Introduction To Parallel Computing

Mohamed Iskandarani and Ashwanth Srinivasan

November 12, 2008
Outline

Overview

Concepts

Parallel Memory Architecture

Parallel Programming Paradigms
  - Shared memory paradigm
  - Message passing paradigm
  - Data parallel paradigm

Parallelization Strategies
What is Parallel Computing

- Harnessing multiple computer resources to solve a computational problem
  - single computer with multiple processors
  - a set of networked computers
  - networked multi-processors

- Computational problem
  - Can be broken into independent tasks and/or data
  - Can execute multiple instructions
  - Can be solved faster with multiple CPUs

- Examples
  - Geophysical fluid dynamics
    - ocean/atmosphere weather, climate
  - Optimization problems
  - Statigraphy
  - Genomics
  - Graphics
Why Use Parallel Computing

1. Overcome limits to serial computing
   1.1 Limits to increase transistor density
   1.2 Limits to data transmission speed
   1.3 Prohibitive cost of supercomputer (niche market)

2. Commodity (cheap) components to achieve high performance

3. Faster turn-around time

4. Solve larger problems
Serial Von Neumann Architecture

- Memory stores program instructions and data
- CPU fetches instructions/data from memory
- CPU executes instructions sequentially
- results are written back to memory
Flynn’s classification

Classify Parallel Computer Along Data and Instruction axes

<table>
<thead>
<tr>
<th>Data Stream</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>SISD</td>
<td>SIMD</td>
</tr>
<tr>
<td>Single Instruction Single Data</td>
<td>Single Instruction Multiple Data</td>
</tr>
<tr>
<td>MISD</td>
<td>MIMD</td>
</tr>
<tr>
<td>Multiple Instruction Single Data</td>
<td>Multiple Instruction Multiple Data</td>
</tr>
</tbody>
</table>
Single Instruction Single Data

- A serial (non-parallel) computer
- CPU acts on single instruction stream per cycle
- Only one-data item is being used at input each cycle
- Deterministic execution path
- Example: most single CPU laptops/workstations
- Example:

<table>
<thead>
<tr>
<th>load A</th>
<th>Load B</th>
<th>C=A+B</th>
<th>Store C</th>
<th>A=2*B</th>
<th>Store A</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
  +--------+--------+-------+--------+-------+--------+
  |        |        |       |        |       |        |
  |        |        |       |        |       |        |
  | time   |        |       |        |       |        |
Single Instruction Multiple Data (SIMD)

- A type of parallel computer
- **Single Instruction**: All processors execute the same instruction at any clock cycle
- **Multiple Data**: Each processor unit acts on different data elements
- Typically high speed and high-bandwidth internal network
- A large number of small capacity instruction units
- Synchronous and deterministic execution
- Best suited for problems with high regularity, e.g. image processing, graphics
- Examples:
  - Vector processors: Cray C90, NEC SX2, IBM9000
  - Processor arrays: Connection Machine CM-2, Maspar MP-1
## Single Instruction Multiple Data (SIMD)

<table>
<thead>
<tr>
<th>P1</th>
<th>P2</th>
<th>P3</th>
</tr>
</thead>
<tbody>
<tr>
<td>load A(1)</td>
<td>load A(2)</td>
<td>load A(3)</td>
</tr>
<tr>
<td>Load B(1)</td>
<td>Load B(2)</td>
<td>Load B(3)</td>
</tr>
<tr>
<td>C(1) = A(1) + B(1)</td>
<td>C(2) = A(2) + B(2)</td>
<td>C(3) = A(3) + B(3)</td>
</tr>
<tr>
<td>Store C(1)</td>
<td>Store C(2)</td>
<td>Store C(3)</td>
</tr>
<tr>
<td>A(1) = 2 * B(1)</td>
<td>A(2) = 2 * B(2)</td>
<td>A(3) = 2 * B(3)</td>
</tr>
<tr>
<td>Store A(1)</td>
<td>Store A(2)</td>
<td>Store A(3)</td>
</tr>
</tbody>
</table>
Multiple Instruction Single Data: MISD

- Uncommon type of parallel computers
Multiple Instruction Multiple Data: MIMD

- Most common type of parallel computers
- **Multiple Instruction**: Each processor maybe executing a different instruction stream
- **Multiple Data**: Each processor is working on different data stream.
- Execution could be synchronous or asynchronous
- Execution not necessarily deterministic
- Example: most current supercomputers, clusters, IBM blue-gene
## Multiple Instruction Multiple Data (MIMD)

<table>
<thead>
<tr>
<th>P1</th>
<th>P2</th>
<th>P3</th>
</tr>
</thead>
<tbody>
<tr>
<td>load $A(1)$</td>
<td>$x = y \times z$</td>
<td>$C = A + B$</td>
</tr>
<tr>
<td>Load $B(1)$</td>
<td>sum = sum + x</td>
<td>$D = \max(C, B)$</td>
</tr>
<tr>
<td>$C(1) = A(1) + B(1)$</td>
<td>if (sum &gt; 0.0)</td>
<td>$D = \text{myfunc}(B)$</td>
</tr>
<tr>
<td>Store $C(1)$</td>
<td>call subC(2)</td>
<td>$D = D \times D$</td>
</tr>
</tbody>
</table>
Shared Memory Processors

- All processors access all memory as global address space
- Processors operate independently but share memory resources
Shared Memory Processors
General characteristics

- Advantages
  - Global address space simplified programming
  - Allow incremental parallelization
  - Data sharing between CPUs fast and uniform
- Disadvantages
  - Lack of memory scalability between memory and CPU
  - Increasing CPUs increase memory traffic geometrically on shared memory-CPU paths.
  - Programmers responsible for synchronization of memory accesses
  - Soaring expense of internal network.
Shared Memory Processors Categories

- Uniform memory access (UMA)
  - Also called Symmetric Multi-Processors (SMP)
  - identical processors
  - equal access times to memory from any Pn
  - Cache Coherent: One processor’s update of shared memory is known to all processors. Done at hardware level.

- Non-Uniform memory access (NUMA)
  - Made by physically linking multiple SMPs
  - One SMP can access the memory of another directly.
  - Not all processors have equal access time
  - Memory access within SMP is fast
  - Memory access across network is slow
  - Extra work to maintain Cache-Coherency (CC-NUMA)
Distributed Memory

- Each processor has its own private memory
- No global address space
- Network access to communicate between processors

- Data sharing achieved via message passing
- Programmer responsible for messages and task synchronization
- Network fabric flexible (Ethernet, myrinet)
Distributed Memory

- **Advantages**
  - Memory size scales with CPUs
  - Fast local memory access with no network interference.
  - Cost effective (commodity components)

- **Disadvantages**
  - Programmer responsible for communication details
  - Difficult to map existing data structure, based on global memory, to this memory organization.
  - Non-uniform memory access time. Dependence on network latency, bandwidth, and congestion.
  - All or nothing parallelization.
Hybrid Distributed-Shared Memory

- Most common type of current parallel computers
- Shared memory component is a CC-UMA SMP
- Local global address space within each SMP
Parallel Programming Paradigms

- Several programming paradigms are common
  - Shared Memory (OpenMP, threads)
  - Message Passing
  - Hybrid
  - Data parallel (HPF)
- Programming paradigm abstracts hardware and memory architecture
- Paradigms are NOT specific to a particular type of machine
- Any of these models can (in principle) be implemented on any underlying hardware.
- Shared memory model on distributed hardware: Kendal Square Research
- SGI origin is a shared memory machine which supported effectively message passing.
- Performance depends on choice of programming model, and knowing details of data traffic.
Shared Memory Model

- Parallel tasks share a common global address space
- Read and write can occur asynchronously.
- Locks and semaphores to control shared data access
  - avoid reading stale data from shared memory.
  - avoid multiple CPUs writing to the same shared memory address.
- Compiler translates variables into memory addresses which are global
- User specifies private and shared variables
- Incremental parallelization possible
Threads

- Commonly associated with shared memory machines
- A single process can have multiple execution paths
- Threads communicate via global address space
Threads

program prog  ! main program holds resources
  call serial
  ! Task Parallel section
  call sub1       ! independent task 1
  call sub2       ! independent task 2
  call sub3       ! independent task 3
  call sub4       ! independent task 4
  ! Synchronize here
  do i = 0,n+1    ! Data Parallel section
    A(i) = func(x(i))
  enddo          ! Don’t fuse loops to
  do i = 1,n     ! maintain data independence
    G(i) = (A(i+1)-A(i-1))/(2.0*dx)
  enddo
  call moreserial
end program prog
Threads

- OS loads `prog` which acquires resources to run.
- After some serial work, a number of threads are created.
- All threads share the resources of `prog`.
- Each thread has local data and can access global data.
- Task parallelism: each data calls a separate procedure.
- Synchronize before do-loop starts.
- Threads communicate via global variables.
- Threads can come and go but `prog` remains.
Threads Implementations

- Posix threads
  - library based and require parallel coding
  - adheres to IEEE POSIX standard.
  - provided by most vendors in addition to their proprietary thread implementation.
  - requires considerable attention to detail

- OpenMP
  - Based on compiler directives
  - Allows incremental parallelization
  - Portable and available on numerous platforms
  - Available in C/C++/Fortran implementations
  - Easiest to use
  - Performance requires attention to shared data layout
Message Passing

- Each task uses its private memory
- Multiple tasks may reside on one machine
- Tasks communicate by sending and receiving messages
- Data traffic requires cooperation
each send must have a corresponding receive
Message Passing Implementation

- Programmer is responsible for parallelization
- Parallelization follows data decomposition paradigm
- Programmer calls a communication library to send/receive messages
- **Message Passing Interface (MPI)** defacto standard since 1994
- Portable MPI available (MPICH)
- Use Vendor provided MPI library when possible (same API)
- Shared memory version of MPI communication available (SGI-Origin)
Data Parallel Paradigm

- Parallel operations on data sets (mostly arrays)
- Each task works on portion of data set
- on SMP: data accessed through global addresses
- on Distributed Memory: messages divy up data to tasks
- Effected through library calls or compiler directive
- High Performance Fortran (HPF)
  - Extension to F90
  - Support parallel construct
    `forall, where`
  - Assertions to improve code optimization
  - HPF Compiler hide task communication details
Data Parallel Paradigm

Array \( p(3N) \)
Array \( r(3N) \)

<table>
<thead>
<tr>
<th>Task</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>Task 1</td>
<td>( \text{do } i = 1, N ) ( p(i) = p(i) + r(i) ) ( \text{enddo} )</td>
</tr>
<tr>
<td>Task 2</td>
<td>( \text{do } i = N+1, 2N ) ( p(i) = p(i) + r(i) ) ( \text{enddo} )</td>
</tr>
<tr>
<td>Task 3</td>
<td>( \text{do } i = 2N+1, 3N ) ( p(i) = p(i) + r(i) ) ( \text{enddo} )</td>
</tr>
</tbody>
</table>

\( p = p + r \)
Other programming paradigms

- Hybrid of shared/distributed memory
  - OpenMP within a node
  - MPI across nodes

- Single Program Multiple Data
  - All tasks execute same program
  - A task may execute different set of instructions
  - Tasks use different data

- Multiple Program Multiple Data
  - Different programs are executing simultaneously
  - A parallel Ocean Model
  - A parallel Atmospheric Model
  - Coupling at Air-Sea interface
How to parallelize

- **Automatic (Compiler parallelization)**
  - Easy by using compiler flags
  - Compiler distributes data to processors
  - Limited scalability
  - Clean code to allow compiler analysis
  - May slow down code

- **Manual (Compiler parallelization)**
  - Must understand model and memory architecture
  - Explicit data decomposition
  - Can be done with compiler directives
  - Time consuming for distributed memory

- Ultimately depends on problem and time available
Problem examples

- Embarrassingly parallel problem
  Calculate potential energy of several thousand molecular configurations. When done find minimum.

- Non-parallelizable problem
  Fibonacci series (1,1,2,3,5,8,...)

\[ F(k + 2) = F(k + 1) + F(k) \]

Data dependency prevents parallelization