CS4223
Thread-Level Parallelism Basics

Trevor E. Carlson
National University of Singapore
tcarlson@comp.nus.edu.sg
(Slides from Tulika Mitra)

Point of no return

- We are dedicating all of our future product development to multi-core designs. We believe this is a key inflection point for the industry.

  -- Intel President Paul Otellini

  describing Intel's future direction at the

  Intel Developer Forum 2005

TLP Landscape

- Loop Iterations
  - Each iteration works with independent data elements and is therefore an independent chunk of parallel work
- Task-level parallelism
  - Large, independent functions extracted from a single application are known as tasks
    - Example: word processor invoking background task to perform spell check
- Request-level parallelism
  - Wed server allocating each request coming in from the network to its own task
- Processes
  - Completely independent OS processes, all from different applications and each with own separate virtual address space
  - Also known as multiprogramming

Parallel Programming Models

- Message-passing programming paradigm
  - Oldest and most widely used approach for programming parallel computers
  - Imposes minimal requirements on the underlying hardware
- Shared-memory programming paradigm
  - Increasingly popular with the emergence of shared memory multiprocessors on chip

Message Passing Programming

- Partitioned address space
  - Logical view consists of *p* processes, each with its own exclusive address space
  - Each data element must belong to one of the partitioned address space --- explicit data partitioning
    - Programming difficulty but better locality of access
  - All interactions require cooperation of both the process that has the data and the process that wants the data
    - Programming difficulty but forces programmers to minimize interactions
  - Natural fit for clustered workstations
- Supports only explicit parallelization

Building Blocks: Send and Receive

- Interactions are accomplished by sending and receiving messages
  - `send (void *sendbuf, int nelems, int destination)`
  - `receive(void *recvbuf, int nelems, int source)`

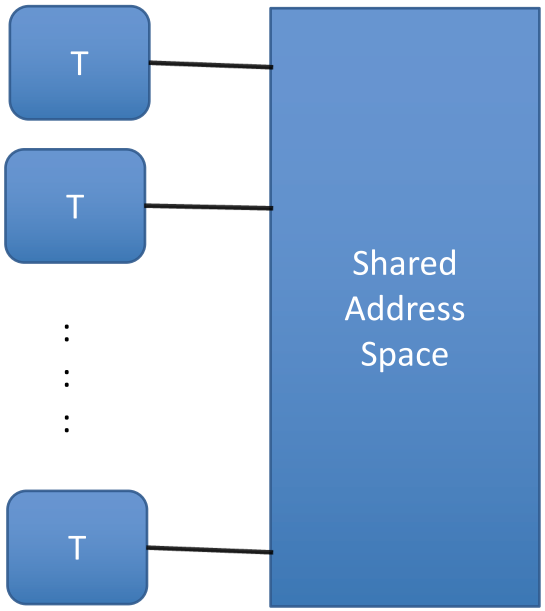
```
P0                      P1

a = 100;                receive(&a, 1, 0);
send (&a, 1, 1);        printf("%d\n", a);
a = 0;
```

Deadlock in send/receive

P0	P1
`send (&a, 1, 1);`	`send(&a, 1, 0);`
`receive (&b, 1, 1);`	`receive(&b, 1, 0);`

Shared memory programming model

- All memory in the logical machine model of a thread is globally accessible to every thread in the system



Threads

- A thread is a single stream of control in the flow of a program

```
for (row = 0; row < n; row++)
  for (col = 0; col < n; col++)
     c[row][col] = create_thread(dotproduct(get_row(a,row),
                                            get_col(b,col)));
```

Why threads

- Software portability: Threaded application can be developed on serial machines and run on parallel machines without any change
- Latency hiding: While one thread is waiting for communication operation, another thread can utilize the processor
- Scheduling and load balancing: Programmer must express concurrency in a way that minimizes communication and idling
- Ease of programming: Shared memory programs are much easier to write than message-passing programs

Synchronization Primitives

- Much effort on synchronizing concurrent threads with respect to their data accesses or scheduling
- When multiple threads attempt to manipulate the same data item, the result can be incoherent if proper care is not taken to synchronize them

```
/* each thread tries to update variable best_cost as follows */
if (my_cost < best_cost)
        best_cost = my_cost;

/* Initial value of best_cost = 100;
   my_cost = 50 and 75 in threads T1 and T2 */
```

Mutex lock

- Support for implementing critical sections and atomic operations
- To access shared data, a thread must first try to acquire mutex
- At any point in time, only one thread can lock a mutex
- If the mutex is already locked, the thread trying to acquire the lock is blocked
- When a thread is done accessing shared data, it should release the mutex

Barrier

- A barrier call is used to hold a thread until all the other threads participating in the barrier have reached the barrier

Sequential Code

```
for (i=0; i<8; i++)
  a[i] = b[i] + c[i];

sum = 0;
for (i=0; i<8; i++)
    if (a[i] > 0)
        sum = sum + a[i];
print sum;
```

Message passing code

```
id = getmyid();
local_iter = 4;
start_iter = id * local_iter;
end_iter = start_iter + local_iter;

if (id == 0)
  send_msg (P1, b[4..7], c[4...7]);
else
  recv_msg (P0, b[4..7], c[4..7];

 for (i=start_iter; i<end_iter; i++)
    a[i] = b[i] + c[i];

local_sum = 0;
for (i=start_iter; i<end_iter; i++)
      if (a[i] > 0)
          local_sum = local_sum + a[i];
if (id == 0) {
     recv_msg (P1, &local_sum1);
     sum = local_sum + local_sum1;
     print sum;
}
else
     send_msg (P0, local_sum);
```

Shared memory code

```
begin parallel    // spawn a child thread
private int start_iter, end_iter, i;
shared int local_iter = 4;
shared double sum = 0.0, a[], b[], c[];
shared lock_type mylock;

start_iter = getid() * local_iter;
end_iter = start_iter + local_iter;
for (i=start_iter; i<end_iter; i++)
    a[i] = b[i] + c[i];
barrier;

for (i=start_iter; i<end_iter; i++)
      if (a[i] > 0) {
          lock (mylock);
          sum = sum + a[i];
          unlock (mylock);
      }
barrier;
end parallel // kill the child thread
print sum;
```

Shared memory vs. message passing

Aspects	Shared Memory	Message Passing
Communication	Implicit (load/store)	explicit (via msg)
Synchronization	Explicit	Implicit (via msg)
Hardware Support	Typically required	None
Development effort	Lower	Higher
Tuning effort	higher	lower

Amdahl's Law

- Performance improvement to be gained from using some faster mode of execution is limited by the fraction of the time the faster mode can be used

$$Speedup = \frac{Time_{old}}{Time_{new}}$$

$$Time_{new} = Time_{old} \times \left(\left(1 - Fraction_{enhanced}\right) + \frac{Fraction_{enhanced}}{Speedup_{enhanced}} \right)$$

$$Speedup = \frac{1}{\left(1 - Fraction_{enhanced}\right) + \frac{Fraction_{enhanced}}{Speedup_{enhanced}}}$$

Corollaries

- Corollary 1: If the enhancement is only applicable for a fraction of a task, we cannot speed up the task by more than

$$\frac{1}{1 - Fraction_{enhanced}}$$

- Corollary 2: Make the common case first (In making a design trade-off, favor the frequent case over the infrequent case)

Amdahl's Law for Multiprocessor

- α: fraction of the serial portion of application
- N: number of processors

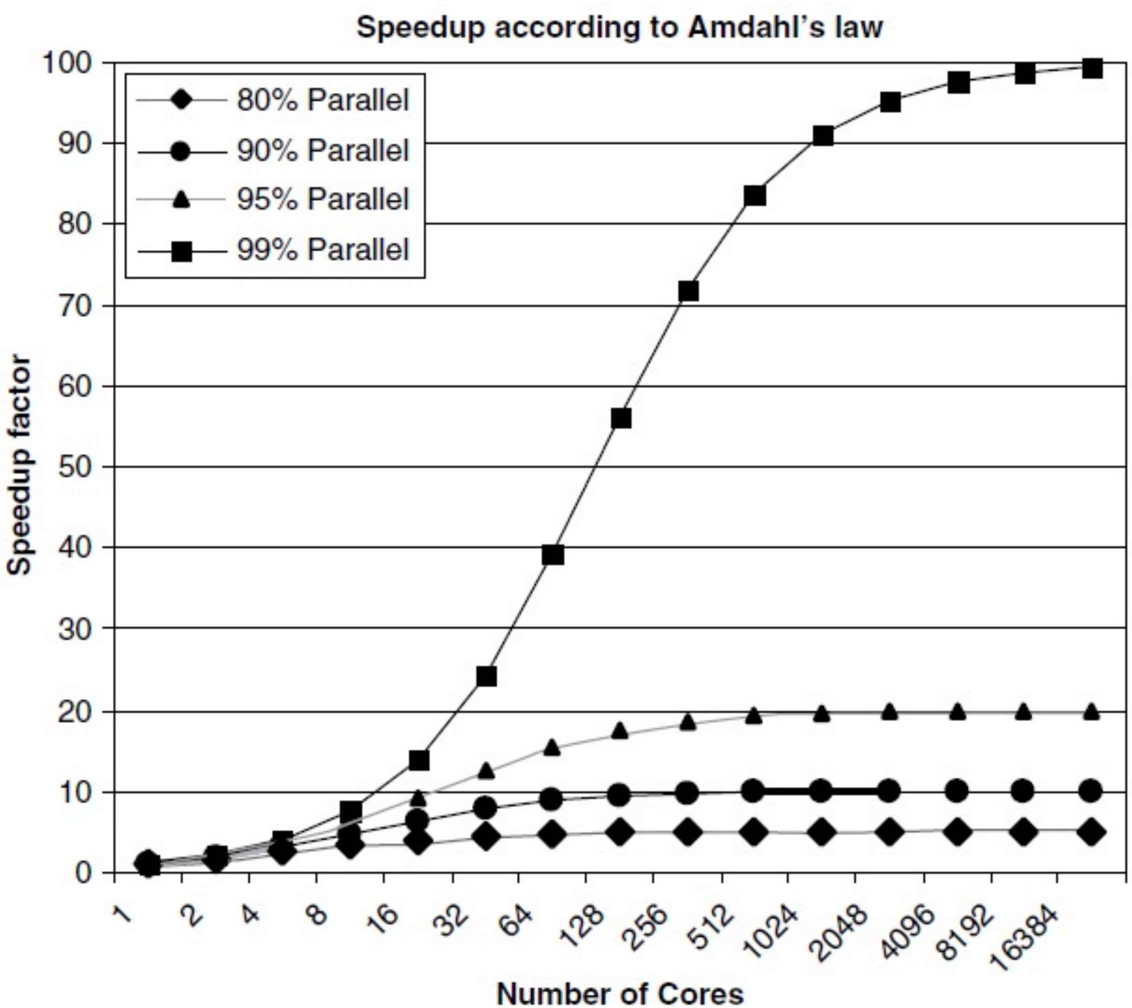
$$Speedup = \frac{1}{\alpha + \frac{1-\alpha}{N}}$$

- When N approaches ∞, the upper bound is

$$Speedup = \frac{1}{\alpha}$$

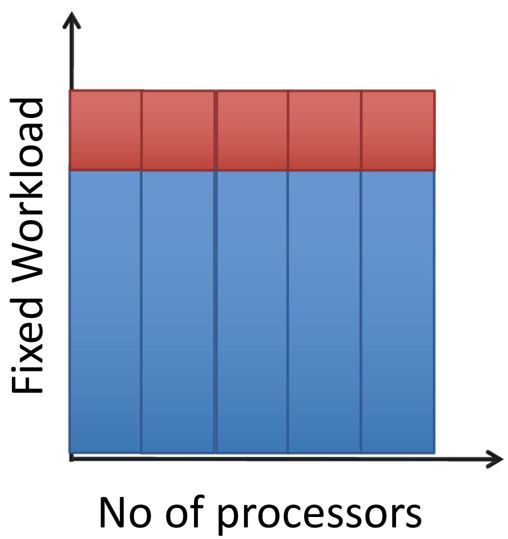
- Assumption: The workload or problem size is fixed

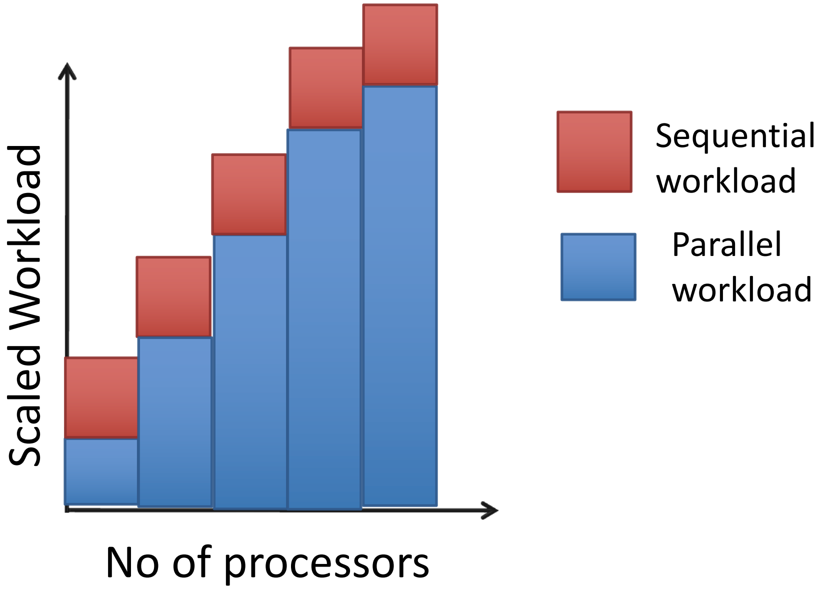
Speedup curve as per Amdahl's Law



Gustafson's Law for Scaled Problems

- As number of processors increases, we may want to increase the problem size to improve the quality of the solution
  - e.g., Reduced grid spacing and increased time steps to improve weather forecasting simulation accuracy





Gustafson's Law

- T(N): execution time of a program on N processors
- ser(N): execution time of sequential portion
- par(N): execution time of parallel portion
- T(N) = ser(N) + par(N) = 1
- T(1) = ser(N) + N × par(N)
- Speedup = T(1) / T(N)

  = ser(N) + N x (1- ser(N))

  = N – (N - 1) x ser(N)

Speedup according to Gustafson's Law