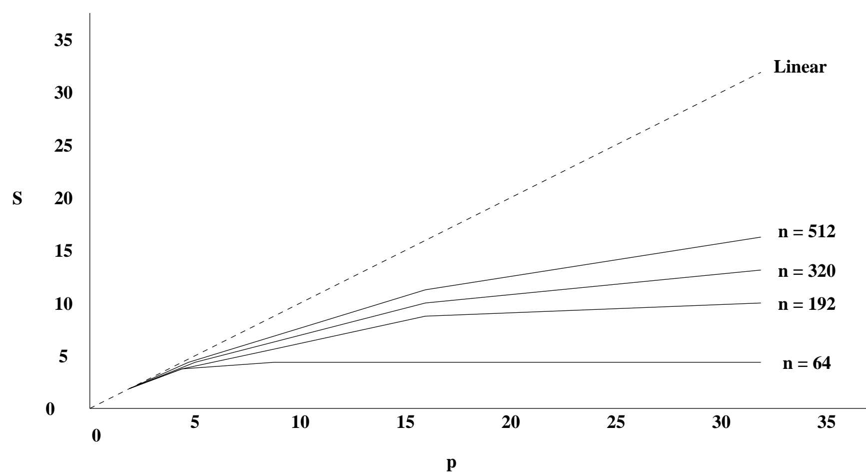
Scalability Measures

Suppose it takes one time unit to add two numbers and also to send a number between two adjacent nodes on a hypercube. Then

$$\begin{aligned} T_p &= local_add\ \ time + \lg p(communication + add\ \ time) \\ &= (\frac{n}{p} - 1) + 2\lg p \\ T_s &= n - 1 \approx n \\ S &= \frac{np}{n + 2p\lg p} \\ E &= \frac{n}{n + 2p\lg p} \end{aligned}$$

Speedup Graph

Speedup tends to flatten as $p \to \infty$



Efficiency Table

n	$p=1$	$p=4$	$p=8$	$p=16$	$p=32$
64	1	.80	.57	.33	.17
192	1	.92	.80	.60	.38
320	1	.95	.87	.71	.50
512	1	.97	.91	.80	.62

Table 1: Efficiency as a Function of n and p

Efficiency decreases as $p \to \infty$

Efficiency increases as $n \to \infty$

We noted before that if $n = \Omega(p \lg p)$, then the algorithm is cost-optimal.

Suppose $n = cp \lg p$. Suppose we continue to choose n and p so that this is true for a fixed c. Then efficiency E will be maintained.

n	p	$p \lg p$	$n/p \lg p$
64	4	8	8
192	8	24	8
512	16	64	8
...	...	...	8

Efficiency can be kept fixed as p and n increase to yield a scalable algorithm/architecture

Minimizing Execution Time

What is the minimum (fastest) execution time of an algorithm if the number of processors is not a constraint?

- To find the T_p^{min}, the minimum value of T_p, solve

$$\frac{\mathrm{d}}{\mathrm{d}p} T_p = 0$$

  and call the solution p_0.

- Then evaluate T_p at this value, substituting $p = p_0$.

Example: adding n numbers on a p node hypercube

$$\begin{aligned} T_p &= \frac{n}{p} + 2\lg p \\ -\frac{n}{p^2} + \frac{2}{p} &= 0 \\ p &= \frac{n}{2} \end{aligned}$$

giving $T_p^{min} = 2\lg n$

Amdahl's Law

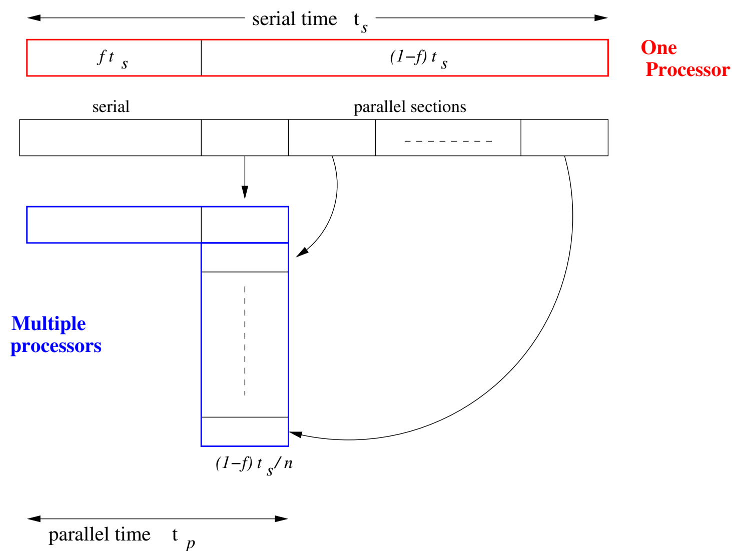
Amdahl's Law predicts speedup based on a *fixed* problem size (i.e. measures constant problem size scaling)

- Let f be the fraction of the code that must run serially
- Define t_s as the execution time on one processor. Then

$$t_s = f \cdot t_s + (1 - f) \cdot t_s$$

- Define t_p as the execution time using n processors. Then

$$t_p = f \cdot t_s + \frac{(1 - f) \cdot t_s}{n}$$



- Equivalently, we can write

$$t_s = s + p$$
$$t_p = s + \frac{p}{n}$$

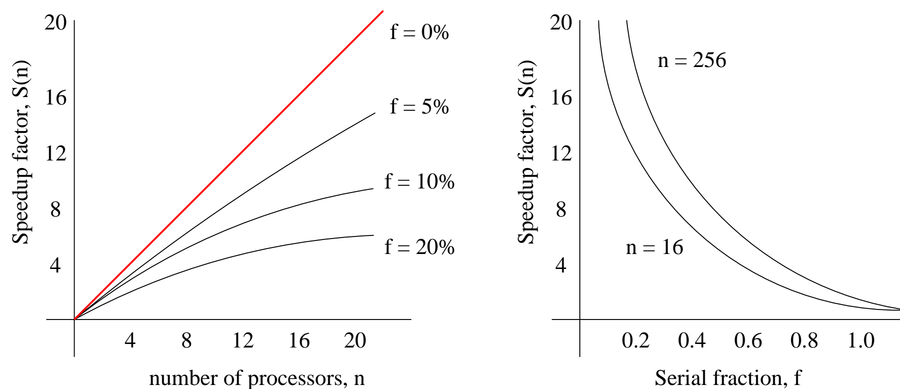
where s is the time for the serial code and p is the time for the parallel code on a single processor.

- Speedup is measured in Amdahl's law as

$$S(n) = \frac{s+p}{s+\frac{p}{n}} = \frac{1}{s+\frac{1-s}{n}} = \frac{n}{1+(n-1)s}$$

when $s + p$ is normalized to 1.

©2004 P.Y. Wang



©2004 P.Y. Wang

Gustafson's Law

Gustafson's law for scaled speedup:

- Is based on an observation that problem size is not independent of the number of the parallel processors
- Assumes that the parallel execution time is fixed (i.e. measures time constrained scaling):
- Let s be the time for the serial code and p be the time for the parallel code when the program is run on a *parallel computer*.

$$\begin{aligned} t_p &= s + p = 1 \text{ (normalized)} \\ t_s &= s + pn \end{aligned}$$

- Speedup is measured by Gustafson as:

$$\begin{aligned} S(n) &= \frac{s + pn}{s + p} = s + pn \\ &= s + (1 - s)n \\ &= n + (1 - n)s \end{aligned}$$

- Scaled speedup, as a function of s (sequential code running on a parallel processor), is a negative slope line.
- Example: n=20, s=0.05 (5% serial code)
  - Scaled speedup measured as 19.05
  - Speedup using Amdahl's law measured as 10.26

Other Performance Measures

There are many metrics and analyses of performance that have been proposed in the literature, including

- iso-efficiency measures
- time-constrained scaling
- memory-constrained scaling
- PRAM efficiency
- approaches based on thread-counting

©2004 P.Y. Wang

Sources of Parallel Overhead

- Interprocessor Communication
  - The time to transfer data is usually the most significant source of overhead
- Load Imbalance
  - Most algorithms require the processors to synchronize at certain points of computation. Idle processors must wait for others.
  - Some parts of computation are inherently serial and can only be performed by one processor

©2004 P.Y. Wang

- Extra Computation
  - Parallelizing a sequential computation can lead to extra computations; results computed by serial code can often be reused– the parallel version may need to recompute those results

©2004 P.Y. Wang