Outline

• Overview
• Theoretical background
• Parallel computing systems
• Parallel programming models
• MPI/OpenMP examples
OVERVIEW
What is Parallel Computing?

- Parallel computing: use of multiple processors or computers working together on a common task.
  - Each processor works on its section of the problem
  - Processors can exchange information

![Diagram of Parallel Computing](grid_of_problem_to_be_solved)

- **CPU #1** works on this area of the problem
- **CPU #2** works on this area of the problem
- **CPU #3** works on this area of the problem
- **CPU #4** works on this area of the problem

---

**Grid of Problem to be solved**

**x**

**y**

---

**TACC**

**The University of Texas at Austin**

**Texas Advanced Computing Center**
Why Do Parallel Computing?

• Limits of single CPU computing
  – performance
  – available memory

• Parallel computing allows one to:
  – solve problems that don’t fit on a single CPU
  – solve problems that can’t be solved in a reasonable time

• We can solve...
  – larger problems
  – the same problem faster
  – more cases

• All computers are parallel these days, even your iphone 4S has two cores...
THEORETICAL BACKGROUND
Speedup & Parallel Efficiency

• Speedup:
  \[ S_p = \frac{T_s}{T_p} \]
  
  – \( p \) = # of processors
  – \( T_s \) = execution time of the sequential algorithm
  – \( T_p \) = execution time of the parallel algorithm with \( p \) processors
  – \( S_p = P \) (linear speedup: ideal)

• Parallel efficiency
  \[ E_p = \frac{S_p}{p} = \frac{T_s}{pT_p} \]
Limits of Parallel Computing

• Theoretical Upper Limits
  – Amdahl’s Law
  – Gustafson’s Law

• Practical Limits
  – Load balancing
  – Non-computational sections

• Other Considerations
  – time to re-write code
Amdahl’s Law

- All parallel programs contain:
  - parallel sections (we hope!)
  - serial sections (we despair!)
- Serial sections limit the parallel effectiveness
- Amdahl’s Law states this formally
  - Effect of multiple processors on speed up

\[ S_p = \frac{T_S}{T_P} \frac{P}{Pf_s + f_p} \]

where
- \( f_s \) = serial fraction of code
- \( f_p \) = parallel fraction of code
- \( P \) = number of processors

Example:
\[ f_s = 0.5, f_p = 0.5, P = 2 \]
\[ S_{p, max} = 1 / (0.5 + 0.25) = 1.333 \]
Amdahl’s Law (strong scaling)
Practical Limits: Amdahl’s Law vs. Reality

- In reality, the situation is even worse than predicted by Amdahl’s Law due to:
  - Load balancing (waiting)
  - Scheduling (shared processors or memory)
  - Cost of Communications
  - I/O
Gustafson’s Law

- Effect of multiple processors on run time of a problem with a fixed amount of parallel work per processor.

\[ S_p \cdot P \times (P - 1) \]

- \( \alpha \) is the fraction of non-parallelized code where the parallel work per processor is fixed (not the same as \( f_s \) from Amdahl’s law) and can vary with \( P \) and with problem size.
- \( P \) is the number of processors
Gustafson’s Law (weak scaling)
Comparison of Amdahl and Gustafson

Amdahl: fixed work

\[ f_p = 0.5 \]

\[
S = \frac{1}{f_s + f_p / N}
\]

\[
S_2 = \frac{1}{0.5 + 0.5 / 2} = 1.33
\]

\[
S_4 = \frac{1}{0.5 + 0.5 / 4} = 1.6
\]

Gustafson: fixed work per processor

\[ = 0.5 \]

\[
S_p \times (P - 1)
\]

\[
S_2 = 2 \times 0.5(2 - 1) = 1.5
\]

\[
S_4 = 4 + 0.5(4 - 1) = 2.5
\]
Scaling: Strong vs. Weak

- We want to know how quickly we can complete analysis on a particular data set by increasing the processor count
  - Amdahl’s Law
  - Known as “strong scaling”

- We want to know if we can analyze more data in approximately the same amount of time by increasing the processor count
  - Gustafson’s Law
  - Known as “weak scaling”
Some things to consider...

• An inefficient parallel algorithm vs. an efficient serial algorithm...
  – An inefficient yet parallel portion of code may increase the *overall* parallel efficiency of the code.
  – The actual performance obtained is not always (or often) intuitive. Try things and time them. Sometimes the results can be surprising.

• Readability/comprehensibility matter too.
PARALLEL SYSTEMS
“Old school” hardware classification

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

**SISD**  No parallelism in either instruction or data streams (mainframes)

**SIMD**  Exploit data parallelism (stream processors, GPUs)

**MISD**  Multiple instructions operating on the same data stream. Unusual, mostly for fault-tolerance purposes (space shuttle flight computer)

**MIMD**  Multiple instructions operating independently on multiple data streams (most modern general purpose computers, head nodes)

NOTE: GPU references frequently refer to SIMT, or single instruction multiple *thread*
Hardware in parallel computing

Memory access

• Shared memory
  – SGI Altix
  – IBM Power series nodes

• Distributed memory
  – Uniprocessor clusters

• Hybrid/Multi-processor clusters (Lonestar, Stampede)

• Flash based (e.g. Gordon)

Processor type

• Single core CPU
  – Intel Xeon (Prestonia, Wallatin)
  – AMD Opteron (Sledgehammer, Venus)
  – IBM POWER (3, 4)

• Multi-core CPU (since 2005)
  – Intel Xeon (Paxville, Woodcrest, Harpertown, Westmere, Sandy Bridge...)
  – AMD Opteron (Barcelona, Shanghai, Istanbul,...)
  – IBM POWER (5, 6...)
  – Fujitsu SPARC64 VIIIfx (8 cores)

• Accelerators
  – GPGPU
  – MIC
Shared and distributed memory

- All processors have access to a pool of shared memory
- Access times vary from CPU to CPU in NUMA systems
- Example: SGI Altix, IBM P5 nodes

- Memory is local to each processor
- Data exchange by message passing over a network
- Example: Clusters with single-socket blades
Hybrid systems

- A limited number, $N$, of processors have access to a common pool of shared memory

- To use more than $N$ processors requires data exchange over a network

- Example: Cluster with multi-socket blades
Multi-core systems

- Extension of hybrid model

- Communication details increasingly complex
  - Cache access
  - Main memory access
  - Quick Path (QPI) / Hyper Transport socket connections
  - Node to node connection via network
Accelerated (GPGPU and MIC) Systems

- Calculations made in both CPU and accelerator
- Provide abundance of low-cost flops
- Typically communicate over PCI-e bus
- Load balancing critical for performance
Accelerated (GPGPU and MIC) Systems

GPGPU (general purpose graphical processing unit)
- Derived from graphics hardware
- Requires a new programming model and specific libraries and compilers (CUDA, OpenCL)
- Newer GPUs support IEEE 754-2008 floating point standard
- Does not support flow control (handled by host thread)

MIC (Many Integrated Core)
- Derived from traditional CPU hardware
- Based on x86 instruction set
- Supports multiple programming models (OpenMP, MPI, OpenCL)
- Flow control can be handled on accelerator
PROGRAMMING MODELS
Types of parallelism

• Data Parallelism
  – Each processor performs the same task on different data (remember SIMD, MIMD)

• Task Parallelism
  – Each processor performs a different task on the same data (remember MISD, MIMD)

• Many applications incorporate both
Implementation: **Single Program Multiple Data**

- Dominant programming model for shared and distributed memory machines
- One source code is written
- Code can have conditional execution based on which processor is executing the copy
- All copies of code start simultaneously and communicate and synchronize with each other periodically
SPMD Model

program.c (source)

- program
  - process 0
    - processor 0
  - process 1
    - processor 1
  - process 2
    - processor 2
  - process 3
    - processor 3

Communication layer
Data Parallel Programming Example

- One code will run on 2 CPUs
- Program has array of data to be operated on by 2 CPUs so array is split into two parts.

```
program:
... if CPU=a then
    low_limit=1
    upper_limit=50
elseif CPU=b then
    low_limit=51
    upper_limit=100
end if
do I = low_limit, upper_limit
    work on A(I)
end do
... end program
```

```
program:
... low_limit=1
    upper_limit=50
do I = low_limit, upper_limit
    work on A(I)
end do
... end program
```

```
program:
... low_limit=51
    upper_limit=100
do I = low_limit, upper_limit
    work on A(I)
end do
... end program
```
Task Parallel Programming Example

- One code will run on 2 CPUs
- Program has 2 tasks (a and b) to be done by 2 CPUs

```
program.f:
  ...
  initialize
  ...
  if CPU=a then
    do task a
  elseif CPU=b then
    do task b
  end if
  ...
end program
```

```
CPU A
  program.f:
    ...
    initialize
    ...
    do task a
    ...
    end program
```

```
CPU B
  program.f:
    ...
    initialize
    ...
    do task b
    ...
    end program
```
Shared Memory Programming: pthreads

- Shared memory systems (SMPs, ccNUMAs) have a single address space
- Applications can be developed in which loop iterations (with no dependencies) are executed by different processors
- Threads are ‘lightweight processes’ (same PID)
- Allows ‘MIMD’ codes to execute in shared address space
Shared Memory Programming: OpenMP

• Built on top of pthreads
• shared memory codes are mostly data parallel, ‘SIMD’ kinds of codes
• OpenMP is a standard for shared memory programming (compiler directives)
• Vendors offer native compiler directives
Accessing Shared Variables

• If multiple processors want to write to a shared variable at the same time, there could be conflicts:
  – Process 1 and 2
  – read X
  – compute X+1
  – write X

• Programmer, language, and/or architecture must provide ways of resolving conflicts (mutexes and semaphores)
OpenMP Example #1: Parallel Loop

```c
$OMP PARALLEL DO
    do i=1,128
        b(i) = a(i) + c(i)
    end do
$OMP END PARALLEL DO
```

- The first directive specifies that the loop immediately following should be executed in parallel.
- The second directive specifies the end of the parallel section (optional).
- For codes that spend the majority of their time executing the content of simple loops, the PARALLEL DO directive can result in significant parallel performance.
OpenMP Example #2: Private Variables

```c
!$OMP PARALLEL DO SHARED(A,B,C,N) PRIVATE(I,TEMP)
do i=1,N
    TEMP = A(I)/B(I)
    C(I) = TEMP + SQRT(TEMP)
end do
!$OMP END PARALLEL DO
```

- In this loop, each processor needs its own private copy of the variable TEMP.

- If TEMP were shared, the result would be unpredictable since multiple processors would be writing to the same memory location.
Distributed Memory Programming: MPI

• Distributed memory systems have separate address spaces for each processor

• Local memory accessed faster than remote memory

• Data must be manually decomposed

• MPI is the de facto standard for distributed memory programming (library of subprogram calls)
Data Decomposition

• For distributed memory systems, the ‘whole’ grid is decomposed to the individual nodes
  – Each node works on its section of the problem
  – Nodes can exchange information
Typical Data Decomposition

- Example: integrate 2-D propagation problem:

Starting partial differential equation:

\[ \frac{\partial \Psi}{\partial t} = D \cdot \frac{\partial^2 \Psi}{\partial x^2} + B \cdot \frac{\partial^2 \Psi}{\partial y^2} \]

Finite Difference Approximation:

\[ \frac{f_{i,j}^{n+1} - f_{i,j}^n}{\Delta t} = D \cdot \frac{f_{i+1,j}^n - 2f_{i,j}^n + f_{i-1,j}^n}{\Delta x^2} + B \cdot \frac{f_{i,j+1}^n - 2f_{i,j}^n + f_{i,j-1}^n}{\Delta y^2} \]
MPI Example #1

- Every MPI program needs these:

```c
#include "mpi.h"
int main(int argc, char *argv[])
{
    int nPEs, iam;
    /* Initialize MPI */
    ierr = MPI_Init(&argc, &argv);
    /* How many total PEs are there */
    ierr = MPI_Comm_size(MPI_COMM_WORLD, &nPEs);
    /* What node am I (what is my rank?) */
    ierr = MPI_Comm_rank(MPI_COMM_WORLD, &iam);
    ...
    ierr = MPI_Finalize();
}
```
#include "mpi.h"

int main(int argc, char *argv[])
{
    int numprocs, myid;
    MPI_Init(&argc,&argv);
    MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
    MPI_Comm_rank(MPI_COMM_WORLD,&myid);
    /* print out my rank and this run's PE size */
    printf("Hello from %d of %d\n", myid, numprocs);
    MPI_Finalize();
}
MPI: Sends and Receives

• MPI programs must send and receive data between the processors (communication)

• The most basic calls in MPI (besides the three initialization and one finalization calls) are:
  – MPI_Send
  – MPI_Recv

• These calls are blocking: the source processor issuing the send/receive cannot move to the next statement until the target processor issues the matching receive/send.
Message Passing Communication

• Processes in message passing programs communicate by passing messages

• Basic message passing primitives: MPI_CHAR, MPI_SHORT, ...

• Send (parameters list)

• Receive (parameter list)

• Parameters depend on the library used

• Barriers
#include "mpi.h"

int main(int argc,char *argv[])
{
    int numprocs,myid,tag,source,destination,count,buffer;
    MPI_Status status;

    MPI_Init(&argc,&argv);
    MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
    MPI_Comm_rank(MPI_COMM_WORLD,&myid);
    tag=1234;
    source=0;
    destination=1;
    count=1;

    if(myid == source){
        buffer=5678;
        MPI_Send(&buffer,count,MPI_INT,destination,tag,MPI_COMM_WORLD);
        printf("processor %d sent %d\n",myid,buffer);
    }

    if(myid == destination){
        MPI_Recv(&buffer,count,MPI_INT,source,tag,MPI_COMM_WORLD,&status);
        printf("processor %d got %d\n",myid,buffer);
    }
    MPI_Finalize();
}
Final Thoughts

• These are exciting and turbulent times in HPC.
• Systems with multiple shared memory nodes and multiple cores per node are the norm.
• Accelerators are rapidly gaining acceptance.
• Going forward, the most practical programming paradigms to learn are:
  – Pure MPI
  – MPI plus multithreading (OpenMP or pthreads)
  – Accelerator models (MPI or multithreading for MIC, CUDA or OpenCL for GPU)
  – Offloading