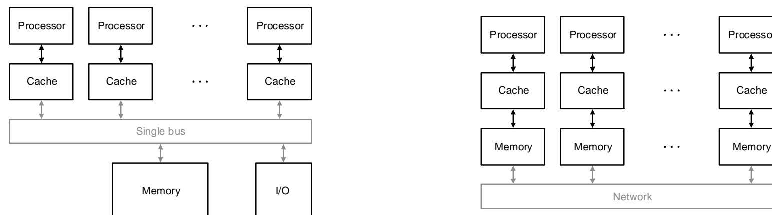
Chapter Nine

Multiprocessors

- **Idea: create powerful computers by connecting many smaller ones**

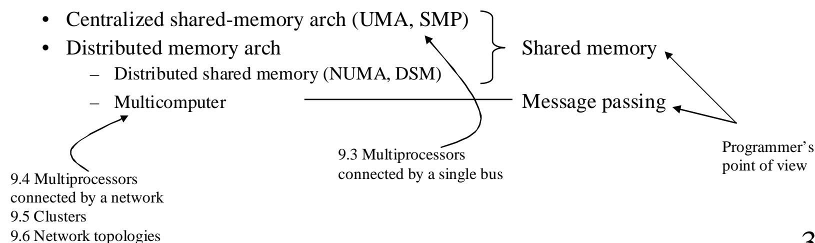
  good news: works for timesharing (better than supercomputer)
  vector processing may be coming back

  bad news: its really hard to write good concurrent programs
  many commercial failures



Flynn’s taxonomy (30 years ago)

- **SISD (Single instruction stream, single data stream) : uniprocessor**
- **SIMD (Single instruction stream, multiple data stream) :**
  - **early models**
  - **Single instruction memory and control processor with multiple PE (processing elements)**
- **MISD (Multiple instruction stream, single data stream) : no products**
- **MIMD (Multiple instruction stream, multiple data stream) :**
  - **offers flexibility in operations**
  - **MIMD can be built with off-the-shelf microprocessors**
  - **MIMD is sub-divided into**



MIMD Systems

- **Centralized shared-memory arch (UMA: uniform memory access): share a single memory interconnect by a bus which can satisfy several CPUs’ memory demand with large cache**

- **Distributed memory arch : memory is distributed among CPUs to support large # of processors, high bandwidth interconnect is required and interprocessor communication is complex**
  - **Distributed shared-memory : single shared address space with distributed memories (DSM, Scalable SM, NUMA – Shared memory arch is classified as UMA or NUMA)**
  - **Multi-computers : multiple address spaces and cannot be addressed by a remote CPU (Message passing machines uses simple network protocol such as RPC, MPI, PVM)**

Good and Bad

- **Shared-memory communication - compatibility, ease of programming, low overhead, h/w caching**
- **Message-passing communication – simpler h/w (no cache coherency), explicit comm programming**
- **Message-passing on top of shared memory – easy to implement (send is memory copy)**
- **Shared memory on top of message passing – all memory accesses need addr translation & make msg**
  (e.g.) Shared virtual memory : use of virtual memory mechanism to share objects at the page level

Questions

- **How do parallel processors share data?**
  — single address space (SMP vs. NUMA)
  — message passing
- **How do parallel processors coordinate?**
  — synchronization (locks, semaphores)
  — built into send / recieve primitives
  — operating system protocols
- **How are they implemented?**
  — connected by a single bus
  — connected by a network

9.3 Multiprocessors Connected by a Single Bus

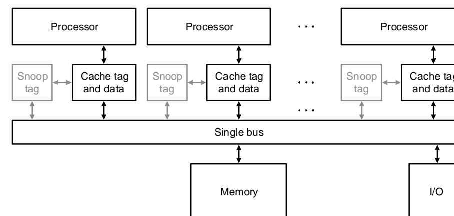
- Fig. 9.3 in page 718
- Example in page 719

Centralized Shared-Memory Architecture

- Inexpensive microprocessors with large cache motivate small-scale multiprocessors
- However, caching of a shared data makes trouble
- What happens when a program running in processor 1 loads data from a memory location that was written (into cache) by processor 2?
  - Cache coherence problem
  - In a system with private caches and shared memory, multiple cached copies of a shared data block must be kept consistent with each other.
- Simplest solution is to prohibit caching of shared data
- Snooping:
  - Cache-coherence protocols track the state of any sharing of a data block to maintain coherent caches

Some Interesting Problems

- **Cache Coherency**



- **Synchronization**
  — provide special atomic instructions (test-and-set, swap, etc.)

Snoop-based Cache Coherence Protocol

- **Fig. 9.4 in page 720**
- **Each cache has the sharing state of the blocks it has, cache controllers monitor (snoop) on the shared-memory bus to determine whether or not they have the requested block**
  - **Write invalidate protocol : invalidates other copies on a write (exclusive access to the data)**
  - **Write update (broadcast) protocol : updates all the cached copies on a write (reading is not a problem)**
- **Implementation Techniques**
  - **Invalidation : The processor acquires bus access and broadcasts the address on write hit -> Other processors continuously snoop the bus and invalidates itself**
  - **Location of data item on cache miss**
    - write-through cache : it simply is in main memory (but, write buffer makes trouble)
    - write-back cache : a processor who has the dirty block snoops and responds and cancel memory access (requires lower memory bw -> choice for multiprocessors)

Write-invalidate in Snoop Protocol

- P1 & P2 have X in their caches
- P1 write X
  - P1 broadcast the address on the bus (change its state to "E")
  - P2 snoops the bus and invalidate the copy in its cache
- P3 reads X
  - P3 broadcast the address on the bus (to read)
  - Memory responds
  - P1 snoops the bus, cancels the memory response and respond itself
  - P1 changes its state to "S" and P3 set its state to "S"

ESI Protocol

- State of cache blocks
  - (I) Invalid – Not a valid cache block
  - (S) Shared – Read only (May be shared by more than one processors)
  - (E) Exclusive – Read/Write (Exclusive copy)

- Local processor activity

Activity	Current status => I	S	E
write hit	N/A	S->E (invalidate others)	E->E
write miss	I->E	N/A	N/A
read hit	N/A	S->S	E->E
read miss	I->S	N/A	N/A
block replacement	N/A	E->I	S->I (write back)

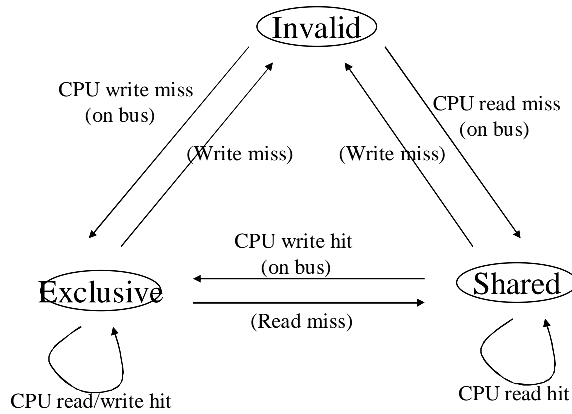
- Snoop activity – send requested block & what ???

Bus action	My block status => S (Shared)	E (Exclusive)
Write hit	Invalidate itself	N/A
Write miss	Invalidate itself	Write back*, invalidate itself
Read hit	N/A	N/A
Read miss	No operation	Write back, change to "S"

- *: Why write-back when it will be written anyway? <= other data in the same block !!!

- Lock pingponging problem – two processes access the same data -> invalidate continuously w/o modify
- False sharing : miss due to invalidation, but they in fact require different words in the same cache block (theses misses will not occur if block size is 1word)

ESI Protocol



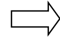
MESI

- **Many modern processors adopt this**
  - **Pentium, PowerPC**
  - **Why do they implement cache coherence protocol?**
  - **What does it mean “they implement”, what is really implemented inside the core?**
- **States**
  - **I : Invalid**
  - **S : Shared (shared, clean)**
  - **E : Exclusive (private, clean)**
  - **M : Modified (private, dirty) = E in ESI**
- **What’s the benefit of having two separate states (S and E) in MESI instead of one (S) in ESI?**

Synchronization

- How to access shared data (X) without interference
- Lock variable or semaphore to gain the access right (critical section)
- Lock variable in [100] = 0 (free to access), 1 (wait until released)
  - Assume $0=0, $1=1
  - (While [100]==1) ;
    access "X"
- Assembly version

```
wait:
    lw      $2, [100]
    beq     $2, $1, wait
    sw      $1, [100]
    "access X"
    sw      $0, [100]
```



```
Initially [100]=0 (unlocked)
P1: lw   $2, [100]
                          P2: lw   $2, [100]
                              beq  $2, $1, wait
                              sw   $1, [100]
                              "access X"
    beq  $2, $1, wait
    sw   $1, [100]
    "access X"
```

Both access "X"

Synchronization

- Needs uninterruptible instruction or instruction sequence capable of "atomically" retrieving or changing a value
  - Examples: "swap", "ll-sc", "test&set"
- Solution using "swap"

```
    add     $s3, $s1, $s0
wait:
    swap    $3, [100]
    beq     $3, $1, wait
    "access X"
    sw      $0, [100]
```

```
Initially, [100]=0 (unlocked)
P1: swap  $3, [100]
                          P2: swap $3, [100]
```

$3=0 & no branch

$3=1 & branch to wait

Synchronization problem solved

- But, it is the solution for uniprocessor system

Synchronization

- In multiprocessor systems,

```
add  $s3, $s1, $s0          P1:  swap  $3, [100]
wait:                             ; write miss, I->E
   swap    $3, [100]                      P2: swap $3, [100]
   beq     $3, $1, wait                       ; write miss, I->E (P1: E->I)
   "access X"                                 beq & swap
   sw      $0, [100]                          ; write hit
                                              beq & swap
                                              ; write hit
```

- Seems no problem, but what if we have P1, P2 and P3 try to access X

```
                                 P2: swap ; write miss, I->E (P1: E->I)
P3: swap ; write miss, I->E (P2: E->I)
                                 P2: swap ; write miss, I->E (P3: E->I)
P3: swap ; write miss, I->E (P2: E->I)
```

→ Each write miss incurs write-back, too.

Synchronization

- A better solution

```
    add   $3, $1, $0
wait:
    lw    $4, [100]
    beq   $4, $1, wait      => Testing first before swaping
    swap $3, [100]          State changes are mostly I<->S instead of I<->E
    beq   $3, $1, wait      Gives a lot of bandwidth savings
    "access X"              => A problem may happen when P1 releases
    sw    $0, [100]            the lock => race condition ...
```

Memory Consistency Problem

- What happens if two processors try to write to the same shared data word in the same clock cycle? In what order, are instructions executed?

  P1: A=0 P2: B=0
  A=1 B=1
  if (B==0)... If (A==0) ...
- SC (Sequential Consistency) requires that the result of any execution be the same as if the accesses executed by each processor were kept in order and the accesses among different processors were arbitrarily interleaved.

  => the two conditions (B==0 and A==0) cannot be satisfied all together

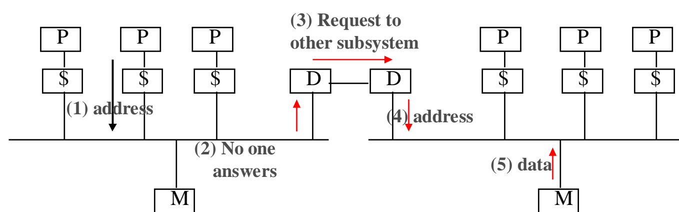
  => But, it is possible to satisfy both. How?

Memory Consistency Problem

- Out-of-order execution model in modern microprocessor
  - "sw": let it go and immediately proceed to the next instruction without waiting until the sw is completed.
  - "lw": can't let it go because the next instruction will likely depend on lw. But just let it go and handle it later.
- Scenario
  - When A=1 is not finished, P1 can proceed to check B==0, which seems independent with the previous instruction
  - At the same time, when B=1 is not finished, P2 can proceed to check A==0, which seems also independent
  - Both conditions are satisfied !!!
- Do we allow it?
  - SC does not allow it happen: the most conservative
  - Weaker model: WO (weak ordering), RC (release consistency)
  - RC is the model that can be combined with dynamically scheduled processors and aggressive compiler optimizations

9.4 Multiprocessors Connected by a Network

- Fig. 9.9 in page 728
- Distributed shared memory (NUMA) or CC-NUMA
  - Directory contains entries with presence and modified bits
  - Directory keeps track of every block that may be cached



Other types

- Multi-computers
  - Message-passing computers
  - MPI (Message-Passing Interface): next slides

MPI (Message Passing Interface)

- History
  - Began at Williamsburg Workshop in April, 1992
  - Organized at Supercomputing '92 (November) => meeting, email, drafts, votes
  - Pre-final draft at Supercomputing '93 with 2-month public comment period
  - Final version of draft in May, 1994

- MPI is a message-passing library specification
  - message-passing model
  - not a compiler specification nor a specific product

- MPI Lacks
  - Mechanism for process creation (fixed number of processes from start to finish)
  - One sided communication (put, get, active messages)
  - Language binding for Fortran 90 and C++

New Features of MPI

- General
  - communicators combine context and group for message security
  - thread safety
- Point-to-point communication
  - structured buffers and derived datatypes, heterogeneity
  - modes: normal (blocking/non-blocking), synchronous, ready (to allow access to fast protocols), buffered
- Collective
  - both built-in and user-defined collective operations
  - large number of data movement routines
  - subgroups defined directly or by topology
- Application-oriented process topologies
  - built-in support for grids and graphs (uses groups)
- Profiling
  - hooks allow users to intercept MPI calls to install their own tools

More History

- MPI-1
  - May, 1994MPI 1.0 Publication
  - June, 1995 MPI 1.1 Enhanced Publication
  - July, 1997 MPI 1.2 Enhanced Publication

- MPI-2
  - Nov., 1996 Draft opened at Supercomputing ‘96
  - Jan.-May, 1997 Public comment
  - July, 1997 MPI-2 Publication

Communication and Computation Model

- “Communicator” is a communication domain
  - intra-communicator, inter-communicator
  - MPI_COMM_WORLD : default intra-communicator

- SPMD model : same program on master and slave

```
main(int argc, char *argv[])
{
        MPI_Init (&argc, &argv);
        ….
        MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
        if (myrank == 0)
                master();
        else
                slave();
        ….
        MPI_Finalize();
}
```

MPI APIs (6 basic, total 125)

- Initializing MPI
  - MPI_Init(int *argc, char **argv)
- MPI_COMM_WORLD Communicator
  - MPI_Comm_rank(MPI_Comm comm, int *rank) : identify
  - MPI_Comm_size(MPI_Comm comm, int *size) : number of processes
- Exiting MPI
  - MPI_Finalize()
- Point-to-point communication
  - MPI_Ssend(void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm)
  - MPI_Recv(void *buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Status *status)
  - MPI_Issend(buf, count, datatype, dest, tag, comm, handle)
  - MPI_Irecv(bufm count, datatype, src, tag, comm, handle)

Communication Modes

- Sender mode
  - standard mode (MPI_Send)
    - if buffering is provided, it can complete before the receiver is reached
  - buffered mode (MPI_Bsend)
    - always completes irrespective of receiver
    - need to supply buffer space by MPI_Buffer_attach()
  - synchronous mode (MPI_Ssend)
    - only completes when the receive has started
  - ready send (MPI_Rsend)
    - send can only start if the receiver has already been reached

Message Passing

- Blocking routines
  - "blocking" routine returns after their local actions complete, though the message transfer may not have been completed (locally complete)
  - MPI_Send()
  - MPI_Recv() : the message has been received in to the destination location
- Non-blocking routines
  - "non-blocking" routine returns immediately assuming the data storage being used for the transfer is not modified
  - provide the ability to overlap communication and computation (very important if communication delay is large)
  - MPI_Isend(…,req1) : returns even before the source location is safe to be altered
  - MPI_Irecv() : returns even if there is no message to accept
  - Completion can be detected by separate routines
    - MPI_Wait(req1,...) : returns when the operation has actually completed
    - MPI_Test() : returns immediately with a flag

Non-Blocking Communication

- Send/receive
  - standard : blocking/non-blocking send/receive
  - buffered, synchronous, ready : blocking/non-blocking send
- A blocking send can be used with a non-blocking receive and vice versa
- Three phases of non-blocking operation
  - initiate communication (MPI_Issend or MPI_Irecv)
  - do some work
  - wait for communication to complete : MPI_Wait(handle, status)
- Types
  - MPI_Isend (buf, count, datatype, src, tag, comm, handle)
  - MPI_Issend : synchronous send
  - MPI_Ibsend : buffered send
  - MPI_Irsend : ready send
  - MPI_Irecv : receive

Collective Communication

- **Characteristics**
  - **Collective action over a communicator**
  - **all processes must communicate**
  - **synchronization may or may not occur**
  - **all collective operations are blocking**
  - **no tags**
  - **receive buffers must be exactly the right size**
- **Types**
  - **Barrier : MPI_Barrier (MPI_Comm comm)**
  - **Broadcast : MPI_Bcast(void *buffer, int count, MPI_datatype, int root, MPI_Comm comm)**
  - **Scatter, gather, reduction : (ex) MPI_Reduce (&x, &result, 1, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD) : sum of all "x" values and save to "result"**

Pi

```
#include "mpi.h"
#include <math.h>

double f(a)
double a;
{
   return (4.0 / (1.0 + a*a));
}

int main(argc,argv)
int argc;
char *argv[];
{
   int done = 0, n, myid, numprocs, i, rc;
   double PI25DT =
       3.141592653589793238462643;
   double mypi, pi, h, sum, x, a;
   double startwtime, endwtime;

   MPI_Init(&argc,&argv);

       MPI_Comm_size(MPI_COMM_WORLD,&
       numprocs);

       MPI_Comm_rank(MPI_COMM_WORLD,&
       myid);
```

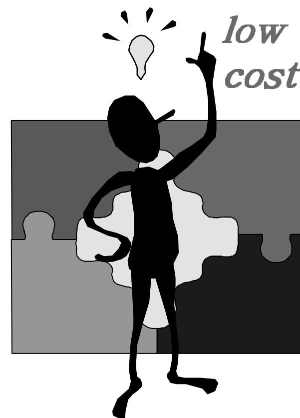
```
n = 0;
  while (!done)
  {
    if (myid == 0)
    {
      if (n==0) n=100; else n=0;
      startwtime = MPI_Wtime();
    }
    MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);
    if (n == 0)
      done = 1;
    else
    {
      h   = 1.0 / (double) n;
      sum = 0.0;
      for (i = myid + 1; i <= n; i += numprocs)
      {
        x = h * ((double)i - 0.5);
        sum += f(x);
      }
      mypi = h * sum;

      MPI_Reduce(&mypi, &pi, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);

      if (myid == 0)
      {
        printf("pi is approximately %.16f, Error is %.16f\n",
            pi, fabs(pi - PI25DT));
        endwtime = MPI_Wtime();
        printf("wall clock time = %f\n",
            endwtime-startwtime);
      }
    }
  }
  MPI_Finalize();
}
```

9.5 Clusters

- Fig. 9.12 in page 735
- Next slides

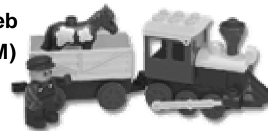
Why Clusters

- **Two Key Points**
  - **Microprocessors kept on getting faster. A lot faster.**
    - No time to research and develop proprietary technologies
    - Lego game with COTS & "Linux"
  - **System availability became a mass-market issue.**
    - Cluster is an interconnected "whole computers"
    - They can be easily isolated and replaceable



Lego Blocks Available

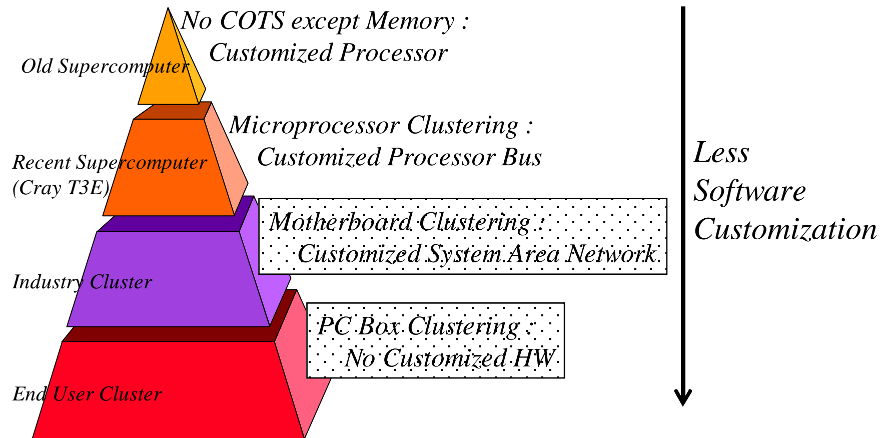
- Fat boxes : very high performance micros
  - no more supercomputers, but there is supercomputing on aggregates of micros
- Fat pipes (SAN): standard high-speed comm.
  - Fibre channel standard (FCS), ATM, SCI, Switched gigabit ethernet, Myrinet
- Thick glue : standard tools for dist. comp.
  - Internet protocols (TCP/IP, UDP/IP) => Web
  - Standard parallel programming (MPI, PVM)



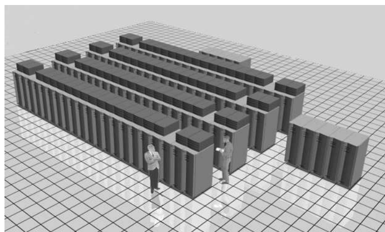
Enabling Technology - Linux

- Open source code
  - being developed in an open and accessible manner
  - easier to identify and fix the problems
  - occasional bug is corrected immediately
- Functionality
  - robust
  - largely POSIX-compliant
  - superior networking performance
  - immediate driver support for a new hardware
  - GNU's compilers and debuggers
  - available for Intel x86, DEC Alpha and PowerPC
- "Extreme Linux" for Clusters

Levels of Clustering



US DOE ASCI Platforms



The ASCI Red Supercomputer
- @Sandia National Lab.
- 4,536 Nodes
- 1.8 Tflops

The ASCI Blue Supercomputer

~~(3 Major HPC Efforts in USA)~~

- **DOE ASIC Program : ~$250 M/year**
  - **TFLOP/s level computing**
- **DOD Modernization : ~$200 M/year**
  - **Foremost academic and industry research**
- **NSF Supercomputer Centers : ~$65 M/year**
  - **University of Illinois - NCSA**
  - **University of California, San Diego - SDSC**

Japan's Earth Simulator

- **National Space Development Agency of Japan (NASDA) Earth Simulator Research and Development Center (ESRDC)**
- **At least 5TFLOPS**
- **640 nodes, 5120 processors, 10TB**

Los Alamos National Lab's Avalon Cluster

- **140 Alpha Nodes (128MB, 3GB)**
  - **Alpha PC 164LX Motherboard**
  - **128MB SDRAM, 3GB Disk**
  - Linux RedHat 5.0
  - **3Com SuperStack II 3900 36-port Fast Ethernet**
  - **Cyclades Cyclom 32-YeP serial concentrators**
- **10 Gflops for $150k**
  - **Submitted for the 1998 Gordon Bell Price/ Perf. Prize with 70 nodes**
  - **Avalon ranks at #113 on Top500 Supercomputers List (Nov.,'98)**

  (SERI Cray T3E $6000k, #78)

Los Alamos National Lab's Loki Cluster

- **Beowulf-class supercomputer built from commodity components.**

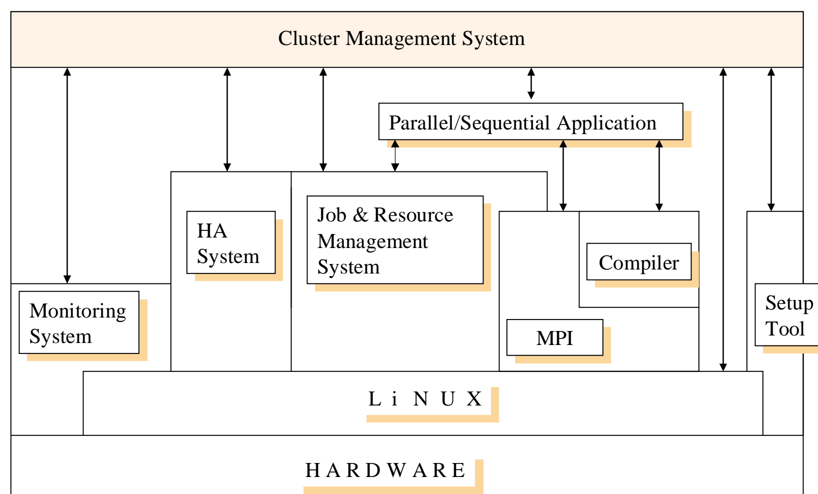
16 Pentium Pro Processors
x 5 Fast Ethernet interfaces
+ 2 Gbytes RAM
+ 50 Gbytes Disk
+ 2 Fast Ethernet switches
+ Linux

= 1.2 real Gflops for $63,000

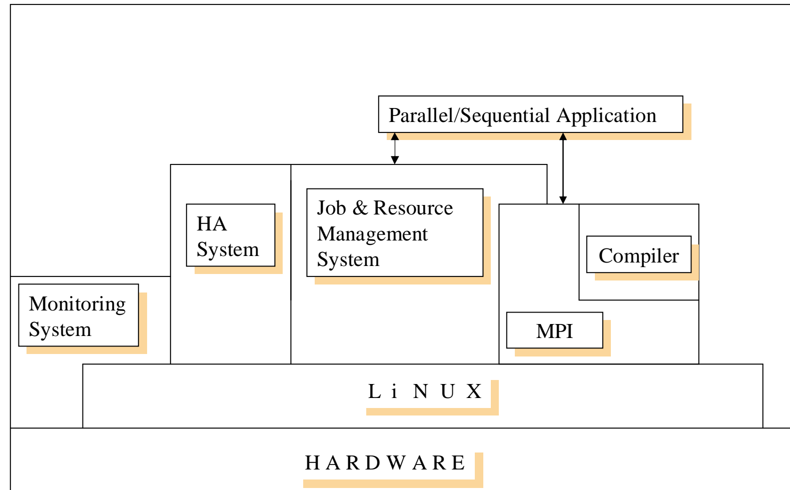
RWC PC Cluster II

- **Real World Computing Partners (Japan)**
  - **Developed at October 1997**
  - **64 Computation PCs**
    - Intel Pentium Pro 200 MHz
    - Memory 256 MB(ECC)
    - Hard Disk 2GB
    - Linux 2.1.119
    - Myrinet (LANai 4.1) 160MB/s
  - **2 Monitor PCs**
    - Consoles of PCs
    - 32 ch. Serial Lines for console
    - Keyboard + Display + Hard Disk

ABC Software Arch. (Server)



ABC Software Arch. (Client)



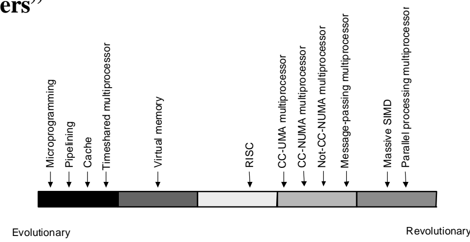
9.6 Network Topologies

- **Fig. 9.13 in page 738**
- **Fig. 9.14 in page 739**

Concluding Remarks

- **Evolution vs. Revolution**

"More often the expense of innovation comes from being too disruptive to computer users"



"Acceptance of hardware ideas requires acceptance by software people; therefore hardware people should learn about software. And if software people want good machines, they must learn more about hardware to be able to communicate with and thereby influence hardware engineers."