MIMD Machine
von Neumann Architecture

- Also known as "stored-program computer" - both program instructions and data are kept in electronic memory.
- Differs from earlier computers which were programmed through "hard wiring".
- Since then, virtually all computers have followed this basic design:
von Neumann Architecture

• Comprised of four main components:
  – Memory
  – Control Unit
  – Arithmetic Logic Unit
  – Input/Output

• Read/write, random access memory is used to store both program instructions and data
  – Program instructions are coded data which tell the computer to do something
  – Data is simply information to be used by the program

• Control unit fetches instructions/data from memory, decodes the instructions and then sequentially coordinates operations to accomplish the programmed task.

• Arithmetic Unit performs basic arithmetic operations
• Input/Output is the interface to the human operator
Parallel computers still follow this basic von Neumann design, just multiplied in units. The basic, fundamental architecture remains the same.
Flynn's Classical Taxonomy

- SISD: Single Instruction stream, Single Data stream
- SIMD: Single Instruction stream, Multiple Data stream
- MISD: Multiple Instruction stream, Single Data stream
- MIMD: Multiple Instruction stream, Multiple Data stream
SISD

load A
load B
C = A + B
store C
A = B * 2
store A

UNIVAC1
IBM 360
CRAY1

CDC 7600
PDP1
Dell Laptop
SIMD

SIMD
- Data Pool
- Instruction Pool
  - PU

prev instruct
- load A(1)
- load B(1)
- C(1)=A(1)*B(1)
- store C(1)
- next instruct

prev instruct
- load A(2)
- load B(2)
- C(2)=A(2)*B(2)
- store C(2)
- next instruct

prev instruct
- load A(n)
- load B(n)
- C(n)=A(n)*B(n)
- store C(n)
- next instruct

P1
<table>
<thead>
<tr>
<th>time</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
</tbody>
</table>

P2
<table>
<thead>
<tr>
<th>time</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
</tbody>
</table>

Pn
<table>
<thead>
<tr>
<th>time</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
</tbody>
</table>

ILLIAC IV
MasPar

Cray X-MP
Cray Y-MP
Thinking Machines CM-2
Cell Processor (GPU)
A SIMD Application

Scalar vs. SIMD Vector Addition
MISD

Figure 1. Basic principle of a systolic system.
A MISD Like Application: File Compression

Figure 3. File compression algorithm executed using pipeline parallelism
MIMD

IBM POWER5

HP/Compaq Alphaserver

Intel IA32

AMD Opteron

Cray XT3

IBM BG/L
Hyperquicksort
Amdahl's Law states that potential program speedup is defined by the fraction of code ($P$) that can be parallelized:

$$\text{speedup} = \frac{1}{1 - P}$$

- If none of the code can be parallelized, $P = 0$ and the speedup = 1 (no speedup).
- If all of the code is parallelized, $P = 1$ and the speedup is infinite (in theory).
- If 50% of the code can be parallelized, maximum speedup = 2, meaning the code will run twice as fast.
Amdahl's Law

- Introducing the number of processors performing the parallel fraction of work,

\[
\text{speedup} = \frac{1}{P + S}
\]

where \( P \) = parallel fraction, \( N \) = number of processors and \( S \) = serial fraction.

- It soon becomes obvious that there are limits to the scalability of parallelism.

<table>
<thead>
<tr>
<th>( N )</th>
<th>( P = .50 )</th>
<th>( P = .90 )</th>
<th>( P = .99 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>1.82</td>
<td>5.26</td>
<td>9.17</td>
</tr>
<tr>
<td>100</td>
<td>1.98</td>
<td>9.17</td>
<td>50.25</td>
</tr>
<tr>
<td>1,000</td>
<td>1.99</td>
<td>9.91</td>
<td>90.99</td>
</tr>
<tr>
<td>10,000</td>
<td>1.99</td>
<td>9.91</td>
<td>99.02</td>
</tr>
<tr>
<td>100,000</td>
<td>1.99</td>
<td>9.99</td>
<td>99.90</td>
</tr>
</tbody>
</table>
Amdahl’s Law and Scalability

• Certain problems demonstrate increased performance by increasing the problem size. For example:

<table>
<thead>
<tr>
<th></th>
<th>Original</th>
<th>New</th>
</tr>
</thead>
<tbody>
<tr>
<td>2D Grid Calculations</td>
<td>85 seconds</td>
<td>680 seconds</td>
</tr>
<tr>
<td>Serial fraction</td>
<td>15 seconds</td>
<td>15 seconds</td>
</tr>
</tbody>
</table>

• We can increase the problem size by doubling the grid dimensions and halving the time step.

<table>
<thead>
<tr>
<th></th>
<th>Original</th>
<th>New</th>
</tr>
</thead>
<tbody>
<tr>
<td>2D Grid Calculations</td>
<td>85 seconds</td>
<td>680 seconds</td>
</tr>
<tr>
<td>Serial fraction</td>
<td>15 seconds</td>
<td>15 seconds</td>
</tr>
</tbody>
</table>

Problems that increase the percentage of parallel time with their size are more scalable than problems with a fixed percentage of parallel time.
Complexity

• In general, parallel applications are much more complex than corresponding serial applications, perhaps an order of magnitude.
• Not only do you have multiple instruction streams executing at the same time, but you also have data flowing between them.
• The costs of complexity are measured in programmer time in virtually every aspect of the software development cycle:
  – Design
  – Coding
  – Debugging
  – Tuning
  – Maintenance
• Adhering to "good" software development practices is essential when working with parallel applications - especially if somebody besides you will have to work with the software.
Portability

• Thanks to standardization in several APIs, such as MPI, POSIX threads, and OpenMP, portability issues with parallel programs are not as serious as in years past.
• All of the usual portability issues associated with serial programs apply to parallel programs.
  – For example, if you use vendor "enhancements" to Fortran, C or C++, portability will be a problem.
• Even though standards exist for several APIs, implementations will differ in a number of details, sometimes to the point of requiring code modifications in order to effect portability.
• Operating systems can play a key role in code portability issues.
• Hardware architectures are characteristically highly variable and can affect portability.
Resource Requirements

• The amount of memory required can be greater for parallel codes than serial codes
  – need to replicate data and there are overheads associated with parallel support libraries and subsystems.

• For short running parallel programs, there can actually be a decrease in performance compared to a similar serial implementation.
  – The overhead costs associated with setting up the parallel environment, task creation, communications and task termination can comprise a significant portion of the total execution time for short runs.
Scalability

• Two types of scaling based on time to solution:
  – Strong scaling: The total problem size stays fixed as more processors are added.
  – Weak scaling: The problem size per processor stays fixed as more processors are added.
• The ability of a parallel program's performance to scale is a result of a number of interrelated factors.
  – Simply adding more processors is rarely the answer; the algorithm may have inherent limits to scalability.
Scalability

• Hardware factors play a significant role in scalability. Examples:
  – Memory-cpu bus bandwidth on an SMP machine
  – Communications network bandwidth
  – Amount of memory available on any given machine or set of machines
  – Processor clock speed

• Parallel support libraries and subsystems software can limit scalability independent of your application.
Parallel Computer Memory Architectures

**Shared Memory**

- **General Characteristics:**
  - Shared memory parallel computers vary widely, but generally have in common the ability for all processors to access all memory as global address space.
  - Multiple processors can operate independently but share the same memory resources.
  - Changes in a memory location effected by one processor are visible to all other processors.
  - Historically, shared memory machines have been classified as *UMA* and *NUMA*, based upon memory access times.

- **Uniform Memory Access (UMA):**
  - Most commonly represented today by *Symmetric Multiprocessor (SMP)* machines
  - Identical processors
  - Equal access and access times to memory
  - Sometimes called CC-UMA - Cache Coherent UMA. Cache coherent means if one processor updates a location in shared memory, all the other processors know about the update. Cache coherency is accomplished at the hardware level.
Parallel Computer Memory Architectures

Shared Memory

Non-Uniform Memory Access (NUMA):
- Often made by physically linking two or more SMPs
- One SMP can directly access memory of another SMP
- Not all processors have equal access time to all memories
- Memory access across link is slower
- If cache coherency is maintained, then may also be called CC-NUMA
  - Cache Coherent NUMA
Parallel Computer Memory Architectures

Shared Memory

• Advantages:
  – Global address space provides a user-friendly programming perspective to memory
  – Data sharing between tasks is both fast and uniform due to the proximity of memory to CPUs

• Disadvantages:
  – Primary disadvantage is the lack of scalability between memory and CPUs.
    • Adding more CPUs can geometrically increases traffic on the shared memory-CPU path
    • For cache coherent systems, geometrically increase traffic associated with cache/memory management.
  – Programmer responsibility for synchronization constructs that ensure "correct" access of global memory.
Example: Sum Reduction

• Sum 100,000 numbers on 100 processor UMA
  – Each processor has ID: $0 \leq P_n \leq 99$
  – Partition 1000 numbers per processor
  – Initial summation on each processor
    \[
    \text{sum}[P_n] = 0;
    \text{for} \; (i = 1000*P_n; \; i < 1000*(P_n+1); \; i = i + 1)
    \text{sum}[P_n] = \text{sum}[P_n] + A[i];
    \]

• Now need to add these partial sums
  – Reduction: divide and conquer
  – Half the processors add pairs, then quarter, ...
  – Need to synchronize between reduction steps
Example: Sum Reduction

half = 100;
repeat
    synch();
    if (half%2 != 0 && Pn == 0)
        sum[0] = sum[0] + sum[half-1];
        /* Conditional sum needed when half is odd;
            Processor0 gets missing element */
    half = half/2; /* dividing line on who sums */
    if (Pn < half) sum[Pn] = sum[Pn] + sum[Pn+half];
until (half == 1);
Cache Coherence

- Each CPU has a private cache while sharing the main memory
- What happens if one CPU writes a value to an address that is cached in other CPUs?
Solving the cache coherence problem using MSI protocol
Cache Coherence Demo
Distributed Memory

General Characteristics:

- Like shared memory systems, distributed memory systems vary widely but share a common characteristic. Distributed memory systems require a communication network to connect inter-processor memory.

- Processors have their own local memory. Memory addresses in one processor do not map to another processor, so there is no concept of a global address space across all processors.
- Because each processor has its own local memory, it operates independently. Changes it makes to its local memory have no effect on the memory of other processors. Hence, the concept of cache coherency does not apply.
- When a processor needs access to data in another processor, it is usually the task of the programmer to explicitly define how and when data is communicated. Synchronization between tasks is likewise the programmer's responsibility.
- The network "fabric" used for data transfer varies widely, though it can be as simple as Ethernet.
Parallel Computer Memory Architectures

Distributed Memory

• Advantages:
  – Memory is scalable with the number of processors. Increase the number of processors and the size of memory increases proportionately.
  – Each processor can rapidly access its own memory without interference and without the overhead incurred with trying to maintain global cache coherency.
  – Cost effectiveness: can use commodity, off-the-shelf processors and networking.

• Disadvantages:
  – The programmer is responsible for many of the details associated with data communication between processors.
  – It may be difficult to map existing data structures, based on global memory, to this memory organization.
  – Non-uniform memory access times - data residing on a remote node takes longer to access than node local data.
Sum Reduction (Again)

- Sum 100,000 on 100 processors
- First distribute 100 numbers to each
  - The do partial sums
    
    ```
    sum = 0;
    for (i = 0; i<1000; i = i + 1)
      sum = sum + AN[i];
    ```

- Reduction
  - Half the processors send, other half receive and add
  - The quarter send, quarter receive and add, ...
Sum Reduction (Again)

- Given `send()` and `receive()` operations
  
  ```
  limit = 100; half = 100;/* 100 processors */
  repeat
    half = (half+1)/2; /* send vs. receive dividing line */
    if (Pn >= half && Pn < limit)
      send(Pn - half, sum);
    if (Pn < (limit/2))
      sum = sum + receive();
    limit = half; /* upper limit of senders */
  until (half == 1); /* exit with final sum */
  ```

- Send/receive also provide synchronization
- Assumes send/receive take similar time to addition
Parallel Programming Models

- Shared Memory (without threads)
- Threads
- Distributed Memory / Message Passing
- Data Parallel
- Hybrid

*Note: each of these models are somewhat independent from the hardware architecture*
Shared Memory (No Threads)

- Processes/tasks share a common address space, which they read and write to asynchronously.
- Various mechanisms such as locks / semaphores are used to control access to the shared memory, resolve contentions and to prevent race conditions and deadlocks.
- All processes see and have equal access to shared memory. Program development can often be simplified.
- An important disadvantage: it becomes more difficult to understand and manage data locality:
  - Keeping data local to the process that works on it conserves memory accesses, cache refreshes and bus traffic that occurs when multiple processes use the same data.
  - Unfortunately, controlling data locality is hard to understand and may be beyond the control of the average user.
Shared Memory (No Threads)
Shared Memory (Threads)

- In the threads model of parallel programming, a single "heavy weight" process can have multiple "light weight", concurrent execution paths.
Example of Threads

- The main program `a.out` is scheduled to run by the operating system. `a.out` acquires all the necessary resources to run: a "heavy weight" process.
- `a.out` performs some serial work, and then creates a number of tasks (threads) that can run concurrently.
- Each thread has local data, but also shares the entire resources of `a.out`.
- Threads communicate with each other through global shared memory, hence, need synchronization.
- Threads can come and go, but `a.out` remains present to provide the necessary shared resources until the application has completed.
From the Dinosaur’s Book

Single-threaded process

Multithreaded process
Distributed Memory / Message Passing Model

• A set of tasks that use their own local memory during computation. Multiple tasks can reside on the same physical machine and/or across an arbitrary number of machines.

• Tasks exchange data through communications by sending and receiving messages.

• Data transfer usually requires cooperative operations to be performed by each process.
  – For example, a send operation must have a matching receive operation.
Message-Passing Model
Data Parallel Model

- Address space is treated globally
- Most of the parallel work focuses on performing operations on a data set. The data set is typically organized into a common structure, such as an array or cube.
- A set of tasks work collectively on the same data structure, however, each task works on a different partition of the same data structure.
- Tasks perform the same operation on their partition of work, for example, "add 4 to every array element"
Data Parallel Model

```plaintext
...  
do i=1,25  
A(i)=B(i)*delta  
end do  
...  

do i=26,50  
A(i)=B(i)*delta  
end do  
...

do i=m,n  
A(i)=B(i)*delta  
end do  
...  
```

task 1  
task 2  
.....  
task n
Hybrid Model

- Threads perform computationally intensive kernels using local, on-node data
- Communications between processes on different nodes occurs over the network using MPI