Lecture 9:

Introduction to Parallel Architecture

Department of Electrical Engineering
Stanford University

http://eeclass.stanford.edu/ee282

Announcements

- Quiz 1, Wed 10/22, 6-9pm, Cubberley auditorium
  - Lectures 1-9, closed books
  - Bring calculator + 1 page of notes
- Local SCPD students must come to campus
- Remote SCPD students
  - Take quiz anytime on Wed 10/22
  - Make arrangements with SCPD about your proctor
- No lecture on 10/22
  - Extra office hours instead in Gates Hall 304

Review: Optimizing SW for Memory Hierarchies

- Goal: eliminate cache misses
  - Enhance temporal locality (reuse) & spatial locality (sequential accesses)
  - Eliminate conflicts
- What can software control?
  - Memory layout (where & how an object is placed)
  - Access order (when an order is placed)
- Steps for code optimization
  1. Select a good algorithm
     - Pitfalls: ignore the constants, ignore problem size
  2. Use efficient libraries
     - Pitfall: library may not address all the problem cases you care about
  3. Select right compiler and compile options
     - Pitfall: overspecialize or underspecialize your executable; caching vs other overheads
  4. Optimize your code manually for locality

Review: Example Optimizations (mostly focusing on array & loops)

- Optimizations that change access order
  - Loop interchange & reversal
  - Loop fusion & fission
  - Array blocking
- Optimizations that focus on layout
  - Array merging
  - Array padding
- Miscellaneous
  - Inline function calls, eliminate if-statements, …
- Always watch out for
  - Code size effects, overheads, branch predictability

Array Padding

```
int a[N], b[N];
for (i=0; i<N; i++)
   sum += a[i]*b[i];
```

- Assume
  - N is large
  - N multiple of cache size
  - Direct mapped cache
  - Arrays allocated sequentially
  - Cache line is 32 bytes
- How many misses?

```
int A[N], pad[8], B[N];
for (i=0; i<N; i++)
   sum += a[i]*b[i];
```

- Any better now?
- What is the cost?
- Can also do intra-array padding
  - Pad rows in a 2D array
  - Can reduce conflicts for banks/associativity

Loop Unrolling

- Consider the following loop:

```
for (i=0; i<100; i++)  a[i] = b[i] + c;
```

- Loop unrolled twice:

```
for (i=0; i<100; i+=2) {
  a[i] =   b[i]   + c;
  a[i+1] = b[i+1] + c;  }
```

- How to unroll K times a loop with N iterations
  - Start with original loop for (N mod K) iterations
  - Continue with unrolled loop for [N/K] iterations
  - This is called "strip mining"

Loop Unrolling

- Advantages
  - Reduces loop overhead (branches)
  - Exposes independent instructions (more ILP)
    - Very useful with wide issue machines
- Disadvantages & limitations
  - Needs more registers for renaming
  - Increased code size
  - Loop-carried dependencies
    - Limited ILP despite unrolling
  - Conditional statements (if) within each iteration

The Compiler is Your Friend

- Choose an optimizing compiler and go beyond –O3
  - Read your compiler's optimization manual
- Optimizations the compiler should always do for you
  - Code motion, strength reduction, common subexpression elimination, register allocation, instruction scheduling, inlining,…
- But it can often do more
  - Unrolling & software pipelining, loop optimizations, optimized build-in functions, inter-procedural analysis, prefetching, padding & alignment
- But be careful…
  - Is the optimization safe for your code (e.g. FP rounding issues)
  - Optimizations can reveal bugs in your code

Keep in Mind: Limitations of Optimizing Compilers

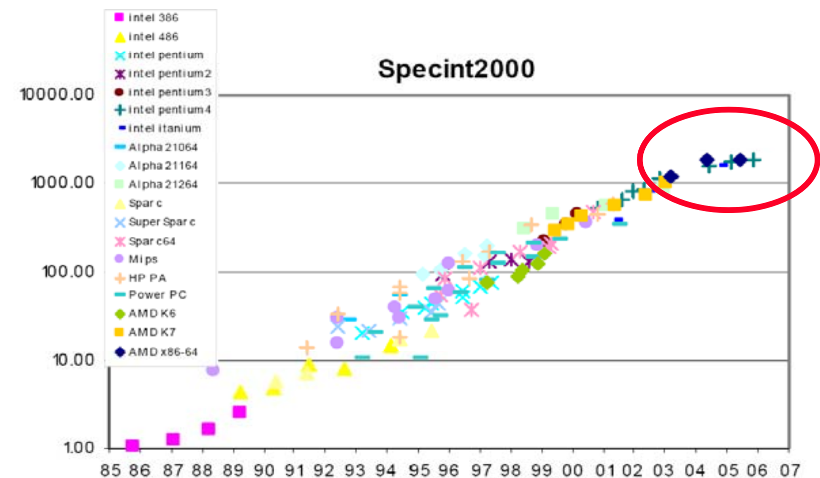
- Operate Under Fundamental Constraint
  - Must not cause any change in program behavior under any possible condition
- Most analysis is performed only within procedures
  - Whole-program analysis is too expensive in most cases
- Most analysis is based only on static information
  - Compiler has difficulty anticipating run-time inputs
  - Profile driven compilation becoming more prevalent (JIT)
- When in doubt, the compiler must be conservative

Introduction to Parallel Machines

Why Parallel Systems?

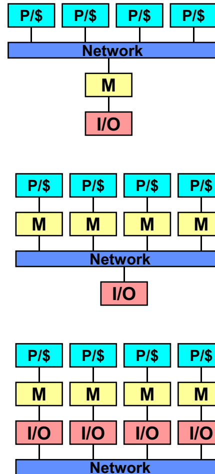
- Need ever increasing performance to
  - Solve large problems faster
    - Reduce turn-around time of big jobs
    - Increase responsiveness of interactive apps
  - Get better solutions in same amount of time
    - Increase resolution & sophistication of models
  - Consistent need across all types of computing
    - Server, personal, embedded…
- Cannot improve uni-processor systems any longer
  - Power consumption
  - Design complexity
  - Diminishing returns from instruction level parallelism

The End of Uniprocessor Performance Improvement



Parallel Systems

- Parallel systems come in many flavors
  - CMPs, SMPs, ccNUMA, MPPs, clusters…
- Differentiating factors to keep in mind
  - Degree of integration
  - Which resources are parallelized?
    - Processors, memory channels, I/O channels, …
  - Uniform Vs. non-uniform storage access
    - For memory and I/O devices
  - Communication through memory or I/O accesses
- These choices have implications on
  - Scaling, suitability to specific apps, cost, software infrastructure, …



How Can We Exploit Parallelism?

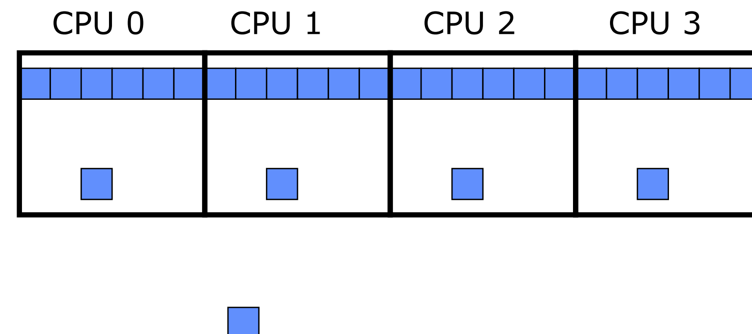
- Parallel programming
  - Go from a sequential algorithm to a parallel algorithm
    - Use multiple processors in collaborative manner to solve a problem
  - Focus on the parts of the program that take most time
    - Remember Amdhal's Law (we'll come back to this)
- A crucial distinction
  - Concurrent execution
    - Multiple computations are active, but only one of them is running at the time
  - Parallel execution
    - Multiple active computations and several are running at the same time
- Approaches to parallelizing a program
  - Domain or data decomposition
  - Task or functional decomposition
  - Task pipelining
  - Combinations of the above…

Domain or Data Decomposition

- Subdivide a large dataset into groups to be computed in parallel
  - Divide data and associated computation among available processors
  - Simple example: add two long vectors
  - Computations may be fully independent or may require communication & coordination

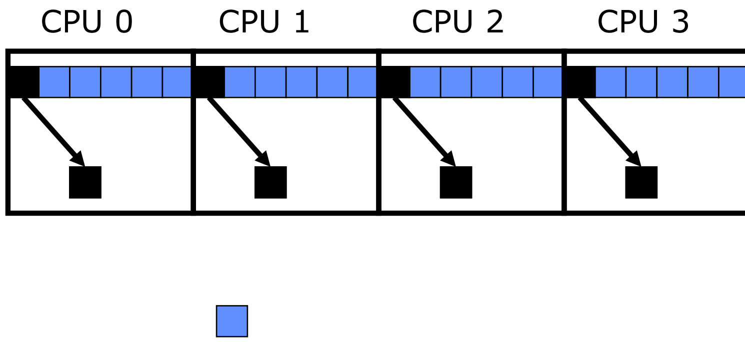
Domain Decomposition Example

Find the largest element of an array



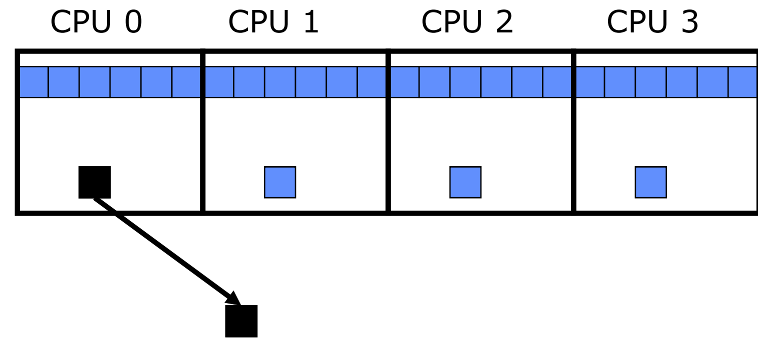
Domain Decomposition Example

Find the largest element of an array



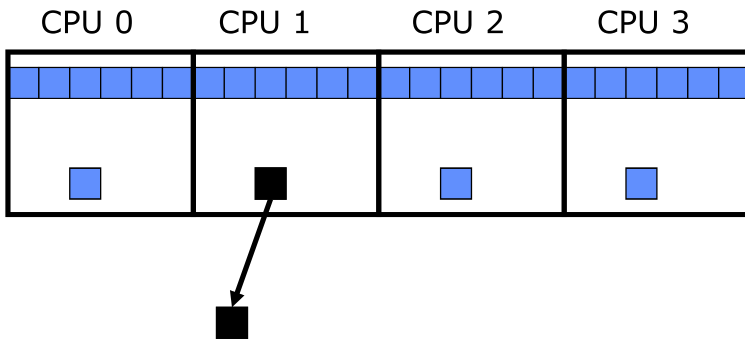
Domain Decomposition Example

Find the largest element of an array



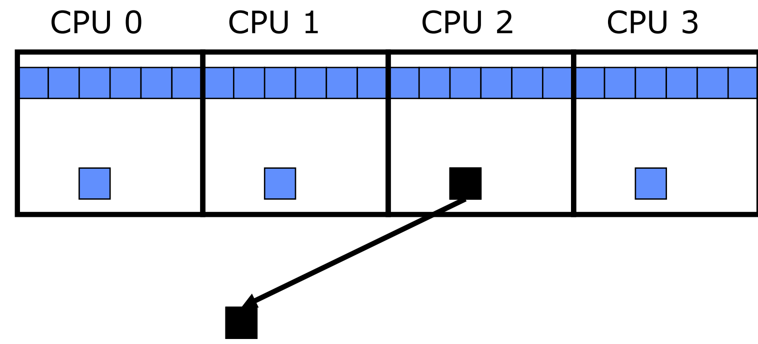
Domain Decomposition Example

Find the largest element of an array



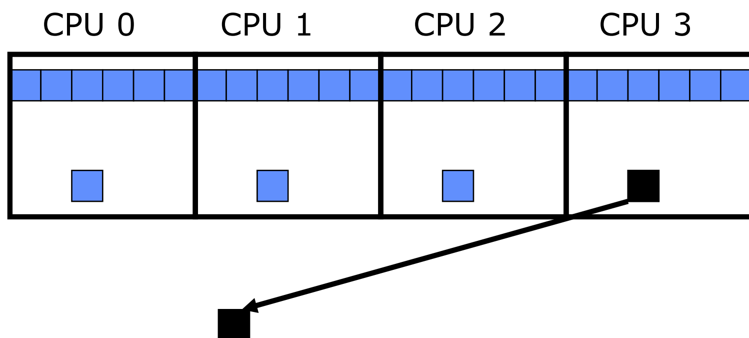
Domain Decomposition Example

Find the largest element of an array



Domain Decomposition Example

Find the largest element of an array



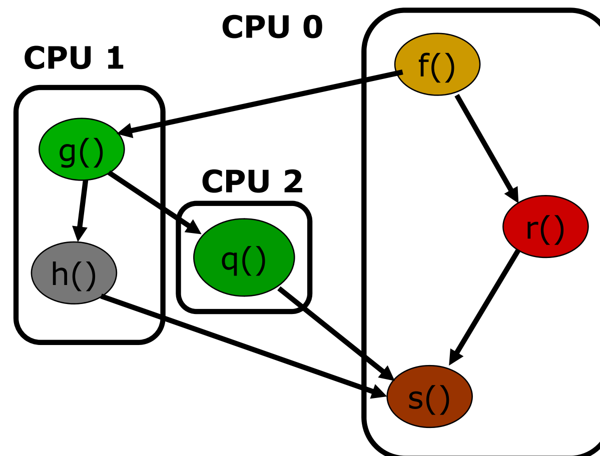
Summary: Domain or Data Decomposition

- Subdivide a large dataset into groups to be computed in parallel
  - Divide data and associated computation among available processors
  - Computations may be fully independent or may require communication & coordination
- Process
  - Decide how data elements should be divided among processors
  - Decide which tasks each processor should be executing
    - What should we take into account in either case?

Task or Functional Decomposition

- Divide computation based on natural set of separate tasks
  - Assign data for each task as needed
    - Simple example: event-handler for GUI
  - Tasks may be fully independent or may require communication & coordination

Task Decomposition Example

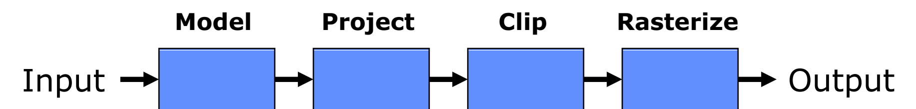


Summary Task or Functional Decomposition

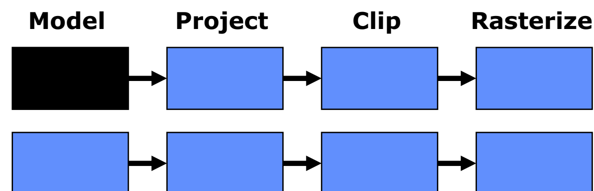
- Divide computation based on natural set of separate tasks
  - Assign data for each task as needed
  - Simple example: event-handler for GUI
  - Tasks may be fully independent or may require communication & coordination
- Process
  - Divide tasks among processors
  - Decide which data elements are going to be accessed (read and/or written) by which processors

Task Pipelining

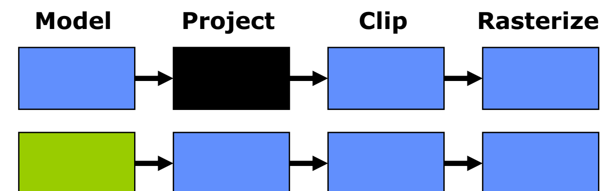
- Assembly line parallelism
  - Tasks executed in parallel on different input data
    - Just like the processor pipeline but in software…
  - This is a special kind of task decomposition
    - What kind of coordination is needed between tasks?
  - Example: video encoding or 3D rendering



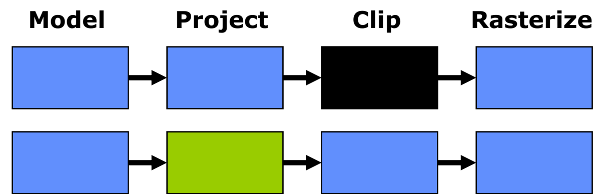
Processing Two Data Sets (Step 1)



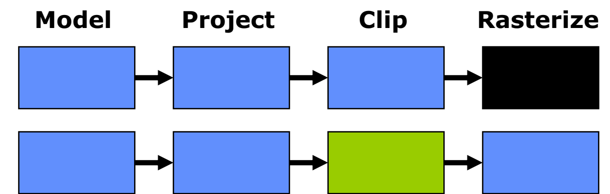
Processing Two Data Sets (Time 2)



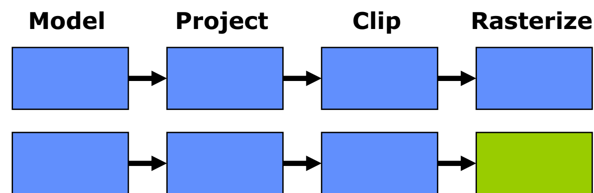
Processing Two Data Sets (Step 3)



Processing Two Data Sets (Step 4)



Processing Two Data Sets (Step 5)



The pipeline processes 2 data sets in 5 steps

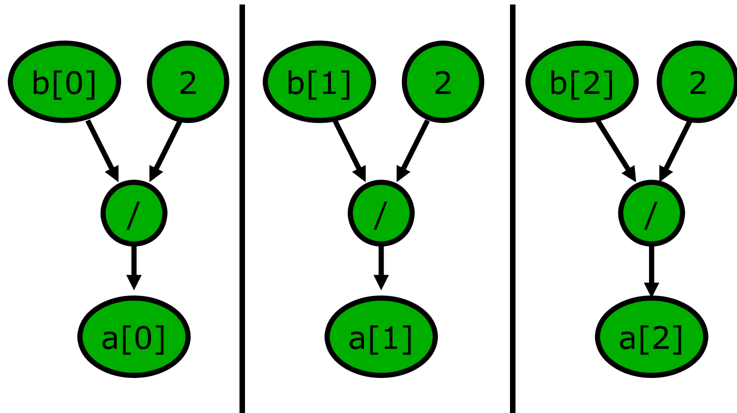
Dealing with Dependencies

- Dependencies between two tasks complicate parallelism
  - They dictate if parallelism is possible
  - They dictate possible approaches
- Types of dependencies to consider
  - Data dependencies: definition?
  - Control dependencies: definition?
- Dependence graphs: a way to visualize dependencies
  - Graph = (nodes, arrows)
  - Node for each variable assignment, constant, operation or function, …
  - Arrows indicate use of variables and constants
    - They capture data flow and control flow
    - They indicate dependencies!

Dependence Graph Example #1

```
for (i = 0; i < 3; i++)
  a[i] = b[i] / 2.0;
```

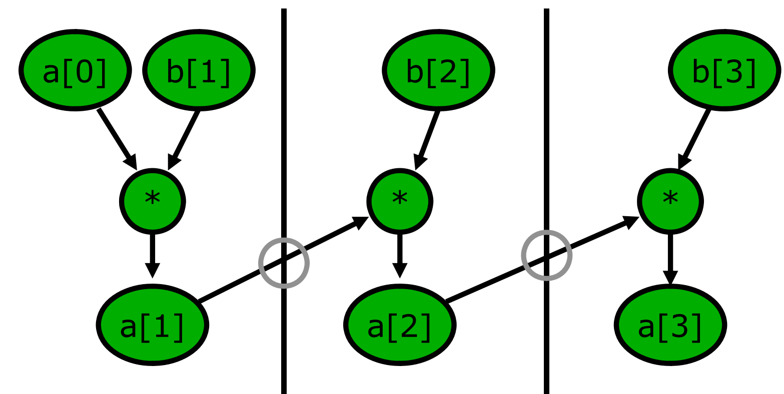
Domain decomposition possible



Dependence Graph Example #2

```
for (i = 1; i < 4; i++)
  a[i] = a[i-1] * b[i];
```

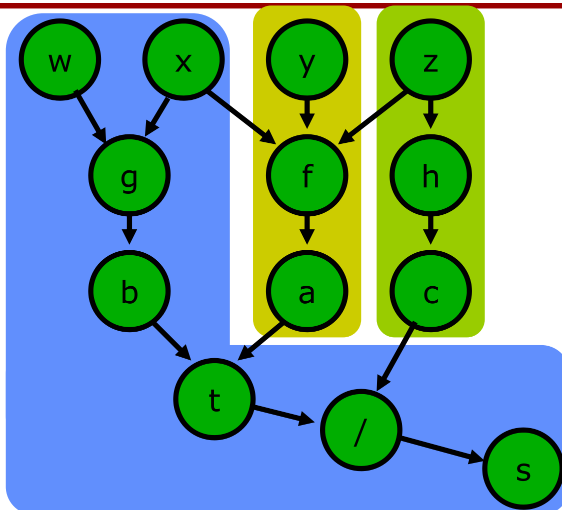
No domain decomposition



Dependence Graph Example #3

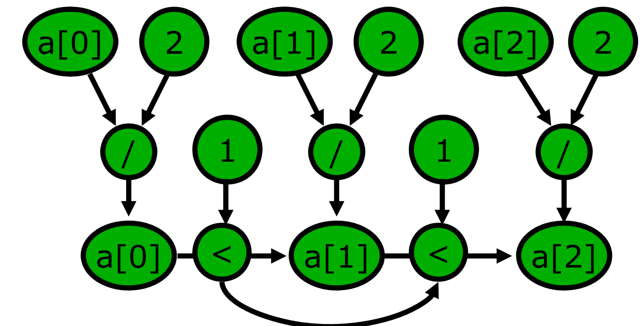
```
a = f(x, y, z);
b = g(w, x);
t = a + b;
c = h(z);
s = t / c;
```

Task decomposition with 3 CPUs.



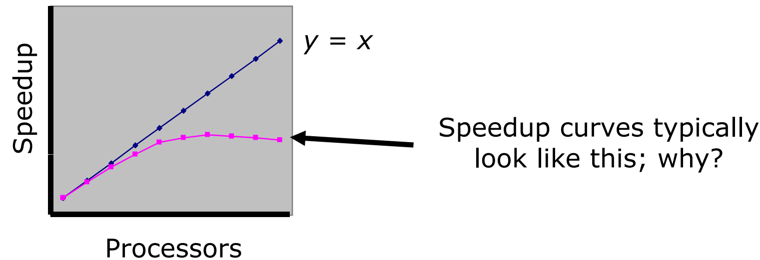
Dependence Graph Example #4

```
for (i = 0; i < 3; i++) {
  a[i] = a[i] / 2.0;
  if (a[i] < 1.0) break;
}
```



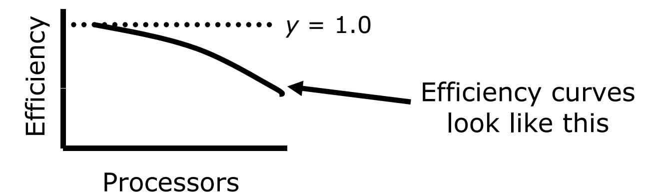
Metrics: Speedup

- The ratio between sequential parallel execution time
  - For example, if the sequential program executes in 6 sec and the parallel program executes in 2 sec, the speedup is 3



Metrics: Efficiency

- Efficiency: speedup divided by number of processors
  - A measure of processor utilization
- Example
  - Program achieves speedup of 3 on 4 CPUs
  - Efficiency is 3 / 4 = 75%



Metrics: Amdahl's Law

$$speedup \leq \frac{(1-f)+f}{(1-f)+f/p} = \frac{1}{(1-f)+f/p}$$

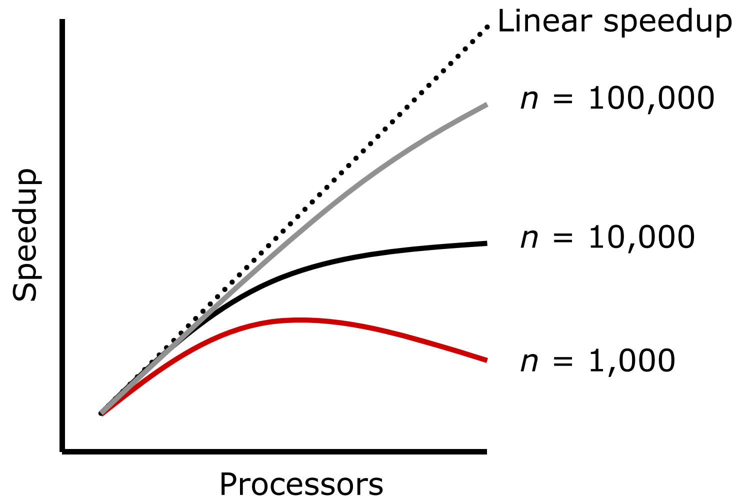
- Amdahl's Law is too optimistic
  - Ignores parallel processing overhead
    - Time spend creating and terminating parallel computations
    - Communication and synchronization overhead
    - Load imbalance
  - Parallel processing overhead is usually an increasing function of the number of processors

Metrics: More General Speedup Formula

- *speedup(n,p)* Speedup for problem of size *n* on *p* CPUs
- $\sigma(n)$ Time spent in sequential portion of code for problem of size *n*
- $\varphi(n)$ Time spent in parallelizable portion of code for problem of size *n*
- $\kappa(n,p)$ Parallel overhead

$$speedup(n,p) \leq \frac{\sigma(n)+\varphi(n)}{\sigma(n)+\varphi(n)/p+\kappa(n,p)}$$

Metrics: Speedup Vs Dataset Size



Metrics: Superlinear Speedup?

- According to speedup formula, maximum speedup is P
  - Assuming P processors
- Yet, we can often observe superlinear speedups
  - Speedup > P
  - It means the computational rate of the processors is faster when the parallel program is executing
- What can cause superlinear speedups?

Scalability

- How do we scale a system:
  - More processors, memory, network paths, I/O paths, …
- A service is **scalable** if when we **increase the resources** in a system, it results in **increased performance** in a manner **proportional** to resources added
  - No change in efficiency
- The catch: scaled up systems typically used with scaled-up problems
  - E.g., larger data-sets

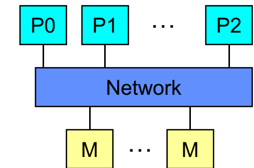
Limitations to Parallel Speedup & Scalability

- Application characteristics
  - Serially dominated workload
  - Parallel overhead
  - Excessive communication & coordination
  - Slow communication & coordination primitives used
  - I/O bottlenecks
  - Load Imbalance
  - Locality issues
- Parallel system characteristics
  - Slow communication & coordination primitives
  - Networking bottlenecks
    - Bandwidth saturation, latency, …
  - Non-uniform storage access

Basic Questions about Parallel Systems

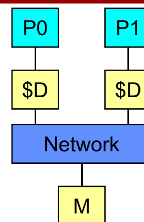
- These questions apply to hardware and software
  - Communication
    - How do parallel operations communicate data results?
  - Synchronization
    - How are parallel operations coordinated?
  - Resource Management
    - How are a large number of tasks scheduled onto finite hardware?
  - Scalability
    - How large a machine/program can be built?
- Two major parallel models for parallel systems
  - Shared-memory
  - Message-passing

Shared Memory Model



- Single memory space visible to all processors
- Control model: parallel threads
  - ≥1 thread per CPU
- Communication
  - Through loads/stores to shared locations (implicit)
  - To communicate lot's of info, pass a pointer
- Synchronization through special operations
  - Locks and barriers
    - We'll discuss barriers soon…
  - Implemented with low-level atomic operations
    - Test & set, fetch & add, load-linked & store conditional

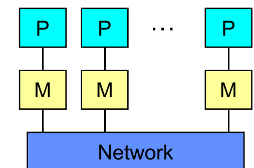
What Makes Shared Memory Difficult to Build: Cache Coherence



- Assume the following sequence
  - P0 loads A in its data cache
  - P1 loads A in its data cache
  - P0 writes to new value A'
  - P1 loads A, which value does it get?
- Hardware cache coherence:
  - Hardware verifies there is a single value of A across system
- Solutions:
  - Shared memory without cache coherence
    - E.g. do not cache remote accesses
    - Scales to 1,000s of processors (e.g. Cray T3D or X1)
  - Bus-based snooping protocol for HW cache coherence
    - Bus provides broadcast & serialization
    - Good for SMPs & CMPs but does not scale beyond 32-64
  - Directory protocol for HW cache coherence
    - HW maintains directory for each memory location (provides serialization)
    - Scales to NUMA but still no ccNUMA systems with >100 processors

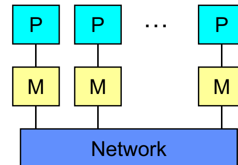
Message Passing Architectures

- Multiple address spaces in the system
- Control model: multiple threads or processes
- Inter-processor communication
  - Send messages over network (explicit)
    - send(tid, tag, message)
    - receive(tid, tag, message)
- Synchronization
  - Block on messages
  - Barriers (use messages to implement)



Implementation of Message Passing Model

- Each node is typically a complete computer
  - CPU, DRAM main memory, IO system, commodity networking
  - It is typically cheaper to implement (commodity hardware)
- Networking may connect at
  - IO bus
  - Memory bus
- Messaging may be a
  - OS-level operation
    - Protected by OS, higher overhead
  - User-level operation
- Clusters are message-passing machines that use commodity hardware



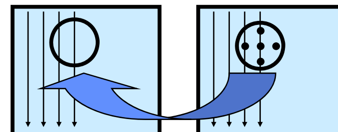
Shared-memory vs Message-Passing

- Single address space for all CPUs
- Communication through regular load/store operations
  - Implicit
- Synchronization using locks and barriers

- Private address-spaces for CPUs
- Communication through message send/receive operations (through memory or I/O network)
  - Explicit
- Synchronization using blocking messages

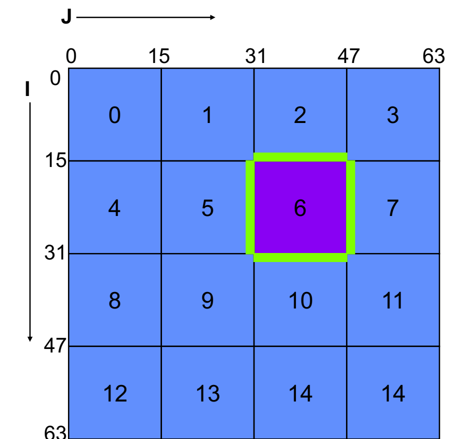
An Example - Iterative Solver

```
double a[2][MAXI+2][MAXJ+2]; //two copies of state
                             //use one to compute the other
for (s=0; s<STEPS; s++) {
    k = s&1;                    // 0 1 0 1 0 1 ...
    m = k^1;                    // 1 0 1 0 1 0 ...
    forall(i=1; i<=MAXI; i++) {  // do iterations in parallel
        forall(j=1; j<=MAXJ; j++){
            a[k][i][j] =  c1*a[m][i][j] + c2*a[m][i-1][j] +
                          c3*a[m][i+1][j] + c4*a[m][i][j-1] +
                          c5*a[m][i][j+1];
        }
    }
}
```



Domain Decomposition

- Divide matrix A over 16 processors
  - Each processor computes a 16x16 submatrix
- Processor 6
  - Owns [i][j] = [32..47][16..31]
  - Shares [i][j] = [31][16..31] and three other strips
- Each processor
  - Communicates to get shared data it needs
  - Computes its data
  - Synchronizes



Shared Memory Code

```
Fork N processes
   each process, p computes
   istart[p], iend[p], jstart[p], jend[p]

For (s=0; s<STEPS; s++) {
   k = s&1;
   m = k^1;
   forall(i=istart[p]; i<=iend[p]; i++) {     // e.g. 32..47
       forall(j=jstart[p]; j<=jend[p]; j++){  // e.g. 16..31
          a[k][i][j] = c1*a[m][i][j] + c2*a[m][i-1][j] +
                       c3*a[m][i+1][j] + c4*a[m][i][j-1] +
                       c5*a[m][i][j+1]; // implicit comm.
      }
   }
   barrier();
}
```


Message Passing Code

```
Fork N processes and distribute subarrays to processors
Each processor computes north[p], south[p], east[p], west[p],
   -1 if no neighbor in direction

for (s=0; s<STEPS; s++) {
   k = s&1;   m = k^1;
   if (north[p]>= 0) send(north[p], NORTH, a[m][1][1..MAXSUBJ]);
   if (east]>= 0) send(east[p], EAST, a[m][1..MAXSUBI][1]);
   same for south and west
   if (north[p]>= 0) receive(NORTH, a[m][0][1..MAXSUBJ]);
   same for other directions
   forall(i=1; i<=MAXSUBI; i++) {
      forall(j=1; j<=MAXSUBJ; j++){
         a[k][i][j] =   c1*a[m][i][j] + c2*a[m][i-1][j] +
                        c3*a[m][i+1][j] + c4*a[m][i][j-1] +
                        c5*a[m][i][j+1];
      }
   }
}
```


Shared-memory Vs Message Passing Programming

- Shared memory
  - Typically easier to write first correct version
    - Communication through load/stores, just get synchronization right
  - Typically more difficult to write fully optimized version
    - Difficult to tell which loads/stores lead to communication
  - Often more difficult to scale
    - Can create fine-grain communication/synchronization
- Message passing
  - Typically more difficult to write first correct version
  - Typically easier to write fully optimized version
    - Communication/synchronization on sends/receives
  - Often easier to scale
    - Typically leads to coarse-grain communication/synchronization

Shared-memory Vs Message Passing Hardware

- Shared-memory hardware
  - Have to worry about fine-grain communication
  - Have to worry about cache coherence
  - Have to worry about consistency model
    - More details in CS315A/EE386A in spring quarter
- Message-passing hardware
  - Have to worry about coarse-grain communication
  - Have to worry about message-passing overhead
    - Buffering, copying, protection etc
  - Have to worry about explicit locality management

Convergence of Models

- Can do
  - Message passing programs on top of shared memory hardware
    - Load/stores to shared buffers to implement messages
    - This is how custom message passing machines work
      - But no coherence…
  - Shared memory programs on top of message passing hardware
    - Use virtual memory system to implement sharing

- Can combine shared-memory & message-passing hardware
  - Message-passing cluster with each node a shared-memory multiprocessor

- Within a chip (multi-core or CMP system), we can greatly improve both shared-memory and message-passing models
  - Lower latency, more bandwidth, simpler networks, specialized HW suport …