

CSCE 5160 Parallel Processing

Prerequisites:

Good programming background in C, C++

Programming on Unix systems

Understanding of complex data structures and Algorithms

graph algorithms, searching & sorting

Understanding how a computer system works

CPU, Memory, Pipelines

Desirable:

Programming using Parallel libraries

MPI, OpenMP and Pthreads

Performance measurement

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.

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

Term Project : Design and Implement parallel programs for a scientific/engineering application

Need to be reasonably complex

Should include performance evaluation and complexity analysis

Exams will be open book - open notes format.

May ask you to think how you will apply what you learned in class

I will also provide sample exams with solutions.

CSCE 5160 Parallel Processing

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 parallel 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

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

- We can handle multiple requests or jobs.
 - Multiple webpages
 - Multiple windows

CSCE 5160 Parallel Processing

Levels of Parallelism -- ranges from hardware to algorithm level

Data paths

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

Instruction Level Parallelism (ILP)

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

Hyperthreading (or multithreading)

CSCE 5160 Parallel Processing

Levels of Parallelism

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

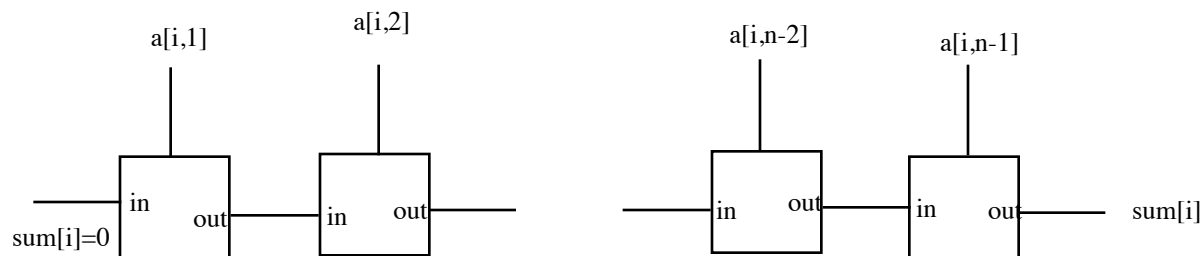
Execute, Memory Access, Write Results Back

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

Consider the following code segment (we will see better examples later)

```
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

- Multiple CPU's (multi-core)
- Cell like processors (with PPE and SPE's)
- NVIDIA GPU/CPU
- Support for Multiple threads

Algorithm Level Parallelism

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 the Matrix multiplication again

```
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]  
        }  
    }
```

We can think of parallelizing any of the loops

Consider “k” loop

each processor computes one multiplications

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

CSCE 5160 Parallel Processing

Let us try to parallelize j loop using OpenMP

```
for (i = 0; i < n; i++) {  
    #pragma omp parallel for  
    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` states the the loop that follows should be executed in parallel

That is, for each value j, the loop will be assigned to a different processor

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

CSCE 5160 Parallel Processing

Consider executing this using task parallel (pseudo code)

```
for (i = 0; i < n; i++)  
    for (j = 0; j < n; j++) { Create_task_to_Compute(i,j); }
```

In this example, **Create_task_to_Compute(i,j)** 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.

We need to learn how to send and receive data among the different processors

CSCE 5160 Parallel Processing

Flynn's classification (most commonly used and dates back to 1966)

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)

More on SIMD and MIMD

SIMD Architectures (Array Processors)

Consider a program segment like

```
for ( i= 1; i < n; i ++ ) { A(i) = B(i) op C(i)}
```

For SIMD, loop index i becomes an AU number.

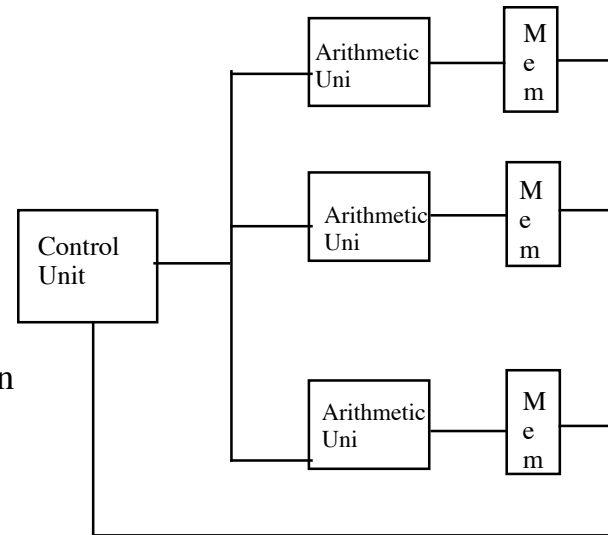
Consider our Data Parallel example for matrix multiplication

```
for ( i = 0; i < n; i ++ )
```

```
    C[i,own] = 0;
```

```
    for ( k = 0; k < n; k ++ )
```

```
        C[i,own] = C[i,own] + A[i,k]*B[k,own]
```



We need more than one index -- local and global indexes

For example, i will be global index, k will be local index

Note j is not specified as an index, but each arithmetic unit will use its number as j index

SIMD -Array Processors

Consider executing a conditional code segment

```
    if (condition) then
        stmt-1
    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.

SIMD -Array Processors

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

Also many graphics processors such as Cell have an array of SPEs and one PPE
SPE's can execute like SIMD

Multimedia instruction sets allow to 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).

Vector processors vs Array Processors

What is a vector processor as opposed array processor?

Vector instructions, for example, Vector ADD

What are the operands?

CDC Star 100

VADD Ra, Ri, Rb, Rj, Rc, Rk, Rn

This indicates an ADD instruction which is repeated as a loop

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

The indexes are specified in registers: Ri, Rj, Rk and Rn contain i,j,k, n

Array addresses are also specified in Registers Ra, Rb and Rc

Once this instruction is issued, the Vector processor completes the entire loop

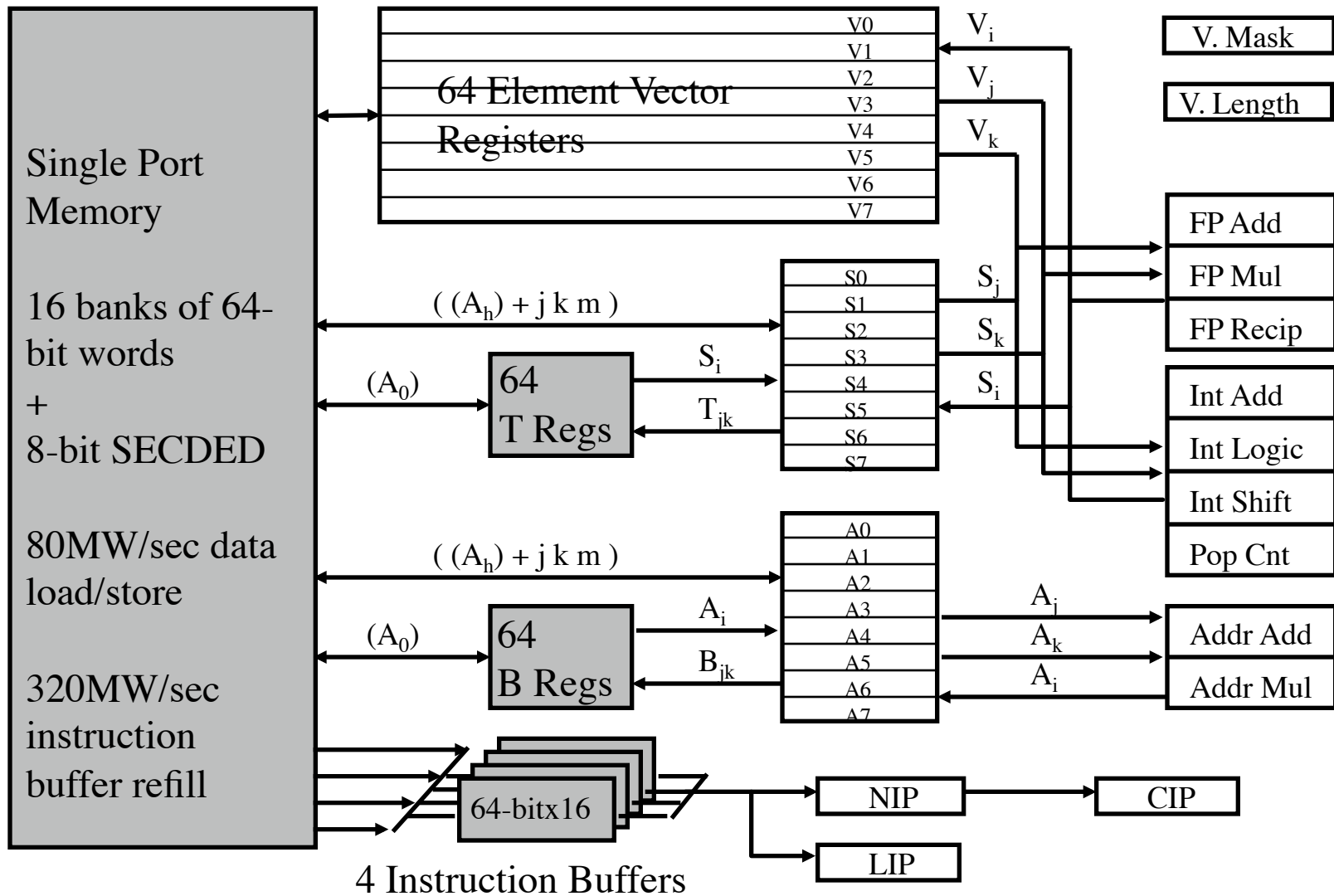
Cray Vector Unit

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)



memory bank cycle 50 ns processor cycle 12.5 ns (80MHz)

Cray-1 (1976)

Vector Code Example

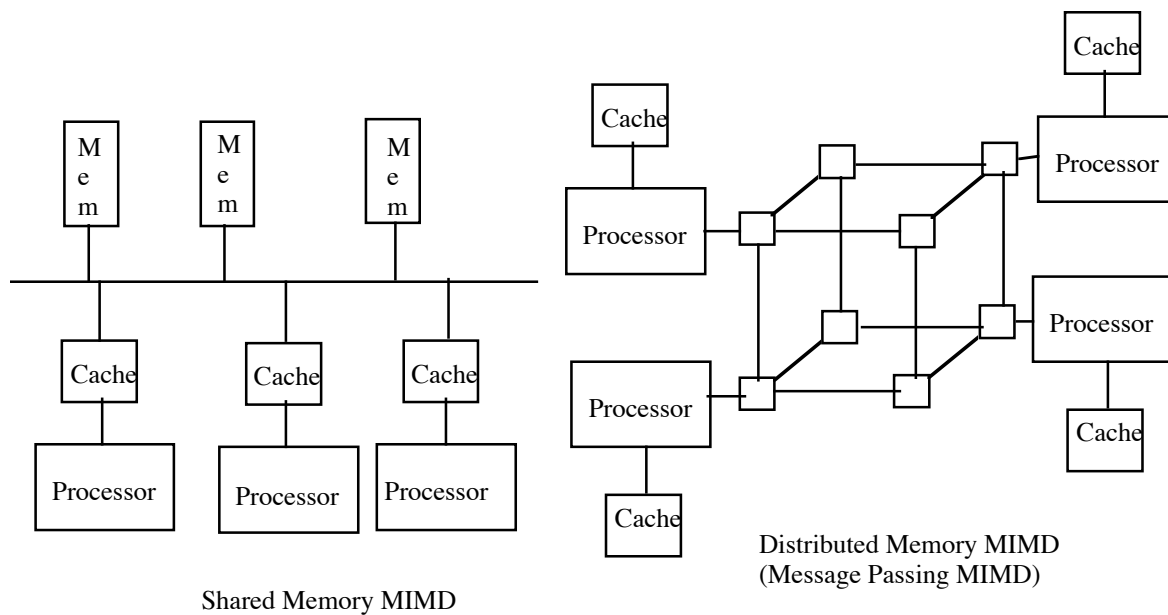
```
# C code
for (i=0; i<64; i++)
    C[i] = A[i] + B[i];
```

```
# Scalar Code
LI R4, 64
loop:
L.D F0, 0(R1)
L.D F2, 0(R2)
ADD.D F4, F2, F0
S.D F4, 0(R3)
DADDIU R1, 8
DADDIU R2, 8
DADDIU R3, 8
DSUBIU R4, 1
BNEZ R4, loop
```

```
# Vector Code
LI VLR, 64
LV V1, R1
LV V2, R2
ADDV.D V3, V1, V2
SV V3, R3
```

MIMD Systems

Two Types: Message Passing and Shared Memory



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.

MIMD vs Cluster Computing

Cluster Computing and Grid Computing

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 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

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