CSCE 5160 Parallel Processing

Prerequisites:
- Good programming background in C, C++ (Java does not help)
- Programming on Unix systems (Windows is not useful)
- Understanding of complex data structures and algorithms
  - pointers, linked lists, multi-dimensional arrays
  - matrix multiplication, transpose etc
  - graph algorithms (shortest path), searching & sorting
- Understanding how a computer system works
  - CPU, Caches, Memory, Pipelines
  - Multithreading

Desirable:
- Programming using Parallel libraries
  - MPI, OpenMP and Pthreads
- Performance measurement
  - tracing tools, performance counters, PAPI
  - power measurement (MacPAT0)
- Complexity analysis – Big Oh notations
- Parallel Architectures and networks
CSCE 5160 Parallel Processing

Homework Assignment, Term Projects and Exams

I will assign small programming problems as homework assignments that require you to use MPI, OpenMP and/or Cuda/OpenCL.

There will be other homework assignments to test your understanding of the theory learned in class.

You can have access to CSE computers and UNT parallel computing system called Talon

Term Project: Design and Implement parallel programs for a scientific/engineering application in OpenMP, MPI, Cuda, OpenCL

Need to be reasonably complex (and something not covered in class)
Can’t be a simple program equivalent to one of the assignments
Should include performance evaluation and complexity analysis
use wall clock, CPU times or performance counters

Exams will be open book - open notes format.

Questions ask you to think on how you can apply what you learned in class; nothing to memorize

I will also provide sample exams with solutions.

Relative Grading: I grade one problem at a time, giving highest grade for the best answer

Why parallel computation?

- Most computers contain multiple cores – or CPU’s
  Clock speed is not increasing but you have multiple processors
  If you want better performance you need to parallelize computations

- Some applications cannot be solved within reasonable time using a single processor.
  Consider the weather forecasting example from textbook
  To forecast weather for for 48 hours, you need 300 hours of computations
CSCE 5160 Parallel Processing

Why parallel computation?

- With parallel processing, we may solve some problems with more accuracy or for larger data sizes -- because we obtain greater computational capability
  - BIG DATA

- We can handle multiple requests or jobs (throughput oriented parallelism)
  - Multiple webpages
  - Multiple windows
  - Multiple transactions

Levels of Parallelism -- ranges from hardware to algorithm level

Data paths (CSCE 2610 – or any computer organization course)

Most modern processors permit parallelism at hardware level multiple functional units (i.e., FP, Integer) multiple buses

CSCE 5160 Parallel Processing

Levels of Parallelism -- ranges from hardware to algorithm level

Instruction Level Parallelism (ILP) – CSCE 5610 or computer architecture course

Multiple instructions can be executed in parallel when multiple functional units and data paths are available

Most modern processors permit ILP
  - Superscalar processors and VLIW architectures
  - Out of order execution
  - Hyper-threading (or multithreading)

Pipelining -- Overlapped execution
  can be at instruction level or algorithm level

At instruction level, the instruction execution is divided into smaller tasks
  - Instruction Fetch, Instruction Decode, Operand Fetch
  - Execute, Memory Access, Write Results Back
CSCE 5160 Parallel Processing

Pipelining – algorithm level, we could divide the overall computation into smaller computations.

Consider the following code segment (we will see better examples later)
```java
for (i=0; i<n; i++) {
    sum[i] = 0.0;
    for (j=0; j<n; j++)
        sum[i] = sum[i]+a[i,j];
}
```

We can pipeline the inner loop (if we have multiple adder units)

---

CSCE 5160 Parallel Processing

Processor Level (CSCE 5610 – computer architecture)
- Multiple CPU's (multi-core)
  - Can be homogeneous or heterogeneous
- NVIDIA GPU/CPU
- Support for Multiple threads
  - more recently, Processing in Memory or Near Data Computations

Algorithm Level Parallelism (This class)
- Function or Task Parallelism (MIMD)
  - The algorithm is broken into sub tasks which can be executed in parallel
- Data parallelism (Array/SIMD/Vector)
  - Loop iterations are executed in parallel
  - Each iteration operates on a different array element
CSCE 5160 Parallel Processing

Algorithm Level Parallelism—continued
Data Parallelism (SPMD or SIMD mode)

Let us consider Matrix multiplication algorithm

```c
for (i = 0; i<n; i++)
  for (j= 0; j<n; j++) {
    C[i,j] = 0;
    for (k = 0; k< n; k++)
      C[i,j] = C[i,j] + A[i,k]*B[k,j]
  }
```

Note we have 3 loops and we can consider parallelizing any one of the loops
that is, execute each iteration of the loop independently and in parallel

We can think of parallelizing any of the loops

Consider "k" loop
each processor computes one multiplications

```
A[i,k]*B[k,j]
```

Any problems with this type of parallelization?
CSCE 5160 Parallel Processing

Let us try to parallelize j loop using OpenMP

for (i = 0; i<n; i++) {
    for (j=0; j<n; j++) {
        C[i,j] = 0;
        for (k = 0; k<n; k++)
            C[i,j] = C[i,j] + A[i,k]*B[k,j];
    }
}

In OpenMP, the pragma omp parallel for
the for loop that follows the pragma should be executed in parallel
In this example, for each value j, the loop will be assigned to a different
processor or thread

We will talk about OpenMP later and see to have individual copies of variable as well as
share data

We can also consider parallelizing i loop
or both i and j loops

CSCE 5160 Parallel Processing

Consider executing this using task parallel model – create tasks to execute in parallel

for (i = 0; i<n; i++) {
    for (j=0; j<n; j++) { Create (task_to_Compute(i,j));}
}

In this example, Create will spawn n tasks, each computing a different element of the result
matrix.

We will see how we can create the tasks on different processors to compute(i,j) using MPI as
well as OpenMP.

We need to learn how to send and receive data among the different processors
Flynn’s classification (most commonly used and dates back to 1966)
- Two fundamental computational entities that can be parallelized
  - Data and Instructions

SISD: Single Instruction stream operating on Single Data stream
  - (sequential processing systems)
SIMD: Single Instruction stream operating on Multiple Data streams
  - Identical instruction applied to different data elements
MISD: Multiple Instruction streams operating on Single Data stream
  - Not common, but could consider performing different types of data analysis using a single data object
    - For example, database type analyses using employee records
MIMD: Multiple Instruction streams operating on Multiple Data streams
  - Two types of MIMD:
    - Shared Memory (SMPs, most multi-core chips)
    - Message Passing (Multi-computers, clusters, grids)

SIMD Architectures (Array Processors)
Consider a program segment like
\[
\text{for } (i = 1; i < n; i++) \{ \text{A}(i) = \text{B}(i) \oplus \text{C}(i) \} \\
\]
For SIMD, loop index i becomes an AU number.

Consider our Data Parallel example for matrix multiplication
\[
\text{for } (i = 0; i < n; i++) \\
\text{\quad C}_{i,\text{own}} = 0; \\
\text{\quad for } (k = 0; k < n; k++) \\
\text{\quad C}_{i,\text{own}} = \text{C}_{i,\text{own}} + \text{A}_{i,k} \times \text{B}_{k,\text{own}}
\]
We need more than one index -- local and global indexes
For example, i will be global index, k will be local index
CSCE 5160 Parallel Processing

SIMD - Array Processors

Consider executing a conditional code segment

\[
\text{if (condition) then} \\
\text{stmt-1} \\
\text{else stmt-2}
\]

We disable all ALU’s that fail condition during first round, and execute stmt-1 on all ALU’s that satisfy the condition.

Then we disable all ALU’s that meet condition during the second round and execute stmt-2 on ALU’s that failed the condition.

Thus, conditional statements need 2-CYCLES to complete.

Array processors are very specialized and are useful only in very specialized applications.

Early implementations were not commercially successful.

---

CSCE 5160 Parallel Processing

There is a recent surge in using Array processors as attached processors for DSP and multimedia applications.

Also many graphics processors such as NVIDIA GPUs can be viewed as array processors

**Multimedia instruction sets (or AVX)** allow us view a 64-bit data as 8 8-bit array elements or 4 16-bit or 2 32-bit elements

A 64 bit ALU can be configured to perform 8, 4, 2 array operations

More generally we can rely on SPMD – single program multiple data approach

We use MIMD type processors to execute the same program (or code) using different data sets.

Unlike early SIMD, the individual processors are *not tightly* synchronized to execute the same instruction on every clock cycle.

Individual processors synchronize only after completing their task (or code).
CSCE 5160 Parallel Processing

Vector processors vs Array Processors

What is a vector processor as opposed array processor?
Vector instructions, for example, Vector ADD
What are the operands?
Vector registers: each register contains “n” element

Cray Vector Unit
Vector Registers (64 words per register)
VADD Vk, Vi, Vj

Here, each V is a 64-register array and the addition is actually 64 additions

Cray-1 (1976)

Single Port Memory
16 banks of 64-bit words + 8-bit SECDED
80MW/sec data load/store
320MW/sec instruction buffer refill

memory bank cycle 50 ns  processor cycle 12.5 ns (80MHz)
<table>
<thead>
<tr>
<th><strong>Cray-1 (1976)</strong></th>
<th><strong>Vector Code Example</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td># C code</td>
<td># Scalar Code</td>
</tr>
<tr>
<td>for (i=0; i&lt;64; i++)</td>
<td>LI R4, 64</td>
</tr>
<tr>
<td>C[i] = A[i] + B[i];</td>
<td>loop:</td>
</tr>
<tr>
<td></td>
<td>L.D F0, 0(R1)</td>
</tr>
<tr>
<td></td>
<td>L.D F2, 0(R2)</td>
</tr>
<tr>
<td></td>
<td>ADD.D F4, F2, F0</td>
</tr>
<tr>
<td></td>
<td>S.D F4, 0(R3)</td>
</tr>
<tr>
<td></td>
<td>DADDIU R1, 8</td>
</tr>
<tr>
<td></td>
<td>DADDIU R2, 8</td>
</tr>
<tr>
<td></td>
<td>DADDIU R3, 8</td>
</tr>
<tr>
<td></td>
<td>DSUBIU R4, 1</td>
</tr>
<tr>
<td></td>
<td>BNEZ R4, loop</td>
</tr>
<tr>
<td></td>
<td>LI VLR, 64</td>
</tr>
<tr>
<td></td>
<td>LV V1, R1</td>
</tr>
<tr>
<td></td>
<td>LV V2, R2</td>
</tr>
<tr>
<td></td>
<td>ADDV.D V3, V1, V2</td>
</tr>
<tr>
<td></td>
<td>SV V3, R3</td>
</tr>
</tbody>
</table>

Cray-1 (1976)
MIMD Systems

Two Types: Message Passing and Shared Memory

Cluster Computing, Grid Computing, Cloud Computing, Big Data, MapReduce

These are basically Distributed memory (or message passing) MIMD systems.

In traditional message passing systems, it is assumed that all nodes in the system already have the program (or code) resident in their local memories. Only data is exchanged via messages.

In cluster/Cloud computing, we cannot make such an assumption. We need to communicate both code and data using messages.

This leads to several interesting issues regarding the granularity of a task that should be executed at each node

Need to understand overhead of transmitting code and data

May need multi-level scheduling issues

A group of task will be assigned a cluster of nodes then we need to schedule individual tasks on individual nodes
CSCE 5160 Parallel Processing

UMA and NUMA systems

Even among Shared memory MIMD, we can distinguish between Uniform Memory Access (UMA) and Non-Uniform Memory Access (NUMA) systems.

In UMA, any location in the global memory is directly accessible to all processors. Hence accessing any location will take the “same” amount of time --- Uniform access.

Bus based Shared Memory systems fall in this category. However, due to the contention on bus, such systems can only support a small number of processors (e.g., 4, 8 or 16).

In NUMA, although the entire memory is accessible to all processors, access to some memory locations is faster than other locations. Thus, we may have the concept of local and remote memory. Most non-bus based architectures fall in this category.

Large caches are now being designed as NUMA (called NUCA caches).

CSCE 5160 Parallel Processing

Memory latencies limit the performance of a parallel processing system
See Example 2.2 on page 17

1GHz CPU (1 ns cycle time) but DRAM latency is 100ns (no cache) has two multiply-add units (4 operations per cycle at peak) We should expect 4 Giga Flops

The best performance we can get is limited by memory to 10M FLOPS since we can read/write one data item in 100ns

Cache memories can help in reducing the latencies
If we use a cache, we can retrieve data at (almost) the same speed as CPU Cache misses are the limiting factor So we need to reduce cache misses!

Caches reduce latencies but require higher bandwidths – what are these?

Latency: Time to get the "first" bit of data
Bandwidth: Number of bits per second
CSCE 5160 Parallel Processing

Consider the example 2.4 on page 18

If we can only get one word at a time from memory, bandwidth is 1 word per 100 cycles (cache block or line size = 1 word)

\[ \text{latency is 100ns} \]

If we can get 4 word at a time from memory (cache block or line size = 4 words)

\[ \frac{4 \text{ words}}{100 \text{ns}} \text{ or bandwidth = 40M words per second} \]

\[ \text{but latency is still 100ns} \]

Consider the dot-product (multiplying two vectors)

\[ \text{sum} = \sum_{} a[i] \times b[i] \]

Get 4 a[i]s and 4 b[i]'s in 200 cycles – and perform 8 operations (4 multiplies and 4 adds)

8 operations in 200ns or get 40 MFLOPS

This can sometimes be related to cache misses

you need to get a new 4-word block once every 4 accesses

or we can say there is a 25% miss rate

---

CSCE 5160 Parallel Processing

Reducing Latency

Other ways of reducing latency

Multithreading and pre-fetching data

Let us start with a single thread

- Consider a single CPU and one thread
  - When a thread issues a "long latency" operation, the thread is blocked and CPU idles
- Let \( L \) be amount of time needed for the long latency operation, and let \( R \) be the average amount of time between such latency operations
  - Utilization without threads \( U_t = \frac{R}{R+L} \)
Reducing Latency

- **Multithreading**
  - Consider a single CPU supporting multiple threads
    - When a thread issues a "long latency" operation, the thread is context switched and a new thread starts execution.
  - In addition to $R$ and $L$, let us assume that $C$ is the time needed to context switch.

![Diagram showing Useful Work and Waiting time]

Useful Work | Waiting | Useful Work | Waiting | Useful Work
---|---|---|---|---
$R$ | | $L$ | | $C$

**Useful Work** = Waiting + Useful Work

CSCE 5160 Parallel Processing

Reducing Latency

- **Multithreading**
  - What is the maximum Utilization that is possible?
    - Maximum utilization with threads $U_{\text{max}} = \frac{R}{R+C}$
      - $C$ is the context switching overhead
  - How many threads are needed to reach $U_{\text{max}}$?
    - $N_{\text{saturated}} = \frac{R+L}{R+C}$
  - If there are fewer threads?
    - Utilization with $N$ threads $U_N = \frac{N \cdot R}{R+L}$