Concurrency

- A process or thread is a potentially active execution context
- Classic von Neumann model – single thread of control, single active execution context
- Concurrency – two or more execution contexts may be active at the same time

Why concurrency?

- Capture the logical structure of certain apps (e.g., netscape, video games/interactive I/O)
- Cope with independent physical devices (e.g., via device interrupts to the OS)
- Performance through use of multiple processors

Concurrency in the form of

- Explicitly concurrent languages
  - e.g., Algol 68, Java
- Compiler-supported extensions
  - e.g., HPF, Power C/Fortran
- Library packages outside the language proper
  - e.g., pthreads

- Multiple threads of control can come from
  - Multiple CPUs
  - Kernel-level multiplexing of single physical machine
  - Language or library-level multiplexing of kernel-level abstraction
- They can run
  - In parallel
  - Unpredictably interleaved
  - Run-until block

Thread Creation Syntax

- Co-begin (e.g., Algol 68)
- Parallel loops (e.g., Occam, Fortran90)
- Launch at elaboration (e.g., Ada, SR)
- Fork-join (e.g., Ada, Modula-3, Java)
- Implicit-receipt
- Early reply

Thread Implementation

- Schedulers give us the ability to switch execution contexts
  - Start with coroutines (e.g., Simula, Modula-2)
  - Make uniprocessor run-until-block threads
  - Add preemption
  - Add multiple processors

Types of Parallelism

- Data
- Functional (task)

Task Parallelism

- Each process performs a different task.
- Two principal flavors:
  - pipelines
  - task queues
- Program Examples: PIPE (pipeline), TSP (task queue).

Coordination and Communication

- Two fundamental models
  - Shared memory
  - Message passing

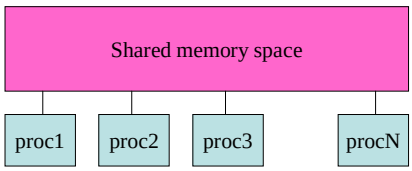
Synchronization

- Two basic types
  - Mutual exclusion
    - Ensure that only 1 thread is executing a region of code or accessing a data element at any given time
  - Condition synchronization
    - Blocks thread until a specific condition holds

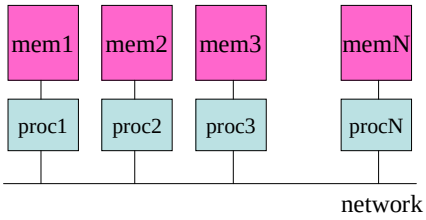
Components of a Synchronization Event

- Acquire method
  - Process tries to get the right to a sync operation (enter critical section, proceed past event)
- Waiting algorithm
  - Method used to wait for resource to become available
    - Busy-waiting/spinning
    - Blocking
- Release method
  - Enable other processes to proceed past a synchronization event

Shared Memory



Distributed Memory - Message Passing



Programming vs. Hardware

- One can implement
  - a shared memory programming model
  - on shared or distributed memory hardware
  - (also in software or in hardware)
- One can implement
  - a message passing programming model
  - on shared or distributed memory hardware

Parallel Processing Issues

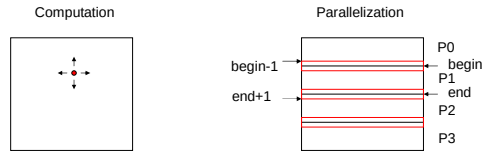
- Data sharing
- Process coordination
- Distributed (NUMA) vs. centralized (UMA) memory
- Connectivity
- Fault tolerance

Correctness and performance issues: deadlock, livelock, starvation

Example Program - Jacobi

```
for (k = 0; k < 100; k++) {

   for (j = 1; j < M-1; j++)
      for (i = 1; i < M-1; i++)
         a[j][i] = (b[j][i-1] + b[j][i+1] +
                   b[j-1][i] + b[j+1][i])/4;

   for (j = 1; j < M-1; j++)
      for (i = 1; i < M-1; i++)
         b[j][i] = a[j][i];
}
```

Example Jacobi Parallelization



a(i,j) = F(b(i-1,j),b(i+1,j),b(I,j-1),b(I,j+1))

Shared Memory Version of Jacobi

```
for (k = 0; k < 100; k++) {

    for (j = begin; j < end; j++)
        for (i = 1; i < M-1; i++)
            a[j][i] = (b[j][i-1] + b[j][i+1] +
                      b[j-1][i] + b[j+1][i])/4;
    barrier();

    for (j = begin; j < end; j++)
        for (i = 1; i < M-1; i++)
            b[j][i] = a[j][i];
    barrier();
}
```

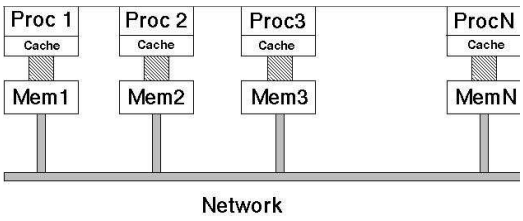
Message Passing Version of Jacobi

```
for (k = 0; k < 100; k++) {

    for (j = begin; j < end; j++)
        for (i = 1; i < M-1; i++)
            a[j][i] = (b[j][i-1] + b[j][i+1] + b[j-1][i] + b[j+1][i])/4;

    for (j = begin; j < end; j++)
        for (i = 1; i < M-1; i++)
            b[j][i] = a[j][i];

    if (p != (nprocs - 1)) {
        send(p+1, b[end-1]);
        recv(p+1, b[end]);
    }
    if (p != 0) {
        send(p-1, b[begin]);
        recv(p-1, b[begin-1]);
    }
}
```

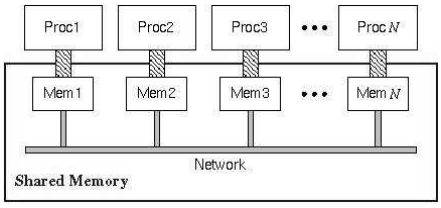
Data Parallel Version of Jacobi (Power C)

```
for (k = 0; k < 100; k++) {

#pragma parallel shared(a, b) local(i,j)
{
    #pragma pfor
    for (j = 1; j < M-1; j++)
        for (i = 1; i < M-1; i++)
            a[j][i] = (b[j][i-1] + b[j][i+1] +
                      b[j-1][i] + b[j+1][i])/4;
}

#pragma parallel shared(a, b) local(i, j)
{
    #pragma pfor
    for (j = 1; j < M-1; j++)
        for (i = 1; i < M-1; i++)
            b[j][i] = a[j][i];
}
}
```

Distributed Memory Hardware



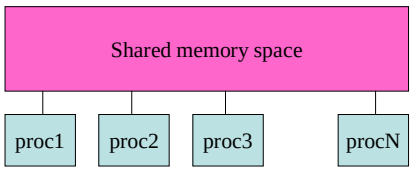
Software Distributed Shared Memory (S-DSM)



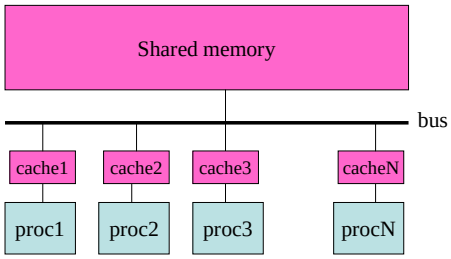
Why is Parallel Computing Hard?

- Amdahl's law - insufficient available parallelism
  - Speedup limited by part that is not parallelized
- Overhead of communication and coordination
- Portability - knowledge of underlying architecture often required

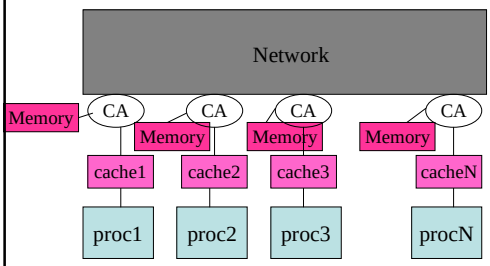
Shared Memory: A Look Underneath



Shared Memory Hardware (Snoopy-Based Coherence)



Shared Memory Hardware (Directory-Based NUMA)



Shared Memory Implementation

- Coherence - defines the behavior of reads and writes to the same memory location
  - ensuring that modifications made by a processor propagate to all copies of the data
  - Program order preserved
  - Writes to the same location by different processors serialized
- Synchronization - coordination mechanism
- Consistency - defines the behavior of reads and writes with respect to access to other memory locations
  - defines when and in what order modifications are propagated to other processors

Basic Hardware Mechanisms for Synchronization

- Test-and-set – atomic exchange
- Fetch-and-increment – returns value and atomically increments it
- Load-locked/store conditional – pair of instructions – deduce atomicity if second instruction returns correct value

Implementing Locks Using Test&Set

- On the SPARC ldstub moves an unsigned byte into the destination register and rewrites the same byte in memory to all 1s

```
_Lock_acquire:
    ldstub [%o0], %o1
    addcc %g0, %o1, %g0
    bne _Lock_acquire
    nop
fin:
    jmpl %r15+8, %g0
    nop
_Lock_release:
    st %g0, [%o0]
    jmpl %r15+8, %g0
    nop
```

Using ll/sc for Atomic Exchange

- Swap the content of R4 with the memory location specified by R1

```
try:  mov   R3, R4      ; mov exchange value
      ll    R2, 0(R1)   ; load linked
      sc    R3, 0(R1)   ; store conditional
      beqz  R3, try     ; branch if store fails
      mov   R4, R2      ; put load value in R4
```

Software Synchronization Algorithms

- Locks - test&test&set, ticket, array-based queue, MCS (linked list)
- Barriers – centralized/sense-reversing, software combining trees, tournament, dissemination
- Some desirable properties - Lock-free, non-blocking, wait-free

Sequential Consistency

- ``A system is sequentially consistent if the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program.'' [Lamport 79]
  - In practice, this means that every write must be seen on all processors before any succeeding read or write can be issued

Consistency Model Classification

- Models vary along the following dimensions
  - Local order - order of a processor's accesses as seen locally
  - Global order - order of a single processor's accesses as seen by each of the other processors
  - Interleaved order - order of interleaving of different processor's accesses on other processors

Code Improvement

- Peep-hole optimization
  - Short sequences of instructions
- Local optimization
  - Basic blocks
- Intra-procedural optimization
  - Across basic blocks but within a procedure/subroutine
- Inter-procedural optimization
  - Across procedures/subroutines

Elements of a Compiler

- Scanner (lexical analysis) (input – character stream, ouput – token stream)
- Parser (syntax analysis) (output – parse tree)
- Semantic analysis (output – AST with annotations)
- Machine-independent
  - Intermediate code generation (output – CFG with basic block pseudo-code)
  - Local redundancy elimination (output - modified CFG)
  - Global redundancy elimination (output - modified CFG)
  - Loop improvement (output - modified CFG)
- Machine-specific
  - Target code generation (output – assembly language)
  - Preliminary instruction scheduling (output – modified assembly)
  - Register allocation (output – modified assembly)
  - Final instruction scheduling (output – modified assembly)
  - Peephole optimization (output – modified assembly)

Peephole Optimization

- Constant folding – e.g., 3X2 → 6
- Constant propagation
- Elimination of redundant loads and stores
- Common subexpression elimination
- Copy propagation
- Strength reduction
- Load and branch delays
- Elimination of useless instructions
- Exploitation of instruction set

Redundancy Elimination

- Can be done at the basic block and intraprocedural level as well
- Requires data flow analysis
  - Static single assignment (SSA) form
  - Global value numbering
    - Assign the same name (number) to any two or more symbolically equivalent computations

Loop optimizations

- Loop invariants – move out of body
- Loop unrolling and software pipelining
  - Helps improve instruction scheduling
- Loop reordering
  - Requires dependence analysis
  - Can improve locality and parallelism