Logistics

• Any questions about the lecture or homework?
• Bin will be in the lab Thursday.
  ➢ I may be there for a short time!
• Office hours are also by appointment.
  ➢ Thursday at 2pm is the normal office hour
  ➢ Do not expect immediate access.
    ▪ There are 21 other students!
• Make sure you watch the class messages board.
  ➢ Fixes or info important to the class is posted there as it happens.
• Install my ssh keys.
  ➢ It you need help with makefiles etc.
Scalable Design Principles

• Based on lessons learned from successful and failed
  ➢ Parallel computer designs
  ➢ Parallel application designs
• Both computers and applications are complex engineering processes
• Principle of independence
• Principle of balanced design
• Principle of design for scalability
• Principle of latency hiding.
Principle of Independence

- **System Components should be independent**
  - Not really possible
  - Interdependence should be
    - Minimized
    - Clearly defined and understood 😊
- **Allows for independent scaling of sub-systems**
  - Sometimes called incremental scaling
- **Commodity components are available for most**
  - Weak link is the network.
- **Understand the difference between the Architecture and Implementation**
- **Use Standard components (Hardware/Software).**
Principle of Independence Examples

• The algorithm should be independent of the architecture
  ➢ Not always possible

• The application should be portable; not dependent upon the platform

• The Programming Model should be independent of the Machine
  ➢ Mostly possible
  ➢ Modularity/independence important in SW

• Node should be independent of the network

• The network interface should not depend on the network topology
Principle of Balanced Design

- Minimize any performance bottlenecks.
- Avoid any single point of failure
  - The SMP “problem”
- The 50% rule
  - Each subsystem/overhead can only degrade the performance (limit) by 50%
    - Load imbalance
    - Parallelism Overhead
    - Communication Start-up
    - Per-byte communication (reciprocal bandwidth)
Amdahl’s Law (Fallacy)

- **Two parts X and Y**
  - X can be parallelized or improved
  - Y is the sequential bottle neck
  - Y = 1-X or 100-X percent

\[
S = \frac{1}{(X/Np)\% + Y\%} = \frac{1}{Y}
\]
Amdahl's Law

Speed Up

Number of Processors

0 4 8 12 16

PP 50%
PP 75%
PP 80%
PP 90%
PP 99.9%
PP 99.99%
PP 99.999%
Ideal
Amdahl's Law

Number of Processors

Speed Up

PP 50%
PP 75%
PP 80%
PP 90%
PP 99.9%
PP 99.99%
PP 99.999%
Ideal
Why is Amdahl’s law a fallacy?

• It really isn’t!
• Most think in terms of number of lines of code to measure the percent parallel and serial.
  ➢ The wrong metric!
• The reality is the time spend executing the parallel and serial pieces of code.
  ➢ How does a good parallel or scalable code perform?
    ▪ 50% to 75% Efficiency on 512 to 1024 nodes.
Design for Scalability

• Over-design
  ➢ Don’t meet minimal requirements
  ➢ Think about the next generation computer
    ▪ Faster processor same memory
    ▪ More memory
  ➢ Deal with obvious trade-offs.

• Backward Compatibility
  ➢ Make sure the over-design doesn’t kill you on current machines.
    ▪ I have to have 10Gbytes memory
    ▪ I need a teraflop processor.
  ➢ Test cases and functionality on workstations
Latency Hiding

- A function of both the memory hierarchy and communication sub-system.

- **Memory**
  - Hardware or software mechanisms to execute data movement and instructions to avoid stalled processors or minimize cache misses.

- **Communication**
  - Hardware or software mechanisms to overlap communication with execution.
    - Another level of concurrency
Amdahl’s Rule of Thumb

• Sometimes called his Other Law.
• Processing speed, memory and communications should be balanced.
  ➢ 1 MIPS, 1MB memory, 1 Mb/S I/O

• Evolved into
  ➢ 1 Mflop (sustained), 1MB memory, 1 Mb/s I/O

• PetaFLOPS Project
  ➢ Memory = (Speed)$^{\frac{3}{4}}$
## Granularity/Latency Balance

<table>
<thead>
<tr>
<th>Granularity</th>
<th>Processor Speed (Mflops)</th>
<th>Latency (μs)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fine 100 Flop</td>
<td>4</td>
<td>25</td>
</tr>
<tr>
<td></td>
<td>20</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td>100</td>
<td>1</td>
</tr>
<tr>
<td>Medium 1000 Flop</td>
<td>10</td>
<td>100</td>
</tr>
<tr>
<td></td>
<td>50</td>
<td>20</td>
</tr>
<tr>
<td></td>
<td>250</td>
<td>4</td>
</tr>
<tr>
<td>Coarse 10000+ Flop</td>
<td>10</td>
<td>1000</td>
</tr>
<tr>
<td></td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td></td>
<td>1000</td>
<td>10</td>
</tr>
</tbody>
</table>
## Work Load/Bandwidth Balance

<table>
<thead>
<tr>
<th>Comm. To Compute Ratio $\alpha$</th>
<th>Processor Speed (Mflops)</th>
<th>$r_{\infty}$ (MB/s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.01 B/flop</td>
<td>4</td>
<td>0.04</td>
</tr>
<tr>
<td></td>
<td>20</td>
<td>0.2</td>
</tr>
<tr>
<td></td>
<td>100</td>
<td>1</td>
</tr>
<tr>
<td>0.1 B/flop</td>
<td>10</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>50</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td>250</td>
<td>25</td>
</tr>
<tr>
<td>1 B/flop</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td></td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td></td>
<td>1000</td>
<td>1000</td>
</tr>
</tbody>
</table>
## What we see in Industry

<table>
<thead>
<tr>
<th>System</th>
<th>Latency (µs)</th>
<th>( r_\infty ) Bandwidth (MB/s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Clusters</td>
<td>200 - 30</td>
<td>80 - 800</td>
</tr>
<tr>
<td>IBM SP</td>
<td>80 - 10</td>
<td>200 – 700</td>
</tr>
<tr>
<td>CRI T3E</td>
<td>13 - 3</td>
<td>200 - 340</td>
</tr>
<tr>
<td>SMP</td>
<td>2.0 – 0.05</td>
<td>300 – 1300</td>
</tr>
<tr>
<td>Wanted</td>
<td>0</td>
<td>( \infty )</td>
</tr>
</tbody>
</table>

Why is Parallel Programming Hard?

- **More complex than sequential programming**
  - Is sequential plus explicit concurrency
- **More than one basic programming model**
  - Concurrency hidden; the compiler does not handle sequential programs
  - Usually explicit in a parallel programming model
- **Software tools are less mature or non-existent.**
- **Languages are adapted/modified from sequential programming models.**
  - FORTRAN and C are the primary languages in both paradigms.
- **Lack of practical experience.**
  - Need to train the current/next generation programmers
What is a programming model?

- There are two basic kinds of programming models.
  - Abstracted
    - What the programmer “sees” or uses when developing a parallel program
    - Usually the developer’s breakdown of algorithms determines the most simple one to use.
  - Native
    - The lowest-level but accurate view for a specific parallel computing platform.
    - Other programming models can be mapped to or implemented on top of the native model.
    - Remember running SIMD on a MIMD box?
How can progress be made?

- There is roughly 15-20 years of practical parallel programming experience
  - Application guru’s
  - Computer Science guru’s
  - A few teams of both!!

- Native models from the vendor offerings are converging
  - Single or multiple address space models

- Abstract Models fall into four categories
  - Implicit (Compilers, Analysis Environments) [Scotty]
  - Data Parallel (SIMD)
  - Shared Variable (SIMD/MIMD)
  - Message passing (MIMD)

- Most applications enforce SPMD on top of these.
Programming Parallel Computers

• Essentially 4 ways to attack the issue
  1. Extend an existing compiler to translate sequential programs into parallel programs
  2. Extend an existing language with new operations to express parallelism
  3. Add a new parallel language layer on top of an existing sequential language.
  4. Define a new language and compiler system.
Shared Memory Programming

• The most simple parallel mechanism available.
  ▶ All nodes/processes see all of the memory 😊
  ▶ Many algorithms can be easily implemented.
  ▶ Either a process or thread based mechanism.
  ▶ There are three main methods in use today
    ▪ System specific shared memory mechanisms
      ♦ Cray, Sun, SGI, Fujitsu, NEC, Hitachi
    ▪ Pthreads (POSIX) thus “portable”
      ♦ Runs on most systems
      ♦ C only
    ▪ OpenMP
      ♦ New standard designed for SMPs.
      ♦ Works with C/C++ and FORTRAN
Thinking in Parallel.

- **It Hurts!! [No pain No gain 😊]**
  - There is a fine line between:
    - parallel-ization and
    - paralyze-ization

- **In sequential computing**
  - All of the right hand side things need to be computed in order!
    - Can be streamlined via a pipeline.
  - All the left hand assignments are made as results are computed.

- **Now picture this happening on multiple nodes.**
  - All operations of each process/thread are independent.
  - Memory may not be independent.
Thinking in Parallel [2]

• Vectorization or sequential optimization is targeted at the inner most loop!!
  ➢ Fine grained work (loop level work)

• Parallelization targets the outer most loop or groups of functions.
  ➢ Medium or course grained work
  ➢ Load Balance is an issue now.
Shared Memory Parallelization

- Loosely called TASKING
  - `-h taskn` compiler flag

- Autotasking
  - Compiler driven tasking
  - Works best on programs in which most of the work is in nested loops that do not contain function calls.

- Microtasking
  - Assisted by programmer

- Macrotasking
  - Driven by the programmer at the function or subroutine level
  - Compilers do not “parallelize” the functions but the calling mechanism is in a parallel region.
Micro/Macro tasking

• This requires that you understand the requirements for tasking and perform your own analysis.

• To direct tasking manually, you must
  ➢ identify the regions of your program that can run in parallel
  ➢ insert directives to specify these regions to the compiler.

• Those portions of applications that do significant amounts of work and are inherently parallel are candidates for tasking.

• You must determine
  ➢ where to insert the directives,
  ➢ the context of variables, and
  ➢ check whether the results are correct.
Task Initialization

• A function that potentially executes tasking code requires some initialization.
  ➢ Sets up the arguments for each slave function.
  ➢ These arguments are usually the variables that the slave task shares with the master task and with other slave tasks.
  ➢ The execution cost for this code is usually small.
  ➢ For functions or programs that contain a large number of distinct parallel regions, however, the cost may become significant.
How does a Shared Memory program work?

• **Starts as a single thread/process**
• **At the point of parallel work asks the system for resources.**
• **May get all or none of the requested resources.**
• **Subtle reconnection mechanism.**
• **Move on to the next program segment**
  ➢ Sequential or parallel.
Architectural Comparisons

• **Number of nodes**
  - **MPP**
    - 256 to 3000
  - **SMP**
    - 32 to 1024
  - **Clusters**
    - 10 to 1024
  - **Distributed Systems**
    - 100 to 1024
Architectural Comparisons

• Node Complexity or Granularity
  ➢ MPP
    ▪ Fine, medium, coarse
  ➢ SMP
    ▪ Fine or medium
  ➢ Clusters
    ▪ Medium or coarse
  ➢ Distributed Systems
    ▪ Medium or coarse
Architectural Comparisons

• Internode Communication Method
  ➢ MPP
    ▪ Message Passing or Distributed Shared Memory
  ➢ SMP
    ▪ Shared Memory
  ➢ Clusters
    ▪ Message Passing or software based Distributed Shared Memory
  ➢ Distributed Systems
    ▪ Shared files (database)
    ▪ RPC, Doors, DCOM, CORBA, etc.
    ▪ Message passing
Architectural Comparisons

• Job Scheduling
  ➢ MPP
    ▪ Single global Queue with job classifications
      ♦ Small, medium, large
      ♦ Use all of the system
  ➢ SMP
    ▪ Single Queue with varying job sizes
  ➢ Clusters
    ▪ Single or Multiple Queues
    ▪ Coordinated utilization
  ➢ Distributed Systems
    ▪ Independent multiple queues
    ▪ Federated Queues now a hot research topic.
Architectural Comparisons

• Single System Image (SSI)
  ➢ MPP
    ▪ Some are (T3E) some are not (The rest)
    ▪ Bproc systems give you the look and feel.
  ➢ SMP
    ▪ By definition yes.
  ➢ Clusters
    ▪ Desired?
    ▪ Used as a single resource but no real SSI (in name only)
  ➢ Distributed Systems
    ▪ By definition no.
Architectural Comparisons

• Node OS: type and number
  ➢ MPP
    ▪ Microkernel with Distributed controlling functions
    ▪ Multiple OS
    ▪ Pseudo-monolithic (bproc)
  ➢ SMP
    ▪ One monolithic OS
  ➢ Clusters
    ▪ Multiple homogenous OS instances
  ➢ Distributed Systems
    ▪ Multiple heterogeneous OS instances
Architectural Comparisons

- **Address Space**
  - **MPP**
    - Multiple (IBM SP)
    - Single (Convex, SUN, SGI)
    - Distributed Shared Memory (T3E)
  - **SMP**
    - Single
  - **Clusters**
    - Multiple – homogenous
    - Multiple - heterogeneous
  - **Distributed Systems**
    - Multiple - heterogeneous
Architectural Comparisons

• **Internode Security**
  - **MPP**
    - User Space on the system only
  - **SMP**
    - User Space on the system only
  - **Clusters**
    - User Space on a systems with gateway firewall
    - Required otherwise
  - **Distributed Systems**
    - Required, Public Key authorization?
Architectural Comparisons

• **Network Protocol**
  - **MPP**
    - Non-standard, proprietary
  - **SMP**
    - Non-standard, proprietary
  - **Clusters**
    - Non-standard, proprietary or
    - Standard (TCP/IP)
  - **Distributed Systems**
    - Standard (TCP/IP)
Architectural Comparisons

• **System Availability**
  - **MPP**
    - Low to Medium
    - End of life cycle, very high
  - **SMP**
    - High but the one processor gotcha.
  - **Clusters**
    - Highly Available (rarely fault-tolerant).
  - **Distributed Systems**
    - Medium
    - High if redundant resources (Throw $$$ from the train).
Architectural Comparisons

• **Performance Metrics**
  - **MPP**
    - Throughput (total amount of work accomplished)
    - Turnaround time (how fast does MY job run?)
  - **SMP**
    - Turnaround time
  - **Clusters**
    - Throughput and Turnaround time
  - **Distributed Systems**
    - Response Time but really Throughput of transactions