Overview

- Basic concepts
- Programming models
- Multiprogramming
- Shared address space model
  - UMA versus NUMA
  - Distributed shared memory
  - Task parallel
  - Data parallel, vector and SIMD
- Message passing model
- Hybrid systems
- BSP model
Parallel Programming: Basic Concepts

- Control
  - How is parallelism created
  - What orderings exist between operations
  - How do different threads of control synchronize

- Naming
  - What data is private or shared
  - How is logically shared data accessed or communicated

- Operations
  - What are the basic operations
  - Which operations are considered atomic

- Cost
  - How do we account for the cost of each of the above
Programming Models

- *Programming model* = conceptualization of the machine that a programmer uses for developing applications

- *Multiprogramming*
  - Independence tasks, no communication or synchronization at program level

- *Shared address space (shared memory) programming*
  - Tasks operate and communicate via shared data, like bulletin boards

- *Message passing* programming
  - Explicit point-to-point communication, like phone calls (connection oriented) or email (connectionless, mailbox posts)
Task versus Data Parallel

- Task parallel (maps to high-level MIMD machine model)
  - Task differentiation, like restaurant cook, waiter, and receptionist
  - Communication via shared address space or message passing
  - Synchronization is explicit (locks)
  - Underscores operations on private data, explicit constructs for communication of shared data and synchronization

- Data parallel (maps to high-level SIMD machine model)
  - Global actions on data by tasks that execute the same code
  - Communication via shared address space or logically shared address space with underlying message passing
  - Synchronization is implicit (lock-step execution or barriers)
  - Underscores operations on shared data, private data must be defined explicitly or is simply mapped onto shared data space
Example: \[ A = \sum_{i=1}^{N} f(a_i) \]

- Parallel decomposition
  - Assign \( N/P \) elements to each processor
  - Each processor computes the partial sum
  - One processor collects the partial sums

- Classes of data
  - Logically shared: array \( a \), global sum \( A \)
  - Logically private: the function evaluations
  - Either logically shared or private: partial sums \( A_j \)
Programming Model 1

- *Shared address space (shared memory)* programming

- Task parallel
  - Program is a collection of threads of control

- Collectively operate on a set of *shared data* items
  - Global static variables, Fortran common blocks, shared heap

- Each thread has *private variables*
  - Thread state data, local variables on the runtime stack

- Threads coordinate explicitly by synchronization operations on shared variables, which involves
  - Thread creation and join
  - Reading and writing flags
  - Locks and semaphores

- Similar to concurrent programming on uniprocessor
Programming Model 1

- **Uniform memory access (UMA) shared memory machine**
  - Each processor has uniform access to memory
  - Symmetric multiprocessors (SMP)

- No local/private memory, private variables are put in shared memory

- Cache makes access to private variables seem “local”

![Programming model](image)

![Machine model](image)
Programming Model 1

- Nonuniform memory access (NUMA) shared memory machine
  - Memory access time depends on location of data relative to processor
  - Local access is faster
- No local/private memory, private variables are put in shared memory
Programming Model 1

- Distributed shared memory machine (DSM)
- Logically shared address space
  - Remote memory access is more expensive (NUMA)
  - Remote memory access requires communication, automatic either done in hardware or via software layer
Programming Model 1

Thread 1

shared A
shared A[1..2]
private i

A[1] := 0
for i = 1..N/2

Thread 2

shared A
shared A[1..2]
private i

for i = N/2+1..N

What could go wrong?
Programming Model 1

Thread 1

\[ \vdots \]

Thread 2

\[ \vdots \]

Thread 2 has not completed in time
Programming Model 1

Thread 1

shared A
shared A[1..2]
private i

A := 0
A[1] := 0
for i = 1..N/2
A := A + A[1]

Thread 2

shared A
shared A[1..2]
private i

A := 0
for i = N/2+1..N

What could go wrong?
### Programming Model 1

#### Thread 1

\[
\begin{align*}
& \vdots \\
A & := A + A[1]
\end{align*}
\]

#### Thread 2

\[
\begin{align*}
& \vdots \\
\end{align*}
\]

**Race condition**

\[
\begin{align*}
\text{reg1} & = A \\
\text{reg2} & = A[1] \\
\text{reg1} & = \text{reg1} + \text{reg2} \\
A & = \text{reg1}
\end{align*}
\]

\[
\begin{align*}
\text{reg1} & = A \\
\text{reg2} & = A[2] \\
\text{reg1} & = \text{reg1} + \text{reg2} \\
A & = \text{reg1}
\end{align*}
\]

Programming Model 1

Thread 1

shared A
shared A[1..2]
private i

A[1] := 0
for i = 1..N/2
atomic A := A + A[1]

Thread 2

shared A
shared A[1..2]
private i

for i = N/2+1..N

Solution with atomic operations to prevent race condition
Programming Model 1

Thread 1

shared A
shared A[1..2]
private i

A[1] := 0
for i = 1..N/2

lock
A := A + A[1]
unlock

Thread 2

shared A
shared A[1..2]
private i

for i = N/2+1..N

lock
unlock

Solution with locks to ensure mutual exclusion
Programmings Model 1

\[
A_j = \sum_{i=(j-1)k+1}^{jk} f(a_i)
\]
\[
A = \sum_{i=1}^{P} A_i
\]

Thread 1

shared A
private Aj
private i

Aj := 0
for i = 1..N/2
    Aj := Aj+f(a[i])
lock
A := A + Aj
unlock
barrier
... := A

Thread 2

shared A
private Aj
private i

Aj := 0
for i = N/2+1..N
    Aj := Aj+f(a[i])
lock
A := A + Aj
unlock
barrier \text{ } \text{ } \text{ } All procs synchronize
... := A

With private A_j and barrier synchronization
Programming Model 2

- **Shared address space** (shared memory) programming
- **Data parallel programming**
  - Single thread of control consisting of parallel operations
  - Parallel operations are applied to (a specific segment of) a data structure, such as an array
- Communication is implicit
- Synchronization is implicit

```
shared array a, x
shared A
a := array of input data
x := f(a)
A := sum(x)
```
Programming Model 2

- E.g. data parallel programming with a vector machine
- One instruction executes across multiple data elements, typically in a pipelined fashion

shared array \( a, x \)
shared \( A \)
\( a := \text{array of input data} \)
\( x := f(a) \)
\( A := \text{sum}(x) \)
Programming Model 2

- Data parallel programming with a SIMD machine
- Large number of (relatively) simple processors
  - Like multimedia extensions (MMX/SSE/AltiVec) on uniprocessors, but with scalable processor grids
- A control processor issues instructions to simple processors
  - Each processor executes the same instruction (in lock-step)
  - Processors are selectively turned off for control flow in program

```fortran
REAL, DIMENSION(6) :: a,b
... WHERE b /= 0.0
  a = a/b
ENDWHERE
```

*Fortran 90 / HPF (High-Performance Fortran)*

*Lock-step execution by an array of processors with some processors temporarily turned off*
Programming Model 3

- Message passing programming

- Program is a set of *named* processes
  - Process has thread of control and local memory with local address space

- Processes communicate via explicit data transfers
  - Messages between source and destination, where source and destination are named processors P0…Pn (or compute nodes)
  - Logically shared data is explicitly partitioned over local memories
  - Communication with send/recv via standard message passing libraries, such as MPI and PVM
Programming Model 3

- Message passing programming
- Each node has a network interface
  - Communication and synchronization via network
  - Message latency and bandwidth is dependent on network topology and routing algorithms

*Programming model*  
*Machine model*
Programming Model 3

- Message passing programming
- Each node has a network interface
  - Communication and synchronization via network
  - Message latency and bandwidth is dependent on network topology and routing algorithms
Programming Model 3

- Message passing programming
- Each node has a network interface
  - Communication and synchronization via network
  - Message latency and bandwidth is dependent on network topology and routing algorithms
Programming Model 3

- Message passing programming
- On shared memory machine
  - Communication and synchronization via shared memory
  - Message passing library copies data (messages) in memory, less efficient (MPI call overhead) but portable

Programming model

Machine model

Message passing on a shared memory machine

Copy data
Programming Model 3

Solution with message passing, where global $a[1..N]$ is distributed such that each processor has a local array $a_1[1..N/2]$.
Programming Model 3

Processor 1

\[ A1 := 0 \]
\[ \text{for } i = 1..N/2 \]
\[ A1 := A1 + f(al[i]) \]
send A1 to P2
receive A2 from P2
A := A1 + A2

Processor 2

\[ A2 := 0 \]
\[ \text{for } i = 1..N/2 \]
\[ A2 := A2 + f(al[i]) \]
send A2 to P1
receive A1 from P1
A := A1 + A2

Alternative solution with message passing, where global \( a[1..N] \) is distributed such that each processor has a local array \( al[1..N/2] \)

What could go wrong?
Programming Model 3

Processor 1

\[
A_1 := 0 \\
\text{for } i = 1..N/2 \\
\quad A_1 := A_1 + f(a_1[i]) \\
\quad \text{send } A_1 \text{ to } P_2 \\
\quad \text{receive } A_2 \text{ from } P_2 \\
A := A_1 + A_2
\]

Processor 2

\[
A_2 := 0 \\
\text{for } i = 1..N/2 \\
\quad A_2 := A_2 + f(a_1[i]) \\
\quad \text{send } A_2 \text{ to } P_1 \\
\quad \text{receive } A_1 \text{ from } P_1 \\
A := A_1 + A_2
\]

Deadlock with synchronous blocking send operations: both processors wait for data to be send to a receiver that is not ready to accept the message

Blocking and non-blocking versions of send/recv operations are available in message passing libraries: compare connection-oriented with rendezvous (telephone) to connectionless (mailbox)
Programming Model 4

- Hybrid systems: clusters of SMPs
- Shared memory within SMP, message passing outside
- Programming model with three choices:
  - Treat as “flat” system: always use message passing, even within an SMP
    - Advantage: ease of programming and portability
    - Disadvantage: ignores SMP memory hierarchy and advantage of UMA shared address space
  - Program in two layers: shared memory programming and message passing
    - Advantage: better performance (use UMA/NUMA intelligently)
    - Disadvantage: harder (and ugly!) to program
  - Program in three layers: SIMD (e.g. SSE instructions) per core, shared memory programming between cores on an SMP node, and message passing between nodes
Programming Model 4

shared \( a[1..N/\text{numnodes}] \)
private \( n = N/\text{numnodes}/\text{numprocs} \)
private \( x[1..n] \)
private \( lo = (\text{pid}-1)\times n \)
private \( hi = lo+n \)
\( x[1..n] = f(a[lo..hi]) \)
\( A[\text{pid}] := \text{sum}(x[1..n]) \)
send \( A[\text{pid}] \) to node 1

\[\begin{align*}
A &:= 0 \\
\text{if node=} 1 \text{ and pid=} 1 \\
\text{for } j = 1..\text{numnodes} \\
&\quad \text{for } i = 1..\text{numprocs} \\
&\quad \quad \text{receive } A_j \text{ from node}(j) \\
&\quad \quad A := A + A_j \\
\text{Extra code for node 1 proc 1}
\end{align*}\]
Programming Model 5

- **Bulk synchronous processing** (BSP)
- A BSP **superstep** consists of three **phases**
  1. Compute phase: processes operate on local data (also read access to shared memory on SMP)
  2. Communication phase: all processes cooperate in exchange of data or reduction of global data
  3. Barrier synchronization

- A parallel program is composed of supersteps
  - Ensures that computation and communication phases are completed before the next superstep

- Simplicity of data parallel programming, without the restrictions
The cost of a BSP superstep \( s \) is composed of three parts:
- \( w_s \) local computation cost of \( s \)
- \( h_s \) is the number of messages send in superstep \( s \)
- \( l \) is the barrier cost

The total cost of a program with \( S \) supersteps is

\[
W + Hg + Sl = \sum_{s=1}^{S} w_s + g \sum_{s=1}^{S} h_s + Sl
\]

where \( g \) is the communication cost such that it takes \( gh \) time to send \( h \) messages
Summary

- Goal is to distinguish the programming model from underlying hardware
- Message passing, data parallel, BSP
  - Objective is portable correct code
- Hybrid
  - Tuning for the architecture
  - Objective is portable fast code
  - Algorithm design challenge (less uniformity)
  - Implementation challenge at all levels (fine to coarse grain)
    - Blocking at loop and data level (compiler and programmer)
    - SIMD vectorization at loop level (compiler and programmer)
    - Shared memory programming for each node (OpenMP)
    - Message passing between nodes (MPI)