**National Institute of Technology Karnataka Surathkal**
**Department of Information Technology**

# IT 301 Parallel Computing

## Introduction

**Dr. Geetha V**

*Assistant Professor*

*Dept of Information Technology*

*NITK Surathkal*

# Index

- 1: Moore's Law and Need for Parallel Processing
- 2. Parallel Processing in Uniprocessor System
  - Bit Level , Instruction level, Task level and Data level parallelism
- 3: Multiprocessor and Multicores System

# 1: Introduction

**Course Plan: Theory:**

**Part A: Parallel Computer Architectures**

Week 1,2,3: ***Introduction to Parallel Computer Architecture:***

Parallel Computing,

Parallel architecture,

bit level, instruction level , data level and task level parallelism.

Instruction level parallelisms: pipelining(Data and control instructions),

scalar and superscalar processors,

vector processors.

Parallel computers and computation.

# 1: Moore's Law and Need for Parallel Processing

- Chip performance doubles every 18-24 months
- Power consumption is proportional to frequency of the system
- Limitations of serial Computing
  - Heating issues
  - Limit to transmission speeds
  - Leakage currents
  - Limit to miniaturization
    [downsizing of electronic devices]

- Smaller transistor => faster processors
- Faster processor => increased power consumption
- Increased power consumption => increased heat
- Increased heat => unreliable processors

- Parallel processing in Uniprocessor systems
  - Bit level parallelism, Instruction level parallelism (pipelining, Superscalar processors), Data parallelism (vector processors) and task level parallelism (threads)
- Parallel processing in Multiprocessor and Multicore systems
  - Multi core processor, clusters, grids

# 2. Parallel Processing in Uniprocessor System

- Bit level parallelism
- Instruction level parallelism (pipelining, Superscalar processors)
- Data parallelism (vector processors)
- Task level parallelism (threads)

# 2. Parallel Processing in Uniprocessor System

- Bit level parallelism
- Adding 16-bit number in 8-bit processor

http://ryanohs.com/2015/09/high-level-architecture/

- Adding 16-bit number in 16-bit processor

http://www.c-jump.com/CIS77/CPU/VonNeumann/lecture.html#V77_0010_von_neumann

# 2. Parallel Processing in Uniprocessor System

- Bit Level Parallelism (eg. 1516H +2829H)
  - Adding 16-bit number in 8-bit processor

```
MVI A, #16H
MVI B, #29H
ADD B
MOV C, A
MVI A, #15H
MVI B, #28H
ADC B
MOV D, A
```

  - Adding 16-bit number in 16-bit processor

```
MVI AX, #1516H
MVI BX, #2829H
ADD AX, BX
MOV CX, AX
```

# 2. Parallel Processing in Uniprocessor System

- Bit Level Parallelism
  - 8-bit processor (1972-1974)

    Intel 8008, Intel 8080
  - 16-bit processor (1979 – 1982)

    Intel 8086, Intel 286
  - 32-bit processor (1985 -2006)

    Intel 386, Intel 486, P5 (Pentium)

    Pentium Pro, Pentium II, Pentium III, Pentium M, Intel Core, Dual-core Xeon LV
  - 64-bit processor (2004 onwards)

    Itanium , Itanium 2 (IA-64), Pentium 4F, Pentium D, Intel 6: Xeon (NetBurst), Intel Core 2, Intel Pentium Dual Core, Celeron, Celeron M; Nehalem: Intel Pentium, Core i3, i5, i7, Xeon…..etc

# 2. Parallel Processing in Uniprocessor System

- Instruction level parallelism (pipelining, Superscalar processors)

- Scalar Processors
  - Goal towards one instruction execution per cycle

https://en.wikipedia.org/wiki/Instruction_pipelining

- Superscalar Processors
  - Multiple Instruction per cycle

http://kkucoecomarch2016-2.blogspot.com/2017/02/mapping-function.html

# 2. Parallel Processing in Uniprocessor System

- Instruction level parallelism (pipelining, Superscalar processors)
  - Scalar Processors
    - Goal towards one instruction execution per cycle
    - Reduced Instruction Set Computer (RISC) Scalar processors
      - Intel i860, Motorola MC8810, SUN's SPARC CY7C601 etc
    - Complex Instruction Set Computer (CISC) scalar Processors
      - Intel 386, 486; Motorola's 68030, 68040; etc
  - Superscalar Processors
    - Multiple Instruction per cycle
      - Pentium , Pentium Pro, Pentium II, Pentium III Motorola 88110 etc.

# 2. Parallel Processing in Uniprocessor System

- Task level parallelism

Task parallelism or function level parallelism is a form of parallelization of computer code across multiple processors in parallel computing environment

In Uniprocessor system, Thread level parallelism is used.

- Models
  - Task dependency graph model
  - Master Slave Model
  - Pipeline/Producer Consumer model

- Data level parallelism

Data parallelism is a form of parallelization of computing across multiple processors in parallel computing environment.

Vector processors or coprocessors are added in case of Uniprocessor System

- Architectures
  - Vector Processors
  - Single Instruction Multiple Data (SIMD)

# 2. Parallel Processing in Uniprocessor System

- Task level parallelism

Intel Xeon hyperthreading

- Data level parallelism

- Vector Processor

Eg. Cray 1

- SIMD

Eg. ILLIAC IV

https://slideplayer.com/slide/14851286/

https://www.researchgate.net/profile/Ahmed_Al-Araji/publication/342107319/figure/fig1/AS:901279303864320@1591893139023/Figure-1-Big-picture-of-SIMD-architecture.ppm
https://www.ausairpower.net/XIMG/vector-1.png

# 3: Multiprocessor and Muticore system

- Multiprocessor system: Two or more CPU within single computer system

- Multicore processor: Multiple Execution units (cores) of the same chip.

# 3: Multiprocessor and Muticore system

- Multiprocessor system: Two or more CPU within single computer system, sharing common memory and peripherals

- Multicore processor: Multiple Execution units (cores) of the same chip.

# 3: Multiprocessor and Muticore system

- Single Core: One execution unit in one chip
- Multiprocessor : Two or more CPU within single computer system
- Hyperthreading : Task level parallelism
- Multicore processor: Multiple Execution units (cores) on the same chip.
- Multicore with shared cache: All core shares same cache
- Multicore with Hyperthreading: task level parallelism

# Hyper Threading and Multiple Threads

# Reference

**Textbooks and/or Reference Books:**

1. Professional CUDA C Programming – John Cheng, Max Grossman, Ty McKercher, 2014

2. Wilkinson, M. Allen, ”Parallel Programming: Techniques and Applications Using Networked Workstations and Parallel Computers”, Pearson Education, 1999

3. I. Foster, ”Designing and building parallel programs”, 2003

4. Parallel Programming in C using OpenMP and MPI – Micheal J Quinn, 2004

5. Introduction to Parallel Programming – Peter S Pacheco, Morgan Kaufmann Publishers, 2011

6. Advanced Computer Architectures: A design approach, Dezso Sima, Terence Fountain, Peter Kacsuk, 2002

7. Parallel Computer Architecture : A hardware/Software Approach, David E Culler, Jaswinder Pal Singh Anoop Gupta, 2011 8. Introduction to Parallel Computing, Ananth Grama, Anshul Gupta, George Karypis, Vipin Kumar, Pearson, 2011

# Thank You

**National Institute of Technology Karnataka Surathkal**
**Department of Information Technology**

# IT 301 Parallel Computing

## Memory Models, Flynn's Classification

**Dr. Geetha V**

*Assistant Professor*

*Dept of Information Technology*

*NITK Surathkal*

# Course Outline

**Course Plan: Theory:**

**Part A: Parallel Computer Architectures**

Week 1,2,3: ***Introduction to Parallel Computer Architecture:*** Parallel Computing, Parallel architecture, bit level, instruction level , data level and task level parallelism. Instruction level parelllelism: pipelining(Data and control instructions), scalar and superscalar processors, vector processors. Parallel computers and computation.

Week 4,5: Memory Models: UMA, NUMA and COMA. Flynns classification, Cache coherence,

Week 6,7: Amdahl's Law. Performance evaluation, Designing parallel algorithms : Divide and conquer, Load balancing, Pipelining.

Week 8 -11: ***Parallel Programming techniques like Task Parallelism using TBB, TL2, Cilk++ etc. and software transactional memory techniques.***

# Course Outline

**Part B: OpenMP/MPI/CUDA**

Week 1,2,3 : ***Shared Memory Programing Techniques:*** Introduction to OpenMP : Directives: parallel, for, sections, task, single, critical, barrier, taskwait, atomic. Clauses: private, shared, firstprivate, lastprivate, reduction, nowait, ordered, schedule, collapse, num_threads, shared, if().

Week 4,5: ***Distributed Memory programming Techniques:*** MPI: Blocking, Non-blocking.

Week 6,7 : CUDA : OpenCL, Execution models, GPU memory, GPU libraries.

Week 10,11,: ***Introduction to accelerator programming using CUDA/OpenCL and Xeon-phi. Concepts of Heterogeneous programming techniques.***

**Practical:**

Implementation of parallel programs using OpenMP/MPI/CUDA.

**Assignment:** Performance evaluation of parallel algorithms (in group of 2 or 3 members)

# Index

1. Flynn's classification
2. Memory models
3. Perspective on parallel programming

# 1. Flynn's Classification

- In 1966, M.J. Flynn proposed a classification for the organization of a computer system by the number of instructions and data items that are manipulated simultaneously.
- Flynn uses the stream concept for describing a machine's structure.
- The sequence of instructions read from memory constitutes an **instruction stream**.
- The operations performed on the data in the processor constitute a **data stream**.

# Flynn's Classification

The classification of computer architectures based on the number of instruction streams and data streams (Flynn's Taxonomy)

Flynn's Classification divides computers into four major groups .

- Single instruction stream, single data stream (SISD)
- Single instruction stream, multiple data stream (SIMD)
- Multiple instruction stream, single data stream (MISD)
- Multiple instruction stream, multiple data stream (MIMD)

# Flynn's Classification

Flynn's Classification divides computers into four major groups .

Ref:https://www.javatpoint.com/flynns-classification-of-computers

# Flynn's Classification: SISD

Flynn's Classification

Ref:https://www.javatpoint.com/flynns-classification-of-computers

Example: Traditional Computers

Ref:https://www.futurelearn.com/courses/how-computers-work/0/steps/49283

# Flynn's Classification: SIMD

Flynn's Classification

Ref:https://www.javatpoint.com/flynns-classification-of-computers

Example: Vector processor CRAY -I

# Flynn's Classification: MISD

Flynn's Classification

Example: Systolic arrays : eg: 3 x 3 matrix multiplication

T1 Cycle

Ref:https://www.javatpoint.com/flynns-classification-of-computers

https://www.youtube.com/watch?v=vADVh1ogNo0

## T2 Cycle

## T3 Cycle

T4 Cycle

T5 Cycle

T6 Cycle

T7 Cycle

T8 Cycle

# Flynn's Classification MIMD

Flynn's Classification

Ref:https://www.javatpoint.com/flynns-classification-of-computers

# 2.Memory Models

## Parallel Computers Architectural Model/ Physical Model

Distinguished by having-

**1. Shared Common Memory:**

Three Shared-Memory Multiprocessor Models are:

i. UMA (Uniform-Memory Access)

ii. NUMA (Non-Uniform-Memory Access)

iii. COMA (Cache-Only Memory Architecture)

**2. Unshared Distributed Memory**

i. CC-NUMA (Cache-Coherent -NUMA)

# UMA Multiprocessor Model

# NUMA - Memory Models

(a) Shared local memories

LM – Local Memory
P - Local Processor

# NUMA - Memory Models

(b) A hierarchical cluster model

(Access of Remote Memory)

P – Processor
CSM – Cluster Shared Memory
CIN – Cluster Interconnection Network
GSM – Global Shared Memory

# COMA Multiprocessor Model

C – Cache
P – Processor
D - Directories

- Distributed Main Memory converted to Cache
- Cache form Global Address Space
- Remote Cache access by – Distributed cache Directories

# 3. Perspective on parallel programming

- Motivating Problems
- Process of creating a parallel program

# 3. Perspective on parallel programming

- Motivating Problems : Simulating Ocean Currents
  - Model as two dimensional grid :
    - Discretize in space and time
    - Finer and temporal resolution => greater accuracy
  - Many different computations per time step

(a) Cross sections

(b) Spatial discretization of a cross section

# 3. Perspective on parallel programming

- Motivating Problems : Simulating interactions of many stars evolving over time.
  - Computing forces is expensive

  Eg. Stars on which forces of other elements need to be computed
  - Many time steps, plenty of concurrency across stars within each step.

# 3. Perspective on parallel programming

- Creating a parallel program
  - Identify the work that can be done in parallel : computation, data access and I/O
  - Partition of work and perhaps data among processes
  - Manage data access, communication and synchronization

# 3. Perspective on parallel programming

- **Decomposition** of computation in tasks
- **Assignment** of tasks to processes
- **Orchestration** of data access, comm, synch.
- **Mapping** processes to processors

# Reference

**Text Books and/or Reference Books:**

1. Professional CUDA C Programming – John Cheng, Max Grossman, Ty McKercher, 2014

2. B.Wilkinson, M.Allen, ”Parallel Programming: Techniques and Applications Using Networked Workstations and Parallel Computers”, Pearson Education, 1999

3. I.Foster, ”Designing and building parallel programs”, 2003

4. Parallel Programming in C using OpenMP and MPI – Micheal J Quinn, 2004

5. Introduction to Parallel Programming – Peter S Pacheco, Morgan Kaufmann Publishers, 2011

6. Advanced Computer Architectures: A design approach, Dezso Sima, Terence Fountain, Peter Kacsuk, 2002

7. **Parallel Computer Architecture : A hardware/Software Approach, David E Culler, Jaswinder Pal Singh Anoop Gupta, 2011 8. Introduction to Parallel Computing, Ananth Grama, Anshul Gupta, George Karypis, Vipin Kumar, Pearson, 2011**

# Thank You

**National Institute of Technology Karnataka Surathkal**
**Department of Information Technology**

# IT 301 Parallel Computing

## Scalar Processors – Instruction Pipeline

**Dr. Geetha V**

*Assistant Professor*

*Dept of Information Technology*

*NITK Surathkal*

# Index


- 1: Instruction Pipeline - Introduction
- 2. Five Stage pipeline design
- 3: Issues of pipeline and Design Solutions
- 4: Performance evaluation of pipelining

# 1: Introduction

**Course Plan: Theory:**

**Part A: Parallel Computer Architectures**

Week 1,2,3: ***Introduction to Parallel Computer Architecture:***

Parallel Computing,

Parallel architecture,

bit level, instruction level , data level and task level parallelism.

**Instruction level parallelisms: pipelining (Data and control instructions),**

**scalar processors** and superscalar processors,

vector processors.

Parallel computers and computation.

# 1: Instruction Pipeline - Introduction

## Features

- Established by John von Neumann in 1945.
- Stored Program Concept
- **PC :** Program Counter
- **CIR:** Current Instruction Register
- **AC:** Accumulator,
- **MAR:** Memory Address Register,
- **MDR:** Memory Data Register
- **Buses:** Address, Data and Control
- **Execution :** One Instruction at a Time

## Von Neumann Architecture

https://www.computerscience.gcse.guru/theory/von-neumann-architecture

# 1: Instruction Pipeline - Introduction

## Features of RISC

- **RISC:** Reduced Instruction Set Computer
- Fixed **instruction size**
- **Load/Store** Memory Operation
- Few **Addressing Modes**
- **Register to Register** operation

- These features leads towards **pipelined execution**

## Von Neumann Architecture

https://www.computerscience.gcse.guru/theory/von-neumann-architecture

# 2. Five Stage Pipeline Design

- Fetch Stage: The address in program counter is sent to Instruction register and instruction is fetched from memory. PC is incremented.
- Decode Stage: The operands are fetched from register file; Sign extension done if necessary.
- Execution Stage: The instruction is executed.
- Memory Stage: The Load and Store instructions execution.
- Write Back Stage: The results are written back to register file.

| | | | | |
|---|---|---|---|---|
| **Fetch (F)** | **Decode (D)** | **Execute (E)** | **Memory Access (M)** | **Write Back (W)** |

# 2. Five Stage Pipeline Design

Image from Wikimedia

# 2. Five Stage Pipeline Design

| | Instruction | (F) | (D) | (E ) | (M) | (W) | |
|---|---|---|---|---|---|---|---|
| $I_1$ | Move R5, #2000h | $I_1$ | | | | | |
| $I_2$ | Move R1, #15 | $I_2$ | $I_1$ | | | | |
| $I_3$ | Move R2, #16 | $I_3$ | $I_2$ | $I_1$ | | | |
| $I_4$ | Move R3, R1 | $I_4$ | $I_3$ | $I_2$ | $I_1$ | | |
| $I_5$ | Add R1, R3 | $I_5$ | $I_4$ | $I_3$ | $I_2$ | $I_1$ | I1: Complete |
| $I_6$ | Move R4, R2 | $I_6$ | $I_5$ | $I_4$ | $I_3$ | $I_2$ | I2: Complete |
| $I_7$ | Add R2, R4 | $I_7$ | $I_6$ | $I_5$ | $I_4$ | $I_3$ | I3: Complete |
| $I_8$ | Add R1, R2 | $I_8$ | $I_7$ | $I_6$ | $I_5$ | $I_4$ | I4: Complete |
| $I_9$ | Store [R5], R1 | $I_9$ | $I_8$ | $I_7$ | $I_6$ | $I_5$ | I5: Complete |
| $I_{10}$ | | | $I_9$ | $I_8$ | $I_7$ | $I_6$ | I6: Complete |
| $I_{11}$ | | | | $I_9$ | $I_8$ | $I_7$ | I7: Complete |
| $I_{12}$ | | | | | $I_9$ | $I_8$ | I8: Complete |
| $I_{13}$ | | | | | | $I_9$ | I9: Complete |

- Adding (2x+2y)
- x=15 y=16
- If no pipeline, then this program takes

**9 x 5 = 45 cycles**

- If 5 stage pipeline is used, then it takes

**5 + 8 = 13 cycles**

(first instruction takes 'n' stages/cycle) + (n-1) instructions

# 3: Issues of pipeline and Design Solutions

| | **Instruction** | (F) | (D) | (E ) | (M) | (W) | |
|---|---|---|---|---|---|---|---|
| $I_1$ | Move R5, #2000h | $I_1$ | | | | | |
| $I_2$ | Move R1, #15 | $I_2$ | $I_1$ | | | | |
| $I_3$ | Move R2, #16 | $I_3$ | $I_2$ | $I_1$ | | | |
| $I_4$ | Move R3, R1 | $I_4$ | $I_3$ | $I_2$ | $I_1$ | | |
| $I_5$ | Add R1, R3 | $I_5$ | $I_4$ | $I_3$ | $I_2$ | $I_1$ | I1: Complete |
| $I_6$ | Move R4, R2 | $I_6$ | $I_5$ | $I_4$ | $I_3$ | $I_2$ | I2: Complete |
| $I_7$ | Add R2, R4 | $I_7$ | $I_6$ | $I_5$ | $I_4$ | $I_3$ | I3: Complete |
| $I_8$ | Add R1, R2 | $I_8$ | $I_7$ | $I_6$ | $I_5$ | $I_4$ | I4: Complete |
| $I_9$ | Store [R5], R1 | $I_9$ | $I_8$ | $I_7$ | $I_6$ | $I_5$ | I5: Complete |
| $I_{10}$ | | | $I_9$ | $I_8$ | $I_7$ | $I_6$ | I6: Complete |
| $I_{11}$ | | | | $I_9$ | $I_8$ | $I_7$ | I7: Complete |
| $I_{12}$ | | | | | $I_9$ | $I_8$ | I8: Complete |
| $I_{13}$ | | | | | | $I_9$ | I9: Complete |

- **Data Dependency**

Dependency of instructions due to operand values unavailability

- **Control Dependency**

Dependency due to unresolved decision on control instructions.

# 3: Issues of pipeline and Design Solutions

**Cycle 1: I1 in fetch stage**

**I1:Move R5, #2000h**

| | |
|---|---|
| $I_1$ | Move R5, #2000h |
| $I_2$ | Move R1, #15 |
| $I_3$ | Move R2, #16 |
| $I_4$ | Move R3, R1 |
| $I_5$ | Add R1, R3 |
| $I_6$ | Move R4, R2 |
| $I_7$ | Add R2, R4 |
| $I_8$ | Add R1, R2 |
| $I_9$ | Store [R5], R1 |

Image from Wikimedia

# 3: Issues of pipeline and Design Solutions

**Cycle 2: I1 in Decode stage, I2 in fetch stage**

**I2: Move R1, #15**

**I1: Move R5, #2000h**

| | |
|---|---|
| $I_1$ | Move R5, #2000h |
| $I_2$ | Move R1, #15 |
| $I_3$ | Move R2, #16 |
| $I_4$ | Move R3, R1 |
| $I_5$ | Add R1, R3 |
| $I_6$ | Move R4, R2 |
| $I_7$ | Add R2, R4 |
| $I_8$ | Add R1, R2 |
| $I_9$ | Store [R5], R1 |

Image from Wikimedia

# 3: Issues of pipeline and Design Solutions

**Cycle 3: I1 in Execute stage, I2 in Decode stage, I3 in fetch stage**

| | |
|---|---|
| $I_1$ | Move R5, #2000h |
| $I_2$ | Move R1, #15 |
| $I_3$ | Move R2, #16 |
| $I_4$ | Move R3, R1 |
| $I_5$ | Add R1, R3 |
| $I_6$ | Move R4, R2 |
| $I_7$ | Add R2, R4 |
| $I_8$ | Add R1, R2 |
| $I_9$ | Store [R5], R1 |

Image from Wikimedia

# 3: Issues of pipeline and Design Solutions

**Cycle 4: I1 in memory access stage, I2 in execute , I3 in Decode and I4 in fetch stage**

| | |
|---|---|
| $I_1$ | Move R5, #2000h |
| $I_2$ | Move R1, #15 |
| $I_3$ | Move R2, #16 |
| $I_4$ | Move R3, R1 |
| $I_5$ | Add R1, R3 |
| $I_6$ | Move R4, R2 |
| $I_7$ | Add R2, R4 |
| $I_8$ | Add R1, R2 |
| $I_9$ | Store [R5], R1 |

**I4: Move R3, R1** **I3: Move R2, #16** **I2: Move R1, #15** **I1: Move R5, #2000h**

Image from Wikimedia

# 3: Issues of pipeline and Design Solutions

**Cycle 5: R1 data must be Written from I1 to Register file ; R1 Data must be read by Instruction 4.**

| | |
|---|---|
| $I_1$ | Move R5, #2000h |
| $I_2$ | Move R1, #15 |
| $I_3$ | Move R2, #16 |
| $I_4$ | Move R3, R1 |
| $I_5$ | Add R1, R3 |
| $I_6$ | Move R4, R2 |
| $I_7$ | Add R2, R4 |
| $I_8$ | Add R1, R2 |
| $I_9$ | Store [R5], R1 |

Image from Wikimedia

# 3: Issues of pipeline and Design Solutions

**Cycle 5: R1 data must be Written from I1 to Register file ; R1 Data must be read by Instruction 4.**

**Data Dependency Issue at Decode Stage: Solve by Read register content after write operation.**

| | |
|---|---|
| $I_1$ | Move R5, #2000h |
| $I_2$ | Move R1, #15 |
| $I_3$ | Move R2, #16 |
| $I_4$ | Move R3, R1 |
| $I_5$ | Add R1, R3 |
| $I_6$ | Move R4, R2 |
| $I_7$ | Add R2, R4 |
| $I_8$ | Add R1, R2 |
| $I_9$ | Store [R5], R1 |

Image from Wikimedia

# 3: Issues of pipeline and Design Solutions

**Cycle 6: R3 data must be Written from I4 to Register file ; R3 Data must be read by Instruction 5.**

**Data Dependency Issue at Decode Stage: Since Data is not ready, R3 old value is read**

| | |
|---|---|
| $I_1$ | Move R5, #2000h |
| $I_2$ | Move R1, #15 |
| $I_3$ | Move R2, #16 |
| $I_4$ | Move R3, R1 |
| $I_5$ | Add R1, R3 |
| $I_6$ | Move R4, R2 |
| $I_7$ | Add R2, R4 |
| $I_8$ | Add R1, R2 |
| $I_9$ | Store [R5], R1 |

Image from Wikimedia

# 3: Issues of pipeline and Design Solutions

**Cycle 7: R3 data not ready due to instruction dependency**

**Data Dependency Issue at Execution stage : Stall the pipeline**

| | |
|---|---|
| $I_1$ | Move R5, #2000h |
| $I_2$ | Move R1, #15 |
| $I_3$ | Move R2, #16 |
| $I_4$ | Move R3, R1 |
| $I_5$ | Add R1, R3 |
| $I_6$ | Move R4, R2 |
| $I_7$ | Add R2, R4 |
| $I_8$ | Add R1, R2 |
| $I_9$ | Store [R5], R1 |

Image from Wikimedia

# 3: Issues of pipeline and Design Solutions

**Cycle 8: Pipeline stall: no new fetch instruction. Only write back happens**

**Data Dependency Issue at Execution stage : Stall the pipeline**

| | |
|---|---|
| $I_1$ | Move R5, #2000h |
| $I_2$ | Move R1, #15 |
| $I_3$ | Move R2, #16 |
| $I_4$ | Move R3, R1 |
| $I_5$ | Add R1, R3 |
| $I_6$ | Move R4, R2 |
| $I_7$ | Add R2, R4 |
| $I_8$ | Add R1, R2 |
| $I_9$ | Store [R5], R1 |

Image from Wikimedia

# 3: Issues of pipeline and Design Solutions

**Cycle 8: We need data of R3 at execution stage , Bu I4 is performing write back to register now.**
**Use the concept of Data Forwarding**

| | |
|---|---|
| $I_1$ | Move R5, #2000h |
| $I_2$ | Move R1, #15 |
| $I_3$ | Move R2, #16 |
| $I_4$ | Move R3, R1 |
| $I_5$ | Add R1, R3 |
| $I_6$ | Move R4, R2 |
| $I_7$ | Add R2, R4 |
| $I_8$ | Add R1, R2 |
| $I_9$ | Store [R5], R1 |

Image from Wikimedia

# 3: Issues of pipeline and Design Solutions

**Cycle 9: Nothing in Writeback stage**

| | |
|---|---|
| $I_1$ | Move R5, #2000h |
| $I_2$ | Move R1, #15 |
| $I_3$ | Move R2, #16 |
| $I_4$ | Move R3, R1 |
| $I_5$ | Add R1, R3 |
| $I_6$ | Move R4, R2 |
| $I_7$ | Add R2, R4 |
| $I_8$ | Add R1, R2 |
| $I_9$ | Store [R5], R1 |

Image from Wikimedia

# 3: Issues of pipeline and Design Solutions

**Cycle 10: Data dependency at Execution stage; Wait**

| | |
|---|---|
| $I_1$ | Move R5, #2000h |
| $I_2$ | Move R1, #15 |
| $I_3$ | Move R2, #16 |
| $I_4$ | Move R3, R1 |
| $I_5$ | Add R1, R3 |
| $I_6$ | Move R4, R2 |
| $I_7$ | Add R2, R4 |
| $I_8$ | Add R1, R2 |
| $I_9$ | Store [R5], R1 |

Image from Wikimedia

# 3: Issues of pipeline and Design Solutions

**Cycle 11: Data dependency at Execution stage; Data forwarding from writeback stage**

| | |
|---|---|
| $I_1$ | Move R5, #2000h |
| $I_2$ | Move R1, #15 |
| $I_3$ | Move R2, #16 |
| $I_4$ | Move R3, R1 |
| $I_5$ | Add R1, R3 |
| $I_6$ | Move R4, R2 |
| $I_7$ | Add R2, R4 |
| $I_8$ | Add R1, R2 |
| $I_9$ | Store [R5], R1 |

Image from Wikimedia

# 3: Issues of pipeline and Design Solutions

**Cycle 12: Data dependency at Execution stage; Wait**

| | |
|---|---|
| $I_1$ | Move R5, #2000h |
| $I_2$ | Move R1, #15 |
| $I_3$ | Move R2, #16 |
| $I_4$ | Move R3, R1 |
| $I_5$ | Add R1, R3 |
| $I_6$ | Move R4, R2 |
| $I_7$ | Add R2, R4 |
| $I_8$ | Add R1, R2 |
| $I_9$ | Store [R5], R1 |

Image from Wikimedia

# 3: Issues of pipeline and Design Solutions

**Cycle 13: Data dependency at Execution stage; Data Forwarding**

| | |
|---|---|
| $I_1$ | Move R5, #2000h |
| $I_2$ | Move R1, #15 |
| $I_3$ | Move R2, #16 |
| $I_4$ | Move R3, R1 |
| $I_5$ | Add R1, R3 |
| $I_6$ | Move R4, R2 |
| $I_7$ | Add R2, R4 |
| $I_8$ | Add R1, R2 |
| $I_9$ | Store [R5], R1 |

Image from Wikimedia

# 3: Issues of pipeline and Design Solutions

**Cycle 14: Data dependency at Execution stage; Stall Pipeline**

| | |
|---|---|
| $I_1$ | Move R5, #2000h |
| $I_2$ | Move R1, #15 |
| $I_3$ | Move R2, #16 |
| $I_4$ | Move R3, R1 |
| $I_5$ | Add R1, R3 |
| $I_6$ | Move R4, R2 |
| $I_7$ | Add R2, R4 |
| $I_8$ | Add R1, R2 |
| $I_9$ | Store [R5], R1 |

**I9: Store [R5], R1.**

**I8: Add R1, R2**

**STALL**

Image from Wikimedia

# 3: Issues of pipeline and Design Solutions

**Cycle 15: Data Dependency at Memory Write Stage : Use Data Forwarding**

| | |
|---|---|
| $I_1$ | Move R5, #2000h |
| $I_2$ | Move R1, #15 |
| $I_3$ | Move R2, #16 |
| $I_4$ | Move R3, R1 |
| $I_5$ | Add R1, R3 |
| $I_6$ | Move R4, R2 |
| $I_7$ | Add R2, R4 |
| $I_8$ | Add R1, R2 |
| $I_9$ | Store [R5], R1 |

**I9: Store [R5], R1.**

**I8: Add R1, R2**

Image from Wikimedia

# 3: Issues of pipeline and Design Solutions

**Cycle 16: Execution of Program Complete**

**Total Number of cycles taken = 16**

**I9: Store [R5], R1.**

| | |
|---|---|
| $I_1$ | Move R5, #2000h |
| $I_2$ | Move R1, #15 |
| $I_3$ | Move R2, #16 |
| $I_4$ | Move R3, R1 |
| $I_5$ | Add R1, R3 |
| $I_6$ | Move R4, R2 |
| $I_7$ | Add R2, R4 |
| $I_8$ | Add R1, R2 |
| $I_9$ | Store [R5], R1 |

Image from Wikimedia

# 2. Five Stage Pipeline Design

| | **Instruction** | (F) | (D) | (E ) | (M) | (W) | |
|---|---|---|---|---|---|---|---|
| $I_1$ | Move R5, #2000h | $I_1$ | | | | | |
| $I_2$ | Move R1, #15 | $I_2$ | $I_1$ | | | | |
| $I_3$ | Move R2, #16 | $I_3$ | $I_2$ | $I_1$ | | | |
| $I_4$ | Move R3, R1 | $I_4$ | $I_3$ | $I_2$ | $I_1$ | | |
| $I_5$ | Add R1, R3 | $I_5$ | $I_4$ | $I_3$ | $I_2$ | $I_1$ | I1: Complete |
| $I_6$ | Move R4, R2 | $I_6$ | $I_5$ | $I_4$ | $I_3$ | $I_2$ | I2: Complete |
| $I_7$ | Add R2, R4 | $I_7$ | $I_6$ | $I_5$ | $I_4$ | $I_3$ | I3: Complete |
| $I_8$ | Add R1, R2 | $I_8$ | $I_7$ | $I_6$ | $I_5$ | $I_4$ | I4: Complete |
| $I_9$ | Store [R5], R1 | $I_9$ | $I_8$ | $I_7$ | $I_6$ | $I_5$ | I5: Complete |
| $I_{10}$ | | | $I_9$ | $I_8$ | $I_7$ | $I_6$ | I6: Complete |
| $I_{11}$ | | | | $I_9$ | $I_8$ | $I_7$ | I7: Complete |
| $I_{12}$ | | | | | $I_9$ | $I_8$ | I8: Complete |
| $I_{13}$ | | | | | | $I_9$ | I9: Complete |

- If no pipeline, then this program takes

**9 x 5 = 45 cycles**

- If 5 stage pipeline is used, then it takes

**5 + 8 = 13 cycles**

(first instruction takes 'n' stages/cycle) + (n-1) instructions

- Actual Cycles taken due to **data dependency = 16 Cycles.**

# Reference

**Textbooks and/or Reference Books:**

1. Professional CUDA C Programming – John Cheng, Max Grossman, Ty McKercher, 2014

2. Wilkinson, M. Allen, ”Parallel Programming: Techniques and Applications Using Networked Workstations and Parallel Computers”, Pearson Education, 1999

3. I. Foster, ”Designing and building parallel programs”, 2003

4. Parallel Programming in C using OpenMP and MPI – Micheal J Quinn, 2004

5. Introduction to Parallel Programming – Peter S Pacheco, Morgan Kaufmann Publishers, 2011

6. Advanced Computer Architectures: A design approach, Dezso Sima, Terence Fountain, Peter Kacsuk, 2002

7. Parallel Computer Architecture : A hardware/Software Approach, David E Culler, Jaswinder Pal Singh Anoop Gupta, 2011 8. Introduction to Parallel Computing, Ananth Grama, Anshul Gupta, George Karypis, Vipin Kumar, Pearson, 2011

# Thank You

**National Institute of Technology Karnataka Surathkal**
**Department of Information Technology**

# IT 301 Parallel Computing

## Scalar Processors – Control Instructions

**Dr. Geetha V**

*Assistant Professor*

*Dept of Information Technology*

*NITK Surathkal*

# Index

- 1. Five Stage pipeline design
- 2: Control Hazards

# 1: Introduction

**Course Plan: Theory:**

**Part A: Parallel Computer Architectures**

Week 1,2,3: ***Introduction to Parallel Computer Architecture:***

Parallel Computing,

Parallel architecture,

bit level, instruction level , data level and task level parallelism.

**Instruction level parallelisms: pipelining (Data and control instructions),**

**scalar processors** and superscalar processors,

vector processors.

Parallel computers and computation.

# 1. Five Stage Pipeline Design

- Fetch Stage: The address in program counter is sent to Instruction register and instruction is fetched from memory. PC is incremented.
- Decode Stage: The operands are fetched from register file; Sign extension done if necessary.
- Execution Stage: The instruction is executed.
- Memory Stage: The Load and Store instructions execution.
- Write Back Stage: The results are written back to register file.

| | | | | |
|---|---|---|---|---|
| **Fetch (F)** | **Decode (D)** | **Execute (E)** | **Memory Access (M)** | **Write Back (W)** |

# 1. Five Stage Pipeline Design

Image from Wikimedia

# 1: Five stage pipeline design

- **Data Dependency (Data Hazard)**

Dependency of instructions due to operand values unavailability

Solution :

(a) Data forwarding

(b) Stall the pipeline untill the data is available.

- **Control Dependency (Control Hazard)**

Dependency due to unresolved decision on control instructions.

# 2. Control hazards

- The branches – conditional branches – cause pipeline hazard
- The outcome of a conditional branch is not known until the end of the EX stage, but is required at IF to load another instruction and keep the pipeline full.

```
        JGE Next
        Add CX, 02
        Dec BL
        JMP NEXT
NEXT:
```

| | | | | |
|---|---|---|---|---|
| **Fetch (F)** | **Decode (D)** | **Execute (E)** | **Memory Access (M)** | **Write Back (W)** |

# 2. Control Hazards

- **Data Dependency (Data Hazard)**

Dependency of instructions due to operand values unavailability

Solution :

(a) Data forwarding

(b) Stall the pipeline untill the data is available.

- **Control Dependency (Control Hazard)**

Dependency due to unresolved decision on control instructions.

(a) Insert NOP (No Operation)

(b) NOP after execution of instruction

(c) Branch Prediction

# 2. Control hazards

| Memory | Instruction | Operation |
|---|---|---|
| 400 | MOV SI, 500 | SI <- 500 |
| 403 | MOV DI, 600 | DI <- 600 |
| 406 | MOV AX, 0000 | AX = 0000 |
| 409 | MOV CL, [SI] | CL <- [SI] |
| 40B | MOV BL, CL | BL <- CL |
| 40D | INC SI | SI = SI + 1 |
| 40E | NXT: ADD AL, [SI] | AL = AL + [SI] |
| 410 | ADC AH, 00 | AH = AH + 00 + cy |
| 412 | INC SI | SI = SI + 1 |
| 413 | DEC CL | CL = CL – 1 |
| 415 | JNZ NXT (40E) | JUMP if ZF = 0 |
| 417 | DIV BL | AX = AX / BL |
| 419 | MOV [DI], AX | [DI] <- AX |
| 41B | HLT | Stop |

- Program to find average of array elements

(a) Inserting NOP

1. Insert NOP (No operation) when Branch instruction

| F | D | E | M | W |
|---|---|---|---|---|
| NOP | NOP | JNZ 40E | DEC CL | INC SI |

| F | D | E | M | W |
|---|---|---|---|---|
| ADD AL, [SI] | NOP | NOP | JNZ 40E | DEC CL |

# 2. Control hazards

| Memory | Instruction | Operation |
|---|---|---|
| 400 | MOV SI, 500 | SI <- 500 |
| 403 | MOV DI, 600 | DI <- 600 |
| 406 | MOV AX, 0000 | AX = 0000 |
| 409 | MOV CL, [SI] | CL <- [SI] |
| 40B | MOV BL, CL | BL <- CL |
| 40D | INC SI | SI = SI + 1 |
| 40E | NXT: ADD AL, [SI] | AL = AL + [SI] |
| 410 | ADC AH, 00 | AH = AH + 00 + cy |
| 412 | INC SI | SI = SI + 1 |
| 413 | DEC CL | CL = CL – 1 |
| 415 | JNZ NXT (40E) | JUMP if ZF = 0 |
| 417 | DIV BL | AX = AX / BL |
| 419 | MOV [DI], AX | [DI] <- AX |
| 41B | HLT | Stop |

- Program to find average of array elements

(b) NOP after execution of instruction

2.Insert NOP (No operation) when Branch is taken (after execution of branch)

| F | D | E | M | W |
|---|---|---|---|---|
| MOV [DI], AX | DIV BL | JNZ 40E | DEC CL | INC SI |

| F | D | E | M | W |
|---|---|---|---|---|
| ~~MOV [DI], AX~~ NOP | ~~DIV BL~~ NOP | JNZ 40E | DEC CL | INC SI |

| F | D | E | M | W |
|---|---|---|---|---|
| ADD AL, [SI] | ~~MOV [DI], AX~~ NOP | ~~DIV BL~~ NOP | JNZ 40E | DEC CL |

# 2. Control hazards

| Memory | Instruction | Operation |
|---|---|---|
| 400 | MOV SI, 500 | SI <- 500 |
| 403 | MOV DI, 600 | DI <- 600 |
| 406 | MOV AX, 0000 | AX = 0000 |
| 409 | MOV CL, [SI] | CL <- [SI] |
| 40B | MOV BL, CL | BL <- CL |
| 40D | INC SI | SI = SI + 1 |
| 40E | NXT: ADD AL, [SI] | AL = AL + [SI] |
| 410 | ADC AH, 00 | AH = AH + 00 + cy |
| 412 | INC SI | SI = SI + 1 |
| 413 | DEC CL | CL = CL – 1 |
| 415 | JNZ NXT (40E) | JUMP if ZF = 0 |
| 417 | DIV BL | AX = AX / BL |
| 419 | MOV [DI], AX | [DI] <- AX |
| 41B | HLT | Stop |

- Program to find average of array elements

(c) Branch Prediction : Branch Taken

3. Branch Prediction: Branch Taken

| F | D | E | M | W |
|---|---|---|---|---|
| ADC AH, 00 | ADD AL, [SI] | JNZ 40E | DEC CL | INC SI |

Branch not taken

| F | D | E | M | W |
|---|---|---|---|---|
| ~~ADC AH, 00~~ NOP | ~~ADD AL, [SI]~~ NOP | JNZ 40E | DEC CL | INC SI |

| F | D | E | M | W |
|---|---|---|---|---|
| DIV BL | ~~ADC AH, 00~~ NOP | ~~ADD AL, [SI]~~ NOP | JNZ 40E | DEC CL |

Branch Taken

| F | D | E | M | W |
|---|---|---|---|---|
| INC SI | ADC AH, 00 | ADD AL, [SI] | JNZ 40E | DEC CL |

# Thank You

# Reference

**Textbooks and/or Reference Books:**

1. Professional CUDA C Programming – John Cheng, Max Grossman, Ty McKercher, 2014

2. Wilkinson, M. Allen, ”Parallel Programming: Techniques and Applications Using Networked Workstations and Parallel Computers”, Pearson Education, 1999

3. I. Foster, ”Designing and building parallel programs”, 2003

4. Parallel Programming in C using OpenMP and MPI – Micheal J Quinn, 2004

5. Introduction to Parallel Programming – Peter S Pacheco, Morgan Kaufmann Publishers, 2011

6. Advanced Computer Architectures: A design approach, Dezso Sima, Terence Fountain, Peter Kacsuk, 2002

7. Parallel Computer Architecture : A hardware/Software Approach, David E Culler, Jaswinder Pal Singh Anoop Gupta, 2011 8. Introduction to Parallel Computing, Ananth Grama, Anshul Gupta, George Karypis, Vipin Kumar, Pearson, 2011

**National Institute of Technology Karnataka Surathkal**
**Department of Information Technology**

# IT 301 Parallel Computing

## Superscalar Processors 1

**Dr. Geetha V**

*Assistant Professor*

*Dept of Information Technology*

*NITK Surathkal*

# Index

- 1. Instruction Level Parallelism
- 2: Data Hazards and Instruction Issue
- 3. Inorder Vs Out of order Execution

# 1: Introduction

**Course Plan: Theory:**

**Part A: Parallel Computer Architectures**

Week 1,2,3: ***Introduction to Parallel Computer Architecture:***

Parallel Computing,

Parallel architecture,

bit level, instruction level , data level and task level parallelism.

Instruction level parallelisms: pipelining (Data and control instructions),

scalar processors and **superscalar processors**,

vector processors.

Parallel computers and computation.

# 1. Instruction level parallelism

## Scheduling and Preserving Sequential Consistency

**Early RISC Processors Without an FP unit**

**RISC I, II (1983)**
**MIPS (1981)**
**IBM 801 (1978)**

**Scalar Processors With an FP unit**

**Pentium (1993)**
**I486 (1988)**
**MIPS R 2/3000 (1988)**

**Supercalar Processors**

**Pentium Pro(1993)**
**PowerPC 603 (1993)**

# 1. Instruction level parallelism

## Case Study : Logical Layout of Pentium's Pipeline

# 1. Instruction level parallelism

## Superscalar vs VLIW

Scalar Pipeline Execution

A Superscalar Processor of Degree 3

VLIW Execution with Degree 3

# 1. Instruction level parallelism

## ILP in Superscalar processor

# 2: Data Hazard and Instruction Issue

## The issue bottleneck

(a): Simplified structure of the mikroarchitecture assuming unbuffered issue

(b): The issue process

# 2: Data Hazard and Instruction Issue

## The issue bottleneck

(a): Simplified structure of the mikroarchitecture assuming unbuffered issue

## Dynamic instruction issue

(a): Simplified structure of the mikroarchitecture assuming buffered issue (shelving)

# 2: Data Hazard and Instruction Issue

- **Data Dependency (Data Hazard)**

Dependency of instructions due to operand values unavailability

- **Control Dependency (Control Hazard)**

Dependency due to unresolved decision on control instructions.

# 2: Data Hazard and Instruction Issue

- **Data Dependency (Data Hazard)**

Dependency of instructions due to operand values unavailability

- Read After Read (RAR) : No dependency
- Read After Write (RAW) : Common Hazard
- Write After Read (WAR) : Hazard in Out of order execution
- Write After Write (WAW) : Hazard in Out of order execution

# 2: Data Hazard and Instruction Issue

- **Data Dependency (Data Hazard)**

Dependency of instructions due to operand values unavailability

- **Read After Read (RAR) :** No dependency

ADD R1, **R2**, R6 ; R1 <- **R2**+R6

SUB R4, **R2**, R7; R4 <- **R2**+R7

# 2: Data Hazard and Instruction Issue

- **Data Dependency (Data Hazard)**

Dependency of instructions due to operand values unavailability

- **Read After Write (RAW)** : Common Hazard

ADD **R1**, R2, R6 ; R1 <- R2+R6

SUB R4, **R1**, R7; R4 <- R1+R7

Solve Using Data Forwarding

# 2: Data Hazard and Instruction Issue

- **Data Dependency (Data Hazard)**

Dependency of instructions due to operand values unavailability

- **Write After Read (WAR) :** Not Common in in-order pipeline

ADD R1, **R2**, R3 ;  R1 <- R2+R3

SUB **R2**, R4, R5;  R2 <- R4+R5

For Out of Order execution, Use dependency check for execution

# 2: Data Hazard and Instruction Issue

- **Data Dependency (Data Hazard)**

Dependency of instructions due to operand values unavailability

- **Write After Write (WAW) :** Not hazard in in-order pipeline

ADD **R1**, R2, R3 ;  R1 <- R2+R3

SUB **R1**, R4, R5;  R1 <- R4+R5

# 2: Data Hazard and Instruction Issue

## The issue bottleneck

(a): Simplified structure of the mikroarchitecture assuming unbuffered issue

## Dynamic instruction issue

(a): Simplified structure of the mikroarchitecture assuming buffered issue (shelving)

# 3. Inorder Vs Out of order execution

- **In-order instruction execution**
  - Instructions are fetched, executed and completed in compiler generated order
  - One instruction stall, stalls all other instructions
  - Instructions are Statistically Scheduled

- **Out-of-order instruction execution**
  - Instructions are fetched in compiler-generated order
  - Instruction completion may be in-order or out-of-order
  - Independent instruction behind a stalled instruction can pass it
  - Instructions are Dynamically executed

https://courses.cs.washington.edu/courses/csep548/06au/lectures/introOOO.pdf

# 3. Inorder Vs Out of order execution

- **Out-of-order procesors**
  - After instruction decode
    - Check for structural hazards
      - An instruction can be issued when a functional unit is available
      - An instruction stalls if no appropriate functional unit
    - Check for Data Hazards
      - An instruction can execute when its operands have been calculated or loaded from memory
      - An instruction stalls if operands are not available.

# 3. Inorder Vs Out of order execution

- **Out-of-order procesors**
  - After instruction decode
    - Independent ready instructions can execute before earlier instructions that are stalled

***in-order processors***

| | |
|---|---|
| `lw $3, 100($4)` | in execution, cache miss |
| `add $2, $3, $4` | waits until the miss is satisfied |
| `sub $5, $6, $7` | waits for the add |

***out-of-order processors***

| | |
|---|---|
| `lw $3, 100($4)` | in execution, cache miss |
| `sub $5, $6, $7` | can execute during the cache miss |
| `add $2, $3, $4` | waits until the miss is satisfied |

# 3. Inorder Vs Out of order execution

- **Out-of-order procesors**
  - After instruction decode
    - Independent ready instructions can execute before earlier instructions that are stalled
      - When path instructions are waiting for a branch condition to be computed
        - The instructions that are issued from the predicted path are issued speculatively, called speculative execution.
        - Speculative isntructions are execute (but not commit) before the branch is resolved
        - If the prediction was wrong, speculative instructions are flushed from the pipeline
        - If prediction is right, instructions are no longer speculative.

# 3. Inorder Vs Out of order execution

- **Speculative Execution**
  - Executing an instruction before it is known that it should be executed
    - All instructions that are fetched because of a prediction are speculative
    - Inorder pipeline : branch is executed before the path
    - Out of order pipeline
      - Path can be executred before the branch
      - Speculative instructions can execute but not committed
      - Getting rid of wrong path instructions is not just a matter of flushing them from the pipeline.

# Thank You

# Reference

**Textbooks and/or Reference Books:**

1. Professional CUDA C Programming – John Cheng, Max Grossman, Ty McKercher, 2014

2. Wilkinson, M. Allen, ”Parallel Programming: Techniques and Applications Using Networked Workstations and Parallel Computers”, Pearson Education, 1999

3. I. Foster, ”Designing and building parallel programs”, 2003

4. Parallel Programming in C using OpenMP and MPI – Micheal J Quinn, 2004

5. Introduction to Parallel Programming – Peter S Pacheco, Morgan Kaufmann Publishers, 2011

6. Advanced Computer Architectures: A design approach, Dezso Sima, Terence Fountain, Peter Kacsuk, 2002

7. Parallel Computer Architecture : A hardware/Software Approach, David E Culler, Jaswinder Pal Singh Anoop Gupta, 2011 8. Introduction to Parallel Computing, Ananth Grama, Anshul Gupta, George Karypis, Vipin Kumar, Pearson, 2011

**National Institute of Technology Karnataka Surathkal**
**Department of Information Technology**

# IT 301 Parallel Computing
# Superscalar Processor 2

**Dr. Geetha V**

*Assistant Professor*

*Dept of Information Technology*

*NITK Surathkal*

Acknowledgment to slides :
www2.latech.edu/~choi/Bens/Teaching/Csc521/

# Superscalar Processors

- 7.1 Introduction
- 7.2 Parallel decoding
- 7.3 Superscalar instruction issue
- 7.4 Shelving -Example
- 7.5 Register renaming
- 7.6 Parallel execution
- 7.7 Preserving the sequential consistency of instruction execution
- 7.8 Preserving the sequential consistency of exception processing
- 7.9 Implementation of superscalar CISC processors using a superscalar RISC core
- 7.10 Case studies of superscalar processors

TECH

Computer Science

# 7.1 Superscalar Processors vs. VLIW

- **VLIW architecture** : (Very large Instruction Word)
- One single word contains many instructions, without any data dependency
- **Superscalar approach** uses dynamic instruction execution with dynamic instruction scheduling
- Multiple instructions are fetched and dependencies are verified dynamically before issuing instruction for execution.

# 7.1 Superscalar Processor: Intro

- Parallel Issue
- Parallel Execution
- {Hardware} Dynamic Instruction Scheduling
- Currently the predominant class of processors
  - **Pentium**
  - **PowerPC**
  - **UltraSparc**
  - **AMD K5-**
  - **HP PA7100-**
  - **DEC α**

# 7.1 Specific tasks of superscalar processing

# 7.2 Parallel decoding {and Dependencies check}

- In scalar processor the decode and instruction issue takes place in same stage.
- In superscalar processor, more than one instruction will be fetched and kept in Instruction Buffer for dependency check.
- More than one instruction is decoded and issued in superscalar processor at the same time.

# 7.2 Decoding and Pre-decoding

- Superscalar processors tend to use 2 and sometimes even 3 or more pipeline cycles for decoding and issuing instructions
- >> Pre-decoding:
  - **shifts a part of the decode task up into loading phase**
  - **resulting of pre-decoding**
    - the instruction class (data transmission, logical, arithmetic, control, memory operations etc)
    - the type of resources required for the execution (FX /FP load/store etc)
    - in some processor (e.g. UltraSparc), branch target addresses calculation as well (advanced branch address calculation for branch taken and not taken path)
  - **the results are stored by attaching 4-7 bits**
- + shortens the overall cycle time or reduces the number of cycles needed

# 7.2 The principle of perdecoding

- In precode stage identify class of instruction, calculate branch target address, resources required for execution etc.
- 4/5/7 bits are attached in precode stage

When instructions are written into the I-cache, the predecode unit appends 4–7 bits to each RISC instruction [1]

[1] In the AMD K5, which is an x86-compatible CISC processor, the predecode unit appends 5 bits to each byte

# 7.3 Specific tasks of superscalar processing: Issue

# 7.3 Superscalar instruction issue

- How and when to send the instruction(s) to EU(s)

# 7.3.1 Issue policies

# 7.3.1 Instruction issue policies of superscalar processors: ---Performance, tread-----→

# 7.3.2 Issue rate {How many instructions/cycle}

- CISC about 2
- RISC:

# 7.3.1 Issue policies: Handing Issue Blockages

Resolving RAW and WAR hazards using register renaming

Speculating as branch taken or not taken based on history bits

# 7.3.1.1 Issue stopped by True dependency

- True dependency → (Blocked: need to wait)

# 7.3.1.1. Issue order of instructions

# 7.3.1.1 Aligned vs. unaligned issue

# 7.3.1 Issue policies: Handing Issue Blockages

# 7.3.1.2 Blocking Issue (Direct Issue)

- The instructions are checked for dependency in decode stage.
- If there is any dependency the instruction is blocked. If there is no dependency the instruction is directly issued.

# 7.3.1.2 The principle of shelving: Indirect Issue

- Shelving is nothing but buffering the instructions
- The instructions are checked for dependency in the buffer called Reservation Station.
- If there is any dependency the instruction is blocked. If there is no dependency the instruction is directly issued.

# 7.3.1.2 Design space of shelving

**a)Partial shelving:** Shelving is restricted to one or to a few instruction types (Power1, Power2 – only FP instructions)

**b)Full Shelving: Shelving** covers all instruction types (PowerPC 620)

**a) Type of the shelving buffer:** S:22

- standalone shelving Buffer (Reservation station)
- Combined buffer for shelving, renaming and reordering (DRIS: Deferred scheduling, register renaming, Instruction shelve) S:23

**b) Number of Shelving buffers**
PowerPC 620 (15 number)
PentiumPro (20 number)

**c) Number of read and write ports**
how many instructions may be written into (input ports) or read out from (output parts) a particular shelving buffer in a cycle
- depend on individual, group, or central reservation stations

**a)Issue Bound Fetch :** Operands are fetched during instruction issue. Then shelving buffer hold source operand value S:24
Eg: PowerPC 620
PentiumPro

**b) Dispatch bound fetch** Operands are fetched during Instruction dispatch. Then shelving buffers hold source register numbers S:25
Eg: Power1, Power2

**a)Dispatch policy** S:26 -
selection rule
-arbitrary rule
-dispatch order S:27

**b) Dispatch rate**
S:28 S:29

**c)Scheme for checking the availability of operands.**
Score boarding
S: 30 S:31 S:32 S:33 S:34

**d) Treatment of an empty reservation station**
--Straightforward approach
--Bypassing
S:35

# 7.3.1.2 Basic variants of shelving buffers

*Power1 (1990)*
*Nx586 (1994)*
*PowerPC 603 (1993)*
*PowerPC 604 (1995)*
*PowerPC 620 (1996)*
*Am29000 sup. (1995)*
*K5 (1995)*

*ES/9000 (1992p)*
*Power2 (1993)*
*R10000 (1996)*
*PM1 (Sparc64) (1995)*

*PentiumPro (1995)*

# 7.3.1.2 Using a combined buffer for shelving, renaming, and reordering

- **The buffer is used for**
  -shelve the instructions
  -Register renaming
  -Reordering Instruction

- **Shelving:** Buffer for saving instructions

- **Reordering** : The execution of instruction may be out-of-order. But final writeback must be in order.
- Here i1 and i3 may complete out of order. But writeback of i3 happens only after completion of i2

**Reorder Buffer**

- **Register Renaming:**

i1: Add **R1,** R2, R3
i2: Store **R1**, Loc1
i3: Add **R1,** R4, R5
i4: Store **R1,** Loc2
Rename as follows
i1: Add **R11,** R2, R3
i2: Store **R11**, Loc1
i3: Add **R12,** R4, R5
i4: Store **R12,** Loc2

# 7.3.1.2 Operand fetch during instruction issue

- Operands are fetched from register file during instruction issue

# 7.3.1.2 Operand fetch during instruction dispatch

- Operands are fetched from register file during instruction dispatch to Execution unit

# 7.3.1.2- Dispatch policy

- Selection Rule
  - **Specifies when instructions are considered executable**
  - **e.g. Dataflow principle of operation**
    - Those instructions whose **operands** are available are executable.
- Arbitration Rule
  - **Needed when more instructions are eligible for execution than can be disseminated.**
  - **e.g. choose the 'oldest' instruction.**
- Dispatch order
  - **Determines whether a non-executable instruction prevents all subsequent instructions from being dispatched.**

# 7.3.1.2 Dispatch policy: Dispatch order

# 7.3.1.2 -Dispatch rate (instructions/cycle)

# Maximum issue rate <= Maximum dispatch rates
# >> issue rate reaches max more often than dispatch rates

**Table 7.3** Maximum issue and dispatch rates of superscalar processors with shelving.

| *Processor/Year of volume shipment* | *Maximum issue rate instr./cycle* | *Maximum dispatch rate*[1] *instr./cycle* |
|---|---|---|
| PowerPC 603 (1993) | 3 | 3 |
| PowerPC 604 (1995) | 4 | 6 |
| PowerPC 620 (1996) | 4 | 6 |
| Power2 (1993) | 4/6[2] | 10 |
| Nx586 (1994) | 3/4[3,4] | 3/4[3,4] |
| K5 (1995) | 4[4] | 5[4] |
| PentiumPro (1995) | 4 | 5[4] |
| PM1 (Sparc 64) (1995) | 4 | 8 |
| PA8000 (1996) | 4 | 4 |
| R10000 (1996) | 4 | 5 |

[1] Because of address calculations performed separately, the given numbers are usually to be interpreted as operations/cycle. For instance, the Power2 performs maximum 10 operations/cycle, which corresponds to 8 instructions/cycle.

[2] The issue rate is 6 for sequential mode and 4 for target mode.

[3] Both rates are 3 without an optional FP unit (labelled Nx587) and 4 with it.

[4] Both rates are related to RISC operations (rather than to the native CISC operations) performed by the superscalar RISC core.

## 7.3.1.2 - Scheme for checking the availability of operands: The principle of scoreboarding

Interpretation:

0 ← When an instruction is issued the scoreboard entry corresponding to the destination register is reset to '0'

1 ← When the execution of an instruction is completed, and the result is written into the destination register, the corresponding scoreboard entry is set to '1'

# 7.3.1.2 Schemes for checking the availability of operand

**Schemes for checking the availability of operands**

**Direct check of the scoreboard bits**

The availability of source operands is not explicitly indicated in the RS. Thus, the scoreboard bits are tested for availability

*Usually employed if operands are fetched during instruction dispatch, as assumed below*

**Check of the explicit status bits**

The availability of source operands is explicitly indicated in the RS. These explicit status bits are tested for availability

*Usually employed if operands are fetched during instruction issue, as assumed below*

# 7.3.1.2 Operands fetched during dispatch or during issue

| | |
|---|---|
| OC: | Operation code |
| $R_{SI}$, $R_{S2}$: | Source register numbers |
| $R_D$: | Destination register number |
| $O_{SI}$, $O_{S2}$: | Source operand values |
| $I_{SI}$, $I_{S2}$: | Source operand identifiers (tags) |
| $V_{SI}$, $V_{S2}$: | Source operand valid bits |
| RS: | Reservation station |

# 7.3.1.2 Use of multiple buses for updaing multiple individual reservation strations

# 7.3.1.2 Interal data paths of the powerpc 604

# 7.3.1.2 -Treatment of an empty reservation station

# 7.4 Detail Example of Shelving

- Issuing the following instruction
  - **cycle i: mul r1, r2, r3**
  - **cycle i+1: ad r2, r3, r5**
  - **ad r3, r4, r6**

    - format: Rs1, Rs2, Rd

mul r1, r2, r3
Ad r2, r3, r5
Ad r3, r4, r6

# Example overview

# Cycle i: Issue of the 'mul' instruction into the reservation station and fetching of the corresponding operands

mul r1, r2, r3
Ad r2, r3, r5
Ad r3, r4, r6

# Cycle i+1: Checking for executable instructions and dispatching of the 'mul' instruction

mul r1, r2, r3
Ad r2, r3, r5
Ad r3, r4, r6

# Cycle i+1 (2nd phase): Issue of the subsequent two 'ad' instructions into the reservation station

mul r1, r2, r3
Ad r2, r3, r5
Ad r3, r4, r6

# Cycle i+2: Checking for executable instruction (mul not yet completed)

mul r1, r2, r3
Ad r2, r3, r5
Ad r3, r4, r6

# Cycle i+3: Updating the FX register file with the result of the ‘mul’ instruction

mul r1, r2, r3
Ad r2, r3, r5
Ad r3, r4, r6

# Cycle i+3 (2nd phase): Checking for executable instructions and dispatching the 'older' 'ad' instruction

mul r1, r2, r3
Ad r2, r3, r5
Ad r3, r4, r6

# 7.3.1 Issue policies: Handing Issue Blockages

Resolving WAW hazards using register renaming

Speculating as branch taken or not taken based on history bits

# Register Remaining and dependency

- three-operand instruction format
- e.g. Rd, Rs1, Rs2
- False dependency (WAW)
  - **mul r2, …, …**
  - **add r2, …, …**
    - two different rename buffer have to allocated
- True data dependency (RAW)
  - **mul r2, …, …**
  - **ad …, r2, …**
    - rename to e.g.
    - mul p12, …, …
    - ad …, p12, ….

# Static or Dynamic Renaming

**Implementation of register renaming**

**Static Implementation**

Performed during compilation, i.e. statically, in parallel optimizing compilers

**Dynamic Implementation**

Performed during execution, i.e. dynamically, in superscalar processors

# >Design space of register renaming

# -Scope of register renaming

**Scope of register renaming**

**Partial renaming**

Renaming is restricted to particular instruction types

*Power1[1] (RS/6000, 1993)*
*Power 2[2] (1993)*
*PowerPC 601[3] (1993)*
*Nx586[4] (1994)*

**Full renaming**

Renaming comprises all eligible instruction types

*PowerPC 603 (1993)*
*PowerPC 604 (1995)*
*PowerPC 620 (1996)*
*R10000 (1996)*
*and*
*most recent superscalar processors except the α-line and Sun's UltraSparc (1995)*

Trend

[1]The Power1 renames only FP loads
[2]The Power2 extends renaming to all FP instructions
[3]The PowerPC 601 renames only the Link and Count registers
[4]Since the Nx586 is an FX processor, it renames only FX instructions

# -Layout of rename buffers

# -Type of rename buffers

# Rename buffers hold intermediate results

- Each time a Destination register is referred to, a new rename register is allocated to it.
- Final results are stored in the Architectural Register file
- Access both rename buffer and architectural register file to find the latest data,
  - **if found in both, the data content in rename buffer (the intermediate result) is chosen.**
- When an instruction completed (retired),
  - **(ROB) {retire only in strict program sequence}**
  - **the correspond rename buffer entry is writing into the architectural register file (as a result modifying the actual program state)**
  - **the correspond rename buffer entry can be de-allocated**

# -Number of rename buffers

| *Implementation of renaming* | | *Number of rename buffers* | |
|---|---|---|---|
| Processor type | | FX | FP |
| **Merged rename and arch. register file** | | | |
| Power1 | (1990) | – | 8 (32 arch. + 8 rename) |
| Power2 | (1993) | – | 22 (32 arch. + 22 rename) |
| ES/9000 | (1992p) | 16 (16 arch. + 16 ren.) | 12 (4 arch. + 12 rename) |
| PM1 | (1995) | 38 (78 arch. + 38 ren.) | 24 (32 arch. + 24 rename) |
| R10000 | (1996) | 32 (32 arch. + 32 ren.) | 32 (32 arch. + 32 rename) |
| **Separate rename register file** | | | |
| PowerPC 603 | (1993) | n.a. | 4 |
| PowerPC 604 | (1995) | 12 | 8 |
| PowerPC 620 | (1996) | 8 | 8 |
| **Renaming within the ROB** | | | |
| Am29000 sup | (1995) | 10 | |
| K5 | (1995) | 16 | |
| PentiumPro | (1995) | 40 | |

# -Basic mechanisms used for accessing rename buffers

- **Rename buffers with associative access (latter e.g.)**
- **Rename buffers with indexed access**
  - (always corresponds to the most recent instance of renaming)

# -Operand fetch policies and Rename Rate

- rename bound: fetch operands during renaming (during instruction issue)
- dispatch bound: fetch operand during dispatching

- Rename Rate
  - **the maximum number of renames per cycle**
  - **equals the issue rate: to avoid bottlenecks.**

# 7.5.8 Detailed example of renaming

- renaming:
  - **mul r2, r0, r1**
  - **ad  r3, r1, r2**
  - **sub r2, r0, r1**

- format:
  - **op  Rd, Rs1, Rs2**

- Assume:
  - **separate rename register file,**
  - **associative access, and**
  - **operand fetching during renaming**

# Structure of the rename buffers and their supposed initial contents

- Latest bit: the most recent rename 1, previous 0

| | Entry valid | Dest reg.no. | Value | Value valid | Latest bit |
|---|---|---|---|---|---|
| 0 | 1 | 4 | 40 | 1 | 1 |
| 1 | 1 | 0 | 0 | 1 | 1 |
| 2 | 1 | 1 | 10 | 1 | 1 |
| 3 | 0 | | | | |
| 4 | 0 | | | | |
| ⋮ | ⋮ | | Rename registers | | |
| | | | | | |

# Renaming steps

- Allocation of a free rename register to a destination register
- Accessing valid source register value or a register value that is not yet available
- Re-allocation of destination register
- Updating a particular rename buffer with a computed result
- De-allocation of a rename buffer that is no longer needed.

# Allocation of a new rename buffer to destination register (circular buffer: Head and Tail) (before allocation)

mul r2, r0, r1:

| | | Entry valid | Dest reg.no. | Value | Value valid | Latest bit |
|---|---|---|---|---|---|---|
| | 0 | 1 | 4 | 40 | 1 | 1 |
| | 1 | 1 | 0 | 0 | 1 | 1 |
| | 2 | 1 | 1 | 10 | 1 | 1 |
| Head → | 3 | 0 | | | | |
| | 4 | 0 | | | | |
| | I | I | | | | |
| | I | I | | Rename registers | | |
| | I | I | | | | |
| (i) Tail → | | 0 | | | | |

# (After allocation) of a destination register

| | Entry valid | Dest reg.no. | Value | Value valid | Latest bit |
|---|---|---|---|---|---|
| 0 | 1 | 4 | 40 | 1 | 1 |
| 1 | 1 | 0 | 0 | 1 | 1 |
| 2 | 1 | 1 | 10 | 1 | 1 |
| 3 | 1 | 2 | | 0 | 1 |
| Head → 4 | 0 | | | | |
| \| | \| | | Rename registers | | |
| \| | \| | | | | |
| \| | \| | | | | |
| (ii) Tail → | 0 | | | | |

# Accessing abailable register values

# Accessing a register value that is not yet available

# Re-allocate of r2 (a destination register)

sub r2, r0, r1

| | Entry valid | Dest reg.no. | Value | Value valid | Latest bit |
|---|---|---|---|---|---|
| 0 | 1 | 4 | 40 | 1 | 1 |
| 1 | 1 | 0 | 0 | 1 | 1 |
| 2 | 1 | 1 | 10 | 1 | 0 |
| 3 | 1 | 2 | | 0 | 0 |
| 4 | 0 1 | 3 | | 0 | 1 |
| 5 | 1 | 2 | | 0 | 1 |
| 6 | 0 | | | | |
| I I I | I I I | | Rename registers | | |
| | 0 | | | | |

# Updating the rename buffers with computed result of {mul r2, r0, r1} (register 2 with the result 0)

Result 0 entry no 3 →

| | Entry valid | Dest reg.no. | Value | Value valid | Latest bit |
|---|---|---|---|---|---|
| 0 | 1 | 4 | 40 | 1 | 1 |
| 1 | 1 | 0 | 0 | 1 | 1 |
| 2 | 1 | 1 | 10 | 1 | 1 |
| 3 | 1 | 2 | 0 | 1 | 0 |
| 4 | 0 1 | 3 | | 0 | 1 |
| 5 | 1 | 2 | | 0 | 1 |
| 6 | 0 | | | | |
| I<br>I<br>I | I<br>I<br>I | | Rename registers | | |
| | 0 | | | | |

# Deallocation of the rename buffer no. 0 (ROB retires instructions) (update tail pointer)

Deallocation of entry no. 0

| | Entry valid | Dest reg.no. | Value | Value valid | Latest bit |
|---|---|---|---|---|---|
| 0 | 0 | 4 | 40 | 1 | 1 |
| 1 | 1 | 0 | 0 | 1 | 1 |
| 2 | 1 | 1 | 10 | 1 | 1 |
| 3 | 0 | | | 0 | 0 |
| 4 | 0 | 3 | | 0 | 1 |
| 5 | 1 | 2 | | 0 | 1 |
| 6 | 0 | | | | |
| ⋮ | ⋮ | | Rename registers | | |
| | 0 | | | | |

# 7.1 Specific tasks of superscalar processing

# 7.6 Parallel Execution

- Executing several instruction in parallel
  - **instructions will generally be finished in out-of-program order**
- to finish
  - **operation of the instruction is accomplished,**
  - **except for writing back the result into**
    - the architectural register or
    - memory location specified, and/or
    - updating the status bits
- to complete
  - **writing back the results**
- to retire (ROB)
  - **write back the results, and**
  - **delete the completed instruction from the last ROB entry**

# 7.1 Specific tasks of superscalar processing

# 7.7 Preserving Sequential Consistency of instruction execution //

- Multiple EUs operating in parallel, the overall instruction execution should
  >> mimic sequential execution
  - **the order in which instruction are completed**
  - **the order in which memory is accessed**

# Sequential consistency models

# Consistency relate to instruction completions or memory access

# Trend and performance

| Detection and resolution of dependencies ensures weak processor consistency | ROB ensures strong processor consistency | Detection and resolution of memory data dependencies ensures weak memory consistency | The ROB may be used to ensure strong memory consistency |
|---|---|---|---|
| *Power1 (1990)* | *ES/9000 (1992p)* | *MC88110 (1993)* | *ES/9000 (1992p)* |
| *Power2 (1993)* | *PowerPC 602–620* | *PowerPC 602–620* | *PowerPC 601 (1993)* |
| *MC88110 (1993)* | *PentiumPro (1995)* | *UltraSparc (1995)* | |
| *PowerPC 601 (1993)* | *UltraSparc (1995)* | *PM1 (1995)* | |
| *α– line up to α 21164* | *PM1 (1995)* | *PA 8000 (1996)* | |
| *R8000 (1994)* | *Am29000 sup (1995)* | *R10000 (1996)* | |
| | *K5 (1995)* | | |
| | *PA 8000 (1996)* | | |
| | *R10000 (1996)* | | |

Trend

Trend, performance

# Allows the reordering of memory access

- it permits load/store reordering
  - **either loads can be performed before pending stores, or vice versa**
  - **a load can be performed before pending stores only IF**
  - **none of the preceding stores has the same target address as the load**
- it makes Speculative loads or stores feasible
  - **When addresses of pending stores are not yet available,**
  - **speculative loads avoid delaying memory accesses, perform the load anywhere.**
  - **When store addresses have been computed, they are compared against the addresses of all younger loads.**
  - **Re-load is needed if any hit is found.**
- it allows cache misses to be hidden
  - **if a cache miss, it allows loads to be performed before the missed load; or it allows stores to be performed before the missed store.**

# Using Re-Order Buffer (ROB) for Preserving: The order in which instruction are <completed>

- 1. Instruction are written into the ROB in strict program order:
  - **One new entry is allocated for each active instruction**
- 2. Each entry indicates the status of the corresponding instruction
  - **issued (i), in execution (x), already finished (f)**
- 3. An instruction is allowed to retire only if it has finished and all previous instruction are already retired.
  - **retiring in strict program order**
  - **only retiring instructions are permitted to complete, that is, to update the program state:**
    - by writing their result into the referenced architectural register or memory

# Principle of the ROB {Circular Buffer}

# Introduction of ROBs in commercial superscalar processors

[1] The Nx586 has scalar issue for CISC instructions but a 3-way superscalar core for converted RISC instructions

# Use ROB for speculative execution

- Guess the outcome of a branch and execution the path
  - **before the condition is ready**
- 1. Each entry is extended to include a speculative status field
  - **indicating whether the corresponding instruction has been executed speculatively**
- 2. speculatively executed instruction are not allow to retire
  - **before the related condition is resolved**
- 3. After the related condition is resolved,
  - **if the guess turn out to be right, the instruction can retire in order.**
  - **if the guess is wrong, the speculative instructions are marked to be cancelled.**
    **Then, instruction execution continue with the correct instructions.**

# Design space of ROBs

# Basic layout of ROBs

# ROB implementation details

| | *ROB size* | *Issue rate* | *Retire rate* | *Intermediate results stored* | *Designation* |
|---|---|---|---|---|---|
| ES/9000 (1992p) | 32 | 2 | 2 | No | Completion control logic |
| PowerPC 602 (1995) | 4 | 2 | 1 | n.a. | Completion unit |
| PowerPC 603 (1993) | 5 | 3 | 2 | No | Completion buffer |
| PowerPC 604 (1995) | 16 | 4 | 4 | No | ROB |
| PowerPC 620 (1996) | 16 | 4 | 4 | No | ROB |
| PentiumPro (1995) | 40 | 3 | 3 | Yes | ROB |
| Am29000 sup (1995) | 10 | 4 | 2 | Yes | ROB |
| K5 (1995) | 16 | 4 | 4 | Yes | ROB |
| PM1 (Sparc64, 1995) | 64 | 4 | 4 | No | Precise state unit |
| UltraSparc (1995) | n.a. | 4 | n.a. | n.a. | Completion unit |
| PA 8000 (1996) | 56 | 4 | 4 | Yes | Instruction reorder buffer |
| R10000 (1996) | 32 | 4 | 4 | No | Active list |

# 7.1 Specific tasks of superscalar processing

# 7.8 Preserving the Sequential consistency of exception processing

- When instructions are executed in parallel,
  - **interrupt request, which are caused by exceptions arising in instruction <execution>,**
  - **are also generated out of order.**
- If the requests are acted upon immediately,
  - **the requests are handled in different order than in a sequential operation processor**
  - **called imprecise interrupts**
- Precise interrupts: handling the interrupts in consistent with the state of a sequential processor

# Sequential consistency of exception processing

**Sequential consistency of exception processing**

**Weak consistency**

↓

**Imprecise interrupts**

*Power1 (1990)*
*Power2 (1993)*
*α processors*

**Strong consistency**

↓

**Precise interrupts**

*MC88110 (1993)*
*Pentium (1993) and usually processors making use of an ROB, such as:*

*ES/9000 (1992p)*
*PowerPC line*
*PA 8000 (1996)*
*R10000 (1996)*

# Use ROB for preserving sequential order of interrupt requests

- Interrupts generated in connection with instruction execution
  - **can handled at the correct point in the execution,**
  - **by accepting interrupt requests only when the related instruction becomes the next to retire.**

# 7.9 Implementation of superscalar CISC processors using superscalar RISC core

- CISC instructions are first converted into RISC-like instructions <during decoding>.
  - **Simple CISC register-to-register instructions are converted to single RISC operation (1-to-1)**
  - **CISC ALU instructions referring to memory are converted to two or more RISC operations (1-to-(2-4))**
    - SUB EAX, [EDI]
      - converted to e.g.
    - MOV EBX, [EDI]
    - SUB EAX, EBX
  - **More complex CISC instructions are converted to long sequences of RISC operations (1-to-(more than 4))**
- On average one CISC instruction is converted to 1.5-2 RISC operations

# The peinciple of superscalar CISC execution using a superscalar RISC core

# PentiumPro: Decoding/converting CISC instructions to RISC operations (are done in program order)

# Case Studies: R10000
# Core part of the micro-architecture of the R10000

# Case Studies: PowerPC 620

# Case Studies: PentiumPro
# Core part of the micro-architecture

# PentiumPro Long pipeline:
# Layout of the FX and load pipelines

**National Institute of Technology Karnataka Surathkal**
**Department of Information Technology**

# IT 301 Parallel Computing

## Performance Evaluation
## Amdahl's Law

**Dr. Geetha V**

*Assistant Professor*

*Dept of Information Technology*

*NITK Surathkal*

# Course Outline

**Course Plan: Theory:**

**Part A: Parallel Computer Architectures**

Week 1,2,3: ***Introduction to Parallel Computer Architecture:*** Parallel Computing, Parallel architecture, bit level, instruction level , data level and task level parallelism. Instruction level parelllelism: pipelining(Data and control instructions), scalar and superscalar processors, vector processors. Parallel computers and computation.

Week 4,5: Memory Models: UMA, NUMA and COMA. Flynns classification, Cache coherence,

Week 6,7: Amdahl's Law. Performance evaluation, Designing parallel algorithms : Divide and conquer, Load balancing, Pipelining.

Week 8 -11: ***Parallel Programming techniques like Task Parallelism using TBB, TL2, Cilk++ etc. and software transactional memory techniques.***

# Course Outline

**Part B: OpenMP/MPI/CUDA**

Week 1,2,3 : ***Shared Memory Programing Techniques:*** Introduction to OpenMP : Directives: *parallel, for, sections, task, master, single, critical, barrier, taskwait, atomic. Clauses: private, shared, firstprivate, lastprivate, reduction, nowait, ordered, schedule, collapse, num_threads, if(), threadprivate, copyin, copyprivate*

Week 4,5: ***Distributed Memory programming Techniques:*** MPI: Blocking, Non-blocking.

Week 6,7 : CUDA : OpenCL, Execution models, GPU memory, GPU libraries.

Week 10,11,: ***Introduction to accelerator programming using CUDA/OpenCL and Xeon-phi. Concepts of Heterogeneous programming techniques.***

**Practical:**

Implementation of parallel programs using OpenMP/MPI/CUDA.

**Assignment:** Performance evaluation of parallel algorithms (in group of 2 or 3 members)

# Parallel Processing → Speed Up

- Speed up is the factor by which the time is reduced compared to a single processor.

$$\textbf{Speedup} = \frac{\textbf{Execution time using one processor}}{\textbf{Execution time using multiple processors}}$$

**Example:** It time taken to perform a task ($T_1$) using **one processor is 1 unit** then time taken to perform the same task ($T_N$) on **N processors is 1/N unit.**

**Then Speed up is =** $T_1/T_N = \frac{1}{1/N}$

# Parallel Processing → Speed Up

- Speedup is linear !
- That means as the number of processors increase, the speed up should also increase.
- Ideally this is what is to be achieved.
- But it is not practically achieved.
- Thus Amdahl's law gives us a better insight.

# Amdahl's Law

- Speed up depends on the portion of program that is serial and portion of program that is parallel or enhanced.
- So the PARALLEL program (T) will have SERIAL part (S) + PARALLEL part (P)
- In single processor Time taken by Parallel program $T_1 = S+P$
- In multiple processor (say n processors) Time taken by $T_n = S+P/n$
- If time required for parallel program on single processor is 1 unit. Then time taken to execute Serial portion is $S=1-P$

$$\text{Speed up} = \frac{T_1}{T_n}$$

FINALLY

$$\text{Speed up} = \frac{1}{(1-P) + P/n}$$

# THANK YOU

**National Institute of Technology Karnataka Surathkal**
**Department of Information Technology**

# IT 301 Parallel Computing

## Memory Models and Cache Coherence

**Dr. Geetha V**

*Assistant Professor*

*Dept of Information Technology*

*NITK Surathkal*

# Course Outline

**Course Plan: Theory:**

**Part A: Parallel Computer Architectures**

Week 1,2,3: ***Introduction to Parallel Computer Architecture:*** Parallel Computing, Parallel architecture, bit level, instruction level , data level and task level parallelism. Instruction level parelllelism: pipelining(Data and control instructions), scalar and superscalar processors, vector processors. Parallel computers and computation.

Week 4,5: Memory Models: UMA, NUMA and COMA. Flynns classification, Cache coherence,

Week 6,7: Amdahl's Law. Performance evaluation, Designing parallel algorithms : Divide and conquer, Load balancing, Pipelining.

Week 8 -11: ***Parallel Programming techniques like Task Parallelism using TBB, TL2, Cilk++ etc. and software transactional memory techniques.***

# Course Outline

**Part B: OpenMP/MPI/CUDA**

Week 1,2,3 : ***Shared Memory Programing Techniques:*** Introduction to OpenMP : Directives: *parallel, for, sections, task, master, single, critical, barrier, taskwait, atomic. Clauses: private, shared, firstprivate, lastprivate, reduction, nowait, ordered, schedule, collapse, num_threads, if(), threadprivate, copyin, copyprivate*

Week 4,5: ***Distributed Memory programming Techniques:*** MPI: Blocking, Non-blocking.

Week 6,7 : CUDA : OpenCL, Execution models, GPU memory, GPU libraries.

Week 10,11,: ***Introduction to accelerator programming using CUDA/OpenCL and Xeon-phi. Concepts of Heterogeneous programming techniques.***

**Practical:**

Implementation of parallel programs using OpenMP/MPI/CUDA.

**Assignment:** Performance evaluation of parallel algorithms (in group of 2 or 3 members)

# Index

1. Memory models
2. Cache Coherence
    1. Cache coherence problem
    2. Snooping Bus Protocols: MSI.MESI

# 2.Memory Models

## Parallel Computers Architectural Model/ Physical Model

Distinguished by having-

**1. Shared Common Memory:**

Three Shared-Memory Multiprocessor Models are:

i. UMA (Uniform-Memory Access)

ii. NUMA (Non-Uniform-Memory Access)

iii. COMA (Cache-Only Memory Architecture)

**2. Unshared Distributed Memory**

i. CC-NUMA (Cache-Coherent -NUMA)

# Memory Models: UMA, NUMA and COMA

Uniform Memory Access

Non Uniform Memory Access

(a) Shared local memories

LM – Local Memory
P - Local Processor

Non Uniform Memory Access

(b) A hierarchical cluster model
(Access of Remote Memory)

P – Processor
CSM – Cluster Shared Memory
CIN – Cluster Interconnection Network
GSM – Global Shared Memory

Cache Only Memory Access

C – Cache
P – Processor
D - Directories

- Distributed Main Memory converted to Cache
- Cache form Global Address Space
- Remote Cache access by – Distributed cache Directories

# Shared Memory Model

Scenario when processors shared data block and the data block is consistent

- Main memory is shared among all the processors.
- Accessing main memory every time is time consuming
- Better to have private cache for all the processors
- Shared memory consistency must be maintained.
- When the data block is shared among all other processors, the change in one cache must be reflected in all other processor cache. Then, the system can be considered as memory consistency is maintained.

# Shared Memory Model

- Cache Coherent Problem:
- Caching of shared data, however, introduces **Cache Coherence problem.**
- **Shared data can have different data in different cache.**
- This must be handled properly

# Shared Memory Model

**Write-Back Protocol**

**Processor modify a data block in cache and updates main memory when other processor request for the modified data block**

- Cache Coherent: A memory system is coherent if any read of a data items return the most recently written value of that data item .

# Shared Memory Model

- A memory system is consistent if the following hold good:

1. A read by a processor P to a location X that follows a write by P to X, with no writes of X by another processor occurring between the write and the read by P, always returns the value written by P. **: Maintains program order**

**Assume X=5;**

**N=Read X; Write X, 10; N=Read X : Here, P must get value 10 , written by P**

2. A read by a processor to location X that follows a write by another processor to X returns the written value if the read and write are sufficiently separated in time and no other writes to X occur between the two accesses. **: Coherent rule of memory**

**Assume X=5;**

**N=Read X by P1; Write X, 10 by P2; N=Read X by P1 : Here, must get value 10 , written by P2**

3. Writes to the same location are *serialized;* that is, two writes to the same location by any two processors are seen in the same order by all processors. This ensures that we do not see the older value after the newer value. **: Serialization of memory operations**

**Assume X=5;**

**Write X, 10 by P1; Write X, 15 by P2; N=Read X by P1 : Here, must get value 15 , written by P2**

# Cache Coherence Protocols

**Write-Through Protocol**
Processor modify a data block in cache and also update main memory with the new copy of that data block

- **Memory Write Operations**

## 1. Write Through:

- When a processor modifies a data block in its cache, it immediately updates the main memory with the new copy of the same data block.
- Main memory has consistent data.
- Cache coherence : Either through update or invalidate

# Cache Coherence Protocols

- **Memory Write Operations**

**1. Write Back:**

- When a processor modifies a data block in its cache, it **does not** immediately update the main memory with the new copy of the same data block. Write back durig cache replacement or request from other processors.
- That processor has the data exclusively
- **Cache coherence : invalidate all other cache entries and update when there is request.**

# Cache Coherence Protocols : Snoopy Bus Protocol

- **Snoopy Bus Protocol**
- Main memory is shared among processors via single bus.
- Cache controller performs snoop operation on all transaction over bus.
- Snoop bus protocol is hardware solution.
- It is used
  for small multiprocessor environment

# Snoopy Bus Protocol MSI:Modified-Shared-Invalid

- **Snoopy Bus Protocol**
- Invalid State: Initially all the processor cache elements are invalid.

# Snoopy Bus Protocol MSI:Modified-Shared-Invalid

- **Snoopy Bus Protocol**
- Shared State: Initially all the processor cache elements are invalid.

I-S: PrRd/BusRdx: P1 sending read

S-M: PrWr/BusUpgr : P1 in shared state, send PrWr

# Snoopy Bus Protocol MSI:Modified-Shared-Invalid

- **Snoopy Bus Protocol**
- Modified State: Initially all the processor cache elements are invalid.

I-M: PrWr/BusRdx: P1 sending Write

M-S: BusRd: P2 sending PrRd

# Snoopy Bus Protocol MESI:Modified-Exclusive-Shared-Invalid

- **Snoopy Bus Protocol: MESI**
- Invalid State: The data items are invalid initially.
- Shared state: Valid copy of data in many cache
- Exclusive: The processor wants to perform write operation, and it has exclusive copy. **Invalid to Excusive**
- Modified State: Data block is modified, and the processor has modified data. Shared to Modified.

# Summary

1. Memory models
2. Cache Coherence
    1. Cache coherence problem
    2. Snooping Bus Protocols: MSI.MESI

# Reference

**Text Books and/or Reference Books:**


1. Professional CUDA C Programming – John Cheng, Max Grossman, Ty McKercher, 2014
2. B.Wilkinson, M.Allen, ”Parallel Programming: Techniques and Applications Using Networked Workstations and Parallel Computers”, Pearson Education, 1999
3. I.Foster, ”Designing and building parallel programs”, 2003
4. Parallel Programming in C using OpenMP and MPI – Micheal J Quinn, 2004
5. Introduction to Parallel Programming – Peter S Pacheco, Morgan Kaufmann Publishers, 2011
6. Advanced Computer Architectures: A design approach, Dezso Sima, Terence Fountain, Peter Kacsuk, 2002
7. Parallel Computer Architecture : A hardware/Software Approach, David E Culler, Jaswinder Pal Singh Anoop Gupta, 2011
8. Introduction to Parallel Computing, Ananth Grama, Anshul Gupta, George Karypis, Vipin Kumar, Pearson, 2011

# Thank You

**National Institute of Technology Karnataka Surathkal**
**Department of Information Technology**

# IT 301 Parallel Computing

## Designing Parallel Algorithm (1)

**Dr. Geetha V**

*Assistant Professor*

*Dept of Information Technology*

*NITK Surathkal*

# Course Outline

**Course Plan: Theory:**

**Part A: Parallel Computer Architectures**

Week 1,2,3: ***Introduction to Parallel Computer Architecture:*** Parallel Computing, Parallel architecture, bit level, instruction level , data level and task level parallelism. Instruction level parelllelism: pipelining(Data and control instructions), scalar and superscalar processors, vector processors. Parallel computers and computation.

Week 4,5: Memory Models: UMA, NUMA and COMA. Flynns classification, Cache coherence,

Week 6,7: Amdahl's Law. Performance evaluation, Designing parallel algorithms : Divide and conquer, Load balancing, Pipelining.

Week 8 -11: ***Parallel Programming techniques like Task Parallelism using TBB, TL2, Cilk++ etc. and software transactional memory techniques.***

# Course Outline

**Part B: OpenMP/MPI/CUDA**

Week 1,2,3 : ***Shared Memory Programing Techniques:*** Introduction to OpenMP : Directives: *parallel, for, sections, task, master, single, critical, barrier, taskwait, atomic. Clauses: private, shared, firstprivate, lastprivate, reduction, nowait, ordered, schedule, collapse, num_threads, if(), threadprivate, copyin, copyprivate*

Week 4,5: ***Distributed Memory programming Techniques:*** MPI: Blocking, Non-blocking.

Week 6,7 : CUDA : OpenCL, Execution models, GPU memory, GPU libraries.

Week 10,11,: ***Introduction to accelerator programming using CUDA/OpenCL and Xeon-phi. Concepts of Heterogeneous programming techniques.***

**Practical:**

Implementation of parallel programs using OpenMP/MPI/CUDA.

**Assignment:** Performance evaluation of parallel algorithms (in group of 2 or 3 members)

# Index

1. Potential for increased computational speed
2. Embarrassingly Parallel Computations
3. Partitions and Divide and Conquer Strategies
4. Pipelined Computations
5. Load balancing and termination Detection

# Index

1. Potential for increased computational speed
   - Speedup factor
   - Maximum speedup
   - Message passing computations
2. Embarrassingly Parallel Computations
3. Partitions and Divide and Conquer Strategies
4. Pipelined Computations
5. Load balancing and termination Detection

# Index

1. Potential for increased computational speed
   - Speedup factor
   - Maximum speedup
   - Message passing computations
2. Embarrassingly Parallel Computations
3. Partitions and Divide and Conquer Strategies
4. Pipelined Computations
5. Load balancing and termination Detection

# 1. Potential for increased computational speedup

- Using parallel processes or parallel processors how much speedup we can achieve?
- How to quantify the improvement?
- Consider there are $p$ processors or processes. It means we are having multiprocessor system. In what way we can compute the speedup?
- What is the scalability factor?

# 1. Potential for increased computational speedup

- **1.1 Speedup Factor**
- The Speedup factor S(p) is a measure of relative performance, which is defined as:

$$S(p) = \frac{\text{Execution time using single processor system (with the best sequential algorithm)}}{\text{Execution time using a multiprocessor with } p \text{ processors}}$$

$$S(p) = \frac{t_s}{t_p} \qquad S(p) \le \frac{t_s}{t_s/p} = p$$

$$S(p) = \frac{\text{Number of computational steps using one processor}}{\text{Number of parallel computational steps with } p \text{ processors}}$$

- The maximum speedup possible is usually p with p processors(linear speedup)
- Superlinear speedup : where S(p) >p, may be seen on occasion, based on memory organisation

# 1. Potential for increased computational speedup

- **1.1 Speedup Factor**
- **Efficiency:** Efficiency of the system indicates how long the processors are being used on the computation.

$$E = \frac{\text{Execution time using one processor}}{\text{Execution time using a multiprocessor} \times \text{number of processors}}$$

$$= \frac{t_s}{t_p \times p}$$

- When E is given as percentage

$$E = \frac{S(p)}{p} \times 100\%$$

- When E=50% it indicates that the processor are being used half the time of the actual computation.
- The efficiency of 100% occurs when all the processors are being used on the computation at all the times and the speedup factor S(p) is p.

# 1. Potential for increased computational speedup

- **1.2 Maximum Speedup**
- Several factors influences on speedup.
  - The clockcyles when processors not performing useful work and idle
  - Extracomputation in parallel version which was not there in serial computation
  - Communication time between the processes

# 1. Potential for increased computational speedup

- **1.2 Maximum Speedup**
- Amdahl's Law: If a fraction of the computation that cannot be divided into concurent tasks, is ***f***, and no overhead is incorred when the computation is divided into concurrent parts, the time to perform the computation with p processors is given by $ft_s+(1-f)t_s/p$.
- So speedup factor is

$$S(p) = \frac{t_s}{ft_s + (1-f)t_s/p} = \frac{p}{1+(p-1)f}$$

- Even with infinite processor,

the maximum speedup achieved

Is limited to ***1/f***.

# 1. Potential for increased computational speedup

- **1.2 Maximum Speedup**
- **Scalability:** Perfromance of the system depend upon the size of the system, I.e., the number of processors, and generally the larger the system the better, but this comes with a cost.
- **Hardware scalability:** increase in hardware design
- **Algorithmic scalability:** A parallel algorithm can accommodate increased data items with low and bounded increase in computational steps.

# 1. Potential for increased computational speedup

- **1.2 Maximum Speedup**
- **Gustafson Law:** Gustafson (1988), presetned an argument based upon scalability concepts to show that Amdahl's law was not as significant as first supposed in determining the potential speedup limits.
- Scaled speedup factor (i.e., the speedup factor when the problem is scaled). In Gustafson's law, the parallel executionm time $t_p$ is constant rather than serial execution time $t_s$.
- Then parallel execution time is $t_p = ft_s + (1-f)t_s/p = 1$
- Serial execution time is $t_s = p + (1-p)ft_s$

$$S_s(p) = \frac{ft_s + (1-f)t_s}{ft_s + (1-f)t_s/p} = \frac{p + (1-p)ft_s}{1} = p + (1-p)ft_s$$

# 1. Potential for increased computational speedup

- **1.2 Maximum Speedup**
- **Gustafson Law:** Then parallel execution time is $t_p = ft_s + (1-f)t_s/p = 1$
- Serial execution time is $t_s = p + (1-p)ft_s$

$$S_s(p) = \frac{ft_s + (1-f)t_s}{ft_s + (1-f)t_s/p} = \frac{p + (1-p)ft_s}{1} = p + (1-p)ft_s$$

- There are two assumptions in this quation:
- A) the parallel execution time is constant;
- B) the part that must be executed by sequentially, $ft_s$ is also constant and not function of $p$
- Gustafson's observation here is that the scaled speedup factor is line of negative slope (1-p).

# 1. Potential for increased computational speedup

- **Example**
- **Suppose we had a 5% serial section and 20 processors, then speed up is as follows:**
- **Amdahl's law:**

$$S(p) = \frac{t_s}{ft_s + (1-f)t_s/p} = \frac{p}{1+(p-1)f}$$

- **20/(1+(20-1)*0.05) = 20/1+0.95 =10.26**

**Gustafson's law:**

$$S_s(p) = \frac{ft_s + (1-f)t_s}{ft_s + (1-f)t_s/p} = \frac{p + (1-p)ft_s}{1} = p + (1-p)ft_s$$

- S(p) = 0.05+0.95(20) = 19.05

# 1. Potential for increased computational speedup

- **1.3 Message passing Computations**
- Parallel programming contains computation and communication part.
- Hence

$$t_p = t_{comm} + t_{comp}$$

where $t_{comm}$ is the communication time, and $t_{comp}$ is the computation time.

- It is essential to reduce the communication overhead because of significant time taken by interprocessor communication.
- Computation/communication ratio can be used as metric.

$$\text{Computation/communication ratio} = \frac{\text{Computation time}}{\text{Communication time}} = \frac{t_{comp}}{t_{comm}}$$

# 1. Potential for increased computational speedup

- **Exercise:**

1. A multiprocessor consists of 100 processors, each capable of a peak execution rate of 2 Gflops. What is the performance of the system as meeasured in Gflops when 10% of the code is sequential and 90% is parallelizable?

# Reference

**Text Books and/or Reference Books:**


1. Professional CUDA C Programming – John Cheng, Max Grossman, Ty McKercher, 2014
2. **B.Wilkinson, M.Allen, ”Parallel Programming: Techniques and Applications Using Networked**

Workstations and Parallel Computers”, Pearson Education, 1999

3. I.Foster, ”Designing and building parallel programs”, 2003
4. Parallel Programming in C using OpenMP and MPI – Micheal J Quinn, 2004
5. Introduction to Parallel Programming – Peter S Pacheco, Morgan Kaufmann Publishers, 2011
6. Advanced Computer Architectures: A design approach, Dezso Sima, Terence Fountain, Peter Kacsuk, 2002
7. Parallel Computer Architecture : A hardware/Software Approach, David E Culler, Jaswinder Pal Singh Anoop Gupta, 2011
8. Introduction to Parallel Computing, Ananth Grama, Anshul Gupta, George Karypis, Vipin Kumar, Pearson, 2011

# Thank You

**National Institute of Technology Karnataka Surathkal**
**Department of Information Technology**

# IT 301 Parallel Computing

## Designing Parallel Algorithm (2)

**Dr. Geetha V**

*Assistant Professor*

*Dept of Information Technology*

*NITK Surathkal*

# Course Outline

**Course Plan: Theory:**

**Part A: Parallel Computer Architectures**

Week 1,2,3: ***Introduction to Parallel Computer Architecture:*** Parallel Computing, Parallel architecture, bit level, instruction level , data level and task level parallelism. Instruction level parelllelism: pipelining(Data and control instructions), scalar and superscalar processors, vector processors. Parallel computers and computation.

Week 4,5: Memory Models: UMA, NUMA and COMA. Flynns classification, Cache coherence,

Week 6,7: Amdahl's Law. Performance evaluation, Designing parallel algorithms : Divide and conquer, Load balancing, Pipelining.

Week 8 -11: ***Parallel Programming techniques like Task Parallelism using TBB, TL2, Cilk++ etc. and software transactional memory techniques.***

# Course Outline

**Part B: OpenMP/MPI/CUDA**

Week 1,2,3 : ***Shared Memory Programing Techniques:*** Introduction to OpenMP : Directives: *parallel, for, sections, task, master, single, critical, barrier, taskwait, atomic. Clauses: private, shared, firstprivate, lastprivate, reduction, nowait, ordered, schedule, collapse, num_threads, if(), threadprivate, copyin, copyprivate*

Week 4,5: ***Distributed Memory programming Techniques:*** MPI: Blocking, Non-blocking.

Week 6,7 : CUDA : OpenCL, Execution models, GPU memory, GPU libraries.

Week 10,11,: ***Introduction to accelerator programming using CUDA/OpenCL and Xeon-phi. Concepts of Heterogeneous programming techniques.***

**Practical:**

Implementation of parallel programs using OpenMP/MPI/CUDA.

**Assignment:** Performance evaluation of parallel algorithms (in group of 2 or 3 members)

# Index

1. Potential for increased computational speed
2. **Embarrassingly Parallel Computations**
3. Partitions and Divide and Conquer Strategies
4. Pipelined Computations
5. Load balancing and termination Detection

# Index

1. Potential for increased computational speed
2. **Embarrassingly Parallel Computations**
   - **Ideal parallel computation**
   - **Geometrical transformations of images**
   - **Analysis**
3. Partitions and Divide and Conquer Strategies
4. Pipelined Computations
5. Load balancing and termination Detection

# 2.1 Embarrassingly Parallel Computations

- A computation that can be divided into several completely independent parts, each of which can be executed by a separate processor is known as ***embarrassingly parallel computation.***
- Ideal parallel computation: where a problem can be divided into completely independent parts that can be executed simultaneously.

**Figure 3.1** Disconnected computational graph (embarrassingly parallel problem).

**Figure 3.2** Practical embarrassingly parallel computational graph with dynamic process creation and the master-slave approach.

# 2.2. Geometrical Transformation of Images

- Consider image processing in computer graphics. Geometrical transformation on images is performing computation with respect to each pixel.
- Here (x,y) values are changed as x' and y'.
- Shifting

$$x' = x + \Delta x$$
$$y' = y + \Delta y$$

Rotation

$$x' = x\cos\theta + y\sin\theta$$
$$y' = -x\sin\theta + y\cos\theta$$

- Scaling

$$x' = xS_x$$
$$y' = yS_y$$

Clipping

$$x_l \le x' \le x_h$$
$$y_l \le y' \le y_h$$

# 2.3. Analysis

- Assume 640 x 480 image and 48 processors
- Partition of data can be in different ways.
- Original bit map in array map[][] and temp bit map in temp_map[][].

(a) Square region for each process

(b) Row region for each process

**Figure 3.3** Partitioning into regions for individual processes.

# 2.3. Analysis

- Master and slave program
- Slave receives row number and considers 10 rows of data from there.

Master

```
for (i = 0, row = 0; i < 48; i++, row = row + 10)   /* for each process*/
   send(row, P_i);                                   /* send row no.*/

for (i = 0; i < 480; i++)                            /* initialize temp */
   for (j = 0; j < 640; j++)
      temp_map[i][j] = 0;

for (i = 0; i < (640 * 480); i++) {                  /* for each pixel */
   recv(oldrow,oldcol,newrow,newcol, P_ANY);         /* accept new coords */
   if !((newrow < 0)||(newrow >= 480)||(newcol < 0)||(newcol >= 640))
      temp_map[newrow][newcol]=map[oldrow][oldcol];
}
for (i = 0; i < 480; i++)                            /* update bitmap */
   for (j = 0; j < 640; j++)
      map[i][j] = temp_map[i][j];
```

Slave

```
recv(row, P_master);                                 /* receive row no. */
for (oldrow = row; oldrow < (row + 10); oldrow++)
   for (oldcol = 0; oldcol < 640; oldcol++) {        /* transform coords */
      newrow = oldrow + delta_x;                     /* shift in x direction */
      newcol = oldcol + delta_y;                     /* shift in y direction */
      send(oldrow,oldcol,newrow,newcol, P_master);   /* coords to master */
   }
```

# 2.3. Analysis

- Suppose each pixel requires two computational steps and there are nxn pixels, then for sequential execution:

$$t_s = 2n^2$$

- Sequential time complexity is $O(n^2)$
- To compute communication cost

$$t_{comm} = t_{startup} + mt_{data}$$

- Where $T_{startup}$ is the (constant) time to form the message and initiate transfer and $t_{data}$ is the time required to send one data item and there are *m* data items.

# 2.3. Analysis

- Suppose p number of processes are considered, then communication cost is

$$t_{comm} = p(t_{startup} + t_{data}) + n^2(t_{startup} + 4t_{data}) = O(p + n^2)$$

- Where $p(t_{startup} + t_{data})$ for sending row number to p processors sequentially.

- Where $n^2(t_{startup} + 4t_{data}) = O(p + n^2)$ for sending nxn pixel values from each p to server. Each time 4 coordinates are transmitted. Hence it is $4t_{data}$.

- Tcomp for parallel execution with p processes is $t_{comp} = 2\left(\frac{n^2}{p}\right) = O(n^2/p)$

# 2.3. Analysis

$$t_{comp} = 2\left(\frac{n^2}{p}\right) = O(n^2/p)$$

***Overall execution time.*** The overall execution time is given by

$$t_p = t_{comp} + t_{comm}$$

For a fixed number of processors, this is $O(n^2)$.

***Speedup factor.*** The speedup factor is

$$\text{Speedup factor} = \frac{t_s}{t_p} = \frac{2n^2}{2\left(\frac{n^2}{p}\right) + p(t_{startup} + t_{data}) + n^2(t_{startup} + 4t_{data})}$$

# 2.3. Analysis

***Computation/communication ratio.*** As introduced in Chapter 1, the ratio of the computation and communication times:

$$\text{Computation/communication ratio} = \frac{\text{Computation time}}{\text{Communication time}} = \frac{t_{\text{comp}}}{t_{\text{comm}}}$$

enables one to see the effects of the communication overhead, especially for increasing problem size. In the problem under consideration, the ratio is

$$\text{Computation/communication ratio} = \frac{2(n^2/p)}{p(t_{\text{startup}} + 2t_{\text{data}}) + 4n^2(t_{\text{startup}} + t_{\text{data}})}$$

$$= O\left(\frac{n^2/p}{p + n^2}\right)$$

# Reference

**Text Books and/or Reference Books:**


1. Professional CUDA C Programming – John Cheng, Max Grossman, Ty McKercher, 2014
2. **B.Wilkinson, M.Allen, ”Parallel Programming: Techniques and Applications Using Networked**

Workstations and Parallel Computers”, Pearson Education, 1999

3. I.Foster, ”Designing and building parallel programs”, 2003
4. Parallel Programming in C using OpenMP and MPI – Micheal J Quinn, 2004
5. Introduction to Parallel Programming – Peter S Pacheco, Morgan Kaufmann Publishers, 2011
6. Advanced Computer Architectures: A design approach, Dezso Sima, Terence Fountain, Peter Kacsuk, 2002
7. Parallel Computer Architecture : A hardware/Software Approach, David E Culler, Jaswinder Pal Singh Anoop Gupta, 2011
8. Introduction to Parallel Computing, Ananth Grama, Anshul Gupta, George Karypis, Vipin Kumar, Pearson, 2011

# Thank You

**National Institute of Technology Karnataka Surathkal**
**Department of Information Technology**

# IT 301 Parallel Computing

## Designing Parallel Algorithm (3)

**Dr. Geetha V**

*Assistant Professor*

*Dept of Information Technology*

*NITK Surathkal*

# Course Outline

**Course Plan: Theory:**

**Part A: Parallel Computer Architectures**

Week 1,2,3: ***Introduction to Parallel Computer Architecture:*** Parallel Computing, Parallel architecture, bit level, instruction level , data level and task level parallelism. Instruction level parelllelism: pipelining(Data and control instructions), scalar and superscalar processors, vector processors. Parallel computers and computation.

Week 4,5: Memory Models: UMA, NUMA and COMA. Flynns classification, Cache coherence,

Week 6,7: Amdahl's Law. Performance evaluation, Designing parallel algorithms : Divide and conquer, Load balancing, Pipelining.

Week 8 -11: ***Parallel Programming techniques like Task Parallelism using TBB, TL2, Cilk++ etc. and software transactional memory techniques.***

# Course Outline

**Part B: OpenMP/MPI/CUDA**

Week 1,2,3 : ***Shared Memory Programing Techniques:*** Introduction to OpenMP : Directives: *parallel, for, sections, task, master, single, critical, barrier, taskwait, atomic. Clauses: private, shared, firstprivate, lastprivate, reduction, nowait, ordered, schedule, collapse, num_threads, if(), threadprivate, copyin, copyprivate*

Week 4,5: ***Distributed Memory programming Techniques:*** MPI: Blocking, Non-blocking.

Week 6,7 : CUDA : OpenCL, Execution models, GPU memory, GPU libraries.

Week 10,11,: ***Introduction to accelerator programming using CUDA/OpenCL and Xeon-phi. Concepts of Heterogeneous programming techniques.***

**Practical:**

Implementation of parallel programs using OpenMP/MPI/CUDA.

**Assignment:** Performance evaluation of parallel algorithms (in group of 2 or 3 members)

# Index

1. Potential for increased computational speed
2. **Embarrassingly Parallel Computations**
3. Partitions and Divide and Conquer Strategies
4. Pipelined Computations
5. Load balancing and termination Detection

# Index

1. Potential for increased computational speed
2. Embarrassingly Parallel Computations
3. **Partitions and Divide and Conquer Strategies**
4. Pipelined Computations
5. Load balancing and termination Detection

# 3.1 Partitioning

- Partitioning Strategies
  - Data partitioning or domain decomposition
  - Functional decomposition
- Example: finding sum of N elements

**Figure 4.1** Partitioning a sequence of numbers into parts and adding them.

# 3.1 Partitioning

- Assume N elements are divided among p groups
- Separate send() and receive() is used then communication is as shown.

Master

```
s = n/p;                                  /* number of numbers for slaves*/
for (i = 0, x = 0; i < p; i++, x = x + s)
   send(&numbers[x], s, P_i);             /* send s numbers to slave */

sum = 0;
for (i = 0; i < p; i++) {                 /* wait for results from slaves */
   recv(&part_sum, P_ANY);
   sum = sum + part_sum;                  /* accumulate partial sums */
}
```

Slave

```
recv(numbers, s, P_master);               /* receive s numbers from master */
part_sum = 0;
for (i = 0; i < s; i++)                   /* add numbers */
   part_sum = part_sum + numbers[i];
send(&part_sum, P_master);                /* send sum to master */
```

# 3.1 Partitioning

- Assume N elements are divided among p groups
- If broadcast or multicast is used, then communication is as follows

Master

```
s = n/p;                                  /* number of numbers for slaves */
bcast(numbers, s, P_slave_group);         /* send all numbers to slaves */
sum = 0;
for (i = 0; i < p; i++) {                 /* wait for results from slaves */
   recv(&part_sum, P_ANY);
   sum = sum + part_sum;                  /* accumulate partial sums */
}
```

Slave

```
bcast(numbers, s, P_master);              /* receive all numbers from master*/
start = slave_number * s;                 /* slave number obtained earlier */
end = start + s;
part_sum = 0;
for (i = start; i < end; i++)             /* add numbers */
   part_sum = part_sum + numbers[i];
send(&part_sum, P_master);                /* send sum to master */
```

# 3.1 Partitioning

- Assume N elements are divided among p groups
- When scatter or reduce is used ...

Master

```
s = n/p;                                        /* number of numbers */
scatter(numbers,&s,P_group,root=master);        /* send numbers to slaves */
reduce_add(&sum,&s,P_group,root=master);        /* results from slaves */
```

Slave

```
scatter(numbers,&s,P_group,root=master);        /* receive s numbers */
                                                /* add numbers */
:
reduce_add(&part_sum,&s,P_group,root=master);   /* send sum to master */
```

# 3.2. Analysis

- Sequential computation requires n-1 additions and time complexity is O(n)
- Parallel computation with p processors:
- Phase –I : communication: Each processor reading their n/p numbers

$$t_{comm1} = p(t_{startup} + (n/p)t_{data})$$

- Is scatter is used then the computation time can be

$$t_{comm1} = t_{startup} + nt_{data}$$

- Phase-II: Computation: (n/p) element addition

$$t_{comp1} = n/p - 1$$

- Phase III: Communication: $t_{comm2} = p(t_{startup} + t_{data})$
- Phase IV: $t_{comp2} = p - 1$

# 3.2 Analysis

- Overall execution time

$$
\begin{aligned}
t_p &= (t_{comm1} + t_{comm2}) + (t_{comp1} + t_{comp2}) \\
&= p(t_{startup} + (n/p)t_{data} + p(t_{startup} + t_{data}) + (n/p - 1 + p - 1) \\
&= 2pt_{startup} + (n + p)t_{data} + p + n/p - 2
\end{aligned}
$$

or

$$t_p = O(n)$$

# 3.2 Analysis

- Speedup factor time

***Speedup Factor.*** The speedup factor is

$$\text{Speedup factor} = \frac{t_s}{t_p} = \frac{n-1}{2pt_{\text{startup}} + (n+p)t_{\text{data}} + p + n/p - 2}$$

which suggests little speedup for a fixed number of processors.

***Computation/communication ratio.*** The computation/communication ratio is given by

$$\text{Computation/communication ratio} = \frac{t_{\text{comp}}}{t_{\text{comm}}} = \frac{p + n/p - 2}{2pt_{\text{startup}} + (n+p)t_{\text{data}}}$$

Ignoring the communication aspect, the speedup factor is given by

$$\text{Speedup factor} = \frac{t_s}{t_p} = \frac{n-1}{n/p + p - 2}$$

# Reference

**Text Books and/or Reference Books:**


1. Professional CUDA C Programming – John Cheng, Max Grossman, Ty McKercher, 2014
2. **B.Wilkinson, M.Allen, ”Parallel Programming: Techniques and Applications Using Networked**

Workstations and Parallel Computers”, Pearson Education, 1999

3. I.Foster, ”Designing and building parallel programs”, 2003
4. Parallel Programming in C using OpenMP and MPI – Micheal J Quinn, 2004
5. Introduction to Parallel Programming – Peter S Pacheco, Morgan Kaufmann Publishers, 2011
6. Advanced Computer Architectures: A design approach, Dezso Sima, Terence Fountain, Peter Kacsuk, 2002
7. Parallel Computer Architecture : A hardware/Software Approach, David E Culler, Jaswinder Pal Singh Anoop Gupta, 2011
8. Introduction to Parallel Computing, Ananth Grama, Anshul Gupta, George Karypis, Vipin Kumar, Pearson, 2011

# Thank You

**National Institute of Technology Karnataka Surathkal**
**Department of Information Technology**

# IT 301 Parallel Computing

## Designing Parallel Algorithm (4)

**Dr. Geetha V**

*Assistant Professor*

*Dept of Information Technology*

*NITK Surathkal*

# Course Outline

**Course Plan: Theory:**

**Part A: Parallel Computer Architectures**

Week 1,2,3: ***Introduction to Parallel Computer Architecture:*** Parallel Computing, Parallel architecture, bit level, instruction level , data level and task level parallelism. Instruction level parelllelism: pipelining(Data and control instructions), scalar and superscalar processors, vector processors. Parallel computers and computation.

Week 4,5: Memory Models: UMA, NUMA and COMA. Flynns classification, Cache coherence,

Week 6,7: Amdahl's Law. Performance evaluation, Designing parallel algorithms : Divide and conquer, Load balancing, Pipelining.

Week 8 -11: ***Parallel Programming techniques like Task Parallelism using TBB, TL2, Cilk++ etc. and software transactional memory techniques.***

# Course Outline

**Part B: OpenMP/MPI/CUDA**

Week 1,2,3 : ***Shared Memory Programing Techniques:*** Introduction to OpenMP : Directives: *parallel, for, sections, task, master, single, critical, barrier, taskwait, atomic. Clauses: private, shared, firstprivate, lastprivate, reduction, nowait, ordered, schedule, collapse, num_threads, if(), threadprivate, copyin, copyprivate*

Week 4,5: ***Distributed Memory programming Techniques:*** MPI: Blocking, Non-blocking.

Week 6,7 : CUDA : OpenCL, Execution models, GPU memory, GPU libraries.

Week 10,11,: ***Introduction to accelerator programming using CUDA/OpenCL and Xeon-phi. Concepts of Heterogeneous programming techniques.***

**Practical:**

Implementation of parallel programs using OpenMP/MPI/CUDA.

**Assignment:** Performance evaluation of parallel algorithms (in group of 2 or 3 members)

# Index

1. Potential for increased computational speed
2. Embarrassingly Parallel Computations
3. Partitions
4. **Divide and Conquer Strategies**
5. Pipelined Computations
6. Load balancing and termination Detection

# 4.1 Divide and Conquer

- Divide and conquer approach is characterized by dividing a problem into subproblems that are of the same form as the larger problem.
- Normally further division of problem is done by recursion.
- Example: adding N elements

```
int add(int *s)                                  /* add list of numbers, s */
{
   if (number(s) <= 2) return (n1 + n2);         /* see explanation */
   else {
      Divide (s, s1, s2);          /* divide s into two parts, s1 and s2 */
      part_sum1 = add(s1);         /*recursive calls to add sub lists */
      part_sum2 = add(s2);
      return (part_sum1 + part_sum2);
   }
}
```

# 4.1 Divide and Conquer

- Divide and conquer

**Figure 4.2** Tree construction.

# 4.1 Divide and Conquer

- Divide and conquer

**Figure 4.3** Dividing a list into parts.

**Figure 4.4** Partial summation.

# 4.1 Divide and Conquer

- Divide and conquer

Process $P_0$

```
                                  /* division phase */
divide(s1, s1, s2);               /* divide s1 into two, s1 and s2 */
send(s2, P4);                     /* send one part to another process */
divide(s1, s1, s2);
send(s2, P2);
divide(s1, s1, s2);
send(s2, P1);
part_sum = *s1;                   /* combining phase */
recv(&part_sum1, P1);
part_sum = part_sum + part_sum1;
recv(&part_sum1, P2);
part_sum = part_sum + part_sum1;
recv(&part_sum1, P4);
part_sum = part_sum + part_sum1;
```

The code for process $P_4$ might take the form

Process $P_4$

```
recv(s1, P0);                     /* division phase */
divide(s1, s1, s2);
send(s2, P6);
divide(s1, s1, s2);
send(s2, P5);
part_sum = *s1;                   /* combining phase */
recv(&part_sum1, P5);
part_sum = part_sum + part_sum1;
recv(&part_sum1, P6);
part_sum = part_sum + part_sum1;
send(&part_sum, P0);
```

# 5.2 Analysis

- Divide and conquer

***Communication.*** There is a logarithmic number of steps in the division phase; that is, $\log p$ steps with $p$ processes. The communication time for this phase is given by

$$t_{\text{comm1}} = \frac{n}{2}t_{\text{data}} + \frac{n}{4}t_{\text{data}} + \frac{n}{8}t_{\text{data}} + \ldots + \frac{n}{p}t_{\text{data}} = \frac{n(p-1)}{p}t_{\text{data}}$$

where $t_{\text{data}}$ is the transmission time for one data word. The time $t_{\text{comm1}}$ is marginally better than a simple broadcast. The combining phase is similar, except that only one data item is sent in each message (the partial sum); that is,

$$t_{\text{comm2}} = (\log p)t_{\text{data}}$$

for a total communication time of

$$t_{\text{comm}} = t_{\text{comm1}} + t_{\text{comm2}} = \frac{n(p-1)}{p}t_{\text{data}} + (\log p)t_{\text{data}}$$

or a time complexity of $O(n)$ for a fixed number of processors.

# 5.2 Analysis

- Divide and conquer

***Computation.*** At the end of the divide phase, the $n/p$ numbers are added together. Then one addition occurs at each stage during the combining phase, leading to

$$t_{\text{comp}} = \frac{n}{p} + \log p$$

**Figure 4.5** Part of a search tree.

again a time complexity of $O(n)$ for a fixed number of processors. For large $n$ and variable $p$, we get $O(n/p)$.

# 5.2 Analysis

- Divide and conquer

***Overall Execution Time.*** The total parallel execution time becomes

$$t_p = \left(\frac{n(p-1)}{p} + \log p\right)t_{\text{data}} + \frac{n}{p} + \log p$$

***Speedup Factor.*** The speedup factor is

$$\text{Speedup factor} = \frac{t_s}{t_p} = \frac{n-1}{((n/p)(p-1) + \log p)t_{\text{data}} + n/p + \log p}$$

The very best speedup we could expect with this method is, of course, $p$ when all $p$ processors are computing their partial sums. The actual speedup will be less than this due to the division and combining phases.

***Computation/communication Ratio.*** The computation/communication ratio is given by

$$\text{Computation/communication ratio} = \frac{t_{\text{comp}}}{t_{\text{comm}}} = \frac{n/p + \log p}{((n/p)(p-1) + \log p)t_{\text{data}}}$$

# Index

1. Potential for increased computational speed
2. Embarrassingly Parallel Computations
3. Partitions
4. **Divide and Conquer Strategies**
5. Pipelined Computations
6. Load balancing and termination Detection

# Reference

**Text Books and/or Reference Books:**


1. Professional CUDA C Programming – John Cheng, Max Grossman, Ty McKercher, 2014
2. **B.Wilkinson, M.Allen, ”Parallel Programming: Techniques and Applications Using Networked**

Workstations and Parallel Computers”, Pearson Education, 1999

3. I.Foster, ”Designing and building parallel programs”, 2003
4. Parallel Programming in C using OpenMP and MPI – Micheal J Quinn, 2004
5. Introduction to Parallel Programming – Peter S Pacheco, Morgan Kaufmann Publishers, 2011
6. Advanced Computer Architectures: A design approach, Dezso Sima, Terence Fountain, Peter Kacsuk, 2002
7. Parallel Computer Architecture : A hardware/Software Approach, David E Culler, Jaswinder Pal Singh Anoop Gupta, 2011
8. Introduction to Parallel Computing, Ananth Grama, Anshul Gupta, George Karypis, Vipin Kumar, Pearson, 2011

# Thank You

**National Institute of Technology Karnataka Surathkal**
**Department of Information Technology**

# IT 301 Parallel Computing

## Designing Parallel Algorithm (5)

**Dr. Geetha V**

*Assistant Professor*

*Dept of Information Technology*

*NITK Surathkal*

# Course Outline

**Course Plan: Theory:**

**Part A: Parallel Computer Architectures**

Week 1,2,3: ***Introduction to Parallel Computer Architecture:*** Parallel Computing, Parallel architecture, bit level, instruction level , data level and task level parallelism. Instruction level parelllelism: pipelining(Data and control instructions), scalar and superscalar processors, vector processors. Parallel computers and computation.

Week 4,5: Memory Models: UMA, NUMA and COMA. Flynns classification, Cache coherence,

Week 6,7: Amdahl's Law. Performance evaluation, Designing parallel algorithms : Divide and conquer, Load balancing, Pipelining.

Week 8 -11: ***Parallel Programming techniques like Task Parallelism using TBB, TL2, Cilk++ etc. and software transactional memory techniques.***

# Course Outline

**Part B: OpenMP/MPI/CUDA**

Week 1,2,3 : ***Shared Memory Programing Techniques:*** Introduction to OpenMP : Directives: *parallel, for, sections, task, master, single, critical, barrier, taskwait, atomic. Clauses: private, shared, firstprivate, lastprivate, reduction, nowait, ordered, schedule, collapse, num_threads, if(), threadprivate, copyin, copyprivate*

Week 4,5: ***Distributed Memory programming Techniques:*** MPI: Blocking, Non-blocking.

Week 6,7 : CUDA : OpenCL, Execution models, GPU memory, GPU libraries.

Week 10,11,: ***Introduction to accelerator programming using CUDA/OpenCL and Xeon-phi. Concepts of Heterogeneous programming techniques.***

**Practical:**

Implementation of parallel programs using OpenMP/MPI/CUDA.

**Assignment:** Performance evaluation of parallel algorithms (in group of 2 or 3 members)

# Index

1. Potential for increased computational speed
2. Embarrassingly Parallel Computations
3. Partitions
4. Divide and Conquer Strategies
5. **Pipelined Computations**
6. Load balancing and termination Detection

# 5.1 Pipelined Computations

- Functional decomposition

**Figure 5.1** Pipelined processes.

```
for (i = 0; i < n; i++)
   sum = sum + a[i];
```

$s_{out} = s_{in} + a[i];$

```
sum = sum + a[0];
sum = sum + a[1];
sum = sum + a[2];
sum = sum + a[3];
sum = sum + a[4];
      :
```

**Figure 5.2** Pipeline for an unrolled loop.

# 5.1 Pipelined Computations

- Functional decomposition

Given that the problem can be divided into a series of sequential tasks, the pipelined approach can provide increased speed under the following three types of computations:

1. If more than one instance of the complete problem is to be executed
2. If a series of data items must be processed, each requiring multiple operations
3. If information to start the next process can be passed forward before the process has completed all its internal operations.

- Let us identify these three solutions as Type 1, Type 2, Type 3.

# 5.1 Pipelined Computations

Type 1:

- Used in internal hardware design of computers
- Example simulation experiments with different parameters
- M instances can be computed on p processes in m+p-1 cycles

**Figure 5.4** Space-time diagram of a pipeline.

# 5.1 Pipelined Computations

Type 2:

- Series of data items need to be processed in a sequence, such as multiplying elements of an array

**Figure 5.6** Pipeline processing ten data elements.

# 5.1 Pipelined Computations

Type 3:

- Parallel programs where there is only one instance of the problem to execute, but each process can pass on information to the next process before it has completed.

**Figure 5.7** Pipeline processing where information passes to next stage before end of process.

# 5.1 Pipelined Computations

Partitioning process on to processors

**Figure 5.8** Partitioning processes onto processors.

# 5.2 Example and Analysis

## Adding List of numbers

```
recv(&accumulation, P_{i-1});
accumulation = accumulation + number;
send(&accumulation, P_{i+1});
```

except for the first process, $P_0$, which is

```
send(&number, P_1);
```

and the last process, $P_{p-1}$, which is

```
recv(&number, P_{p-2});
accumulation = accumulation + number;
```

Thus, an SPMD program could have the form

```
if (process > 0) {
   recv(&accumulation, P_{i-1});
   accumulation = accumulation + number;
}
if (process < p-1) send(&accumulation, P_{i+1});
```

**Figure 5.10** Pipelined addition.

# 5.2 Example and Analysis

Adding List of numbers : Pipeline design (a) ring configuration (b) direct access

**Figure 5.11** Pipelined addition numbers with a master process and ring configuration.

**Figure 5.12** Pipelined addition of numbers with direct access to slave processes.

# 5.2 Example and Analysis

## Adding List of numbers

For analyses of pipelines, it may not be appropriate to consider all the processes as having simultaneous communication and computation phases, as in previous chapters, because each instance of a problem starts at a different time and ends at a different time. Instead, we will assume that each process performs similar actions in each pipeline cycle. Then we will work out the computation and communication required in a pipeline cycle. The total execution time, $t_{total}$, is then obtained by using our pipeline formula for the number of cycles multiplied by the time of one cycle; that is,

$$t_{total} = (\text{time for one pipeline cycle})(\text{number of cycles})$$

$$t_{total} = (t_{comp} + t_{comm})(m + p - 1)$$

where there are $m$ instances of the problem and $p$ pipeline stages (processes). The computation and communication times are given by $t_{comp}$ and $t_{comm}$, respectively. The average time for a computation is given by

$$t_a = \frac{t_{total}}{m}$$

Let us assume that the structure shown in Figure 5.11 is used, and there are $n$ numbers.

# 5.2 Example and Analysis

## Adding List of numbers

***Single Instance of Problem.*** Let us first consider a single number being added in each stage; that is, $n$ is equal to $p$. The period of one pipeline cycle will be dictated by the time of one addition and two communications, one communication from the left and one communication to the right,

$$t_{comp} = 1$$

$$t_{comm} = 2(t_{startup} + t_{data})$$

and each pipeline cycle, $t_{cycle}$, requires at least $t_{comp} + t_{comm}$.

$$t_{cycle} = 2(t_{startup} + t_{data}) + 1$$

The term $t_{data}$ is the usual time to transfer one data word, and $t_{startup}$ is the communication startup time. The last process only has one communication, but this may not help because all the processes are allocated the same time period.

# 5.2 Example and Analysis

## Adding List of numbers

If we were to have only one set of numbers ($m = 1$), the total execution time, $t_{total}$, would be given by

$$t_{total} = (2(t_{startup} + t_{data}) + 1)n$$

(i.e., $n$ pipeline cycles, because each process must wait for the preceding process to complete its computation and pass on its results). The time complexity is $O(n)$.

***Multiple Instances of Problem.*** If, however, we have $m$ groups of $n$ numbers to add, each group resulting in a separate answer, the time of one pipeline cycle remains the same, but now we have $m + n - 1$ cycles, leading to

$$t_{total} = (2(t_{startup} + t_{data}) + 1)(m + n - 1)$$

For large $m$, the average execution time, $t_a$, is approximately

$$t_a = \frac{t_{total}}{m} \approx 2(t_{startup} + t_{data}) + 1$$

that is, one pipeline cycle.

# 5.2 Example and Analysis

## Adding List of numbers

***Data Partitioning with Multiple Instances of Problem.*** Now let us consider data partitioning with each stage processing a group of $d$ numbers. The number of processes is given by $p = n/d$. Each communication will still transfer one result, but the computation will also now require $d$ numbers to be accumulated ($d - 1$ steps) plus the incoming number, so that the following applies:

$$t_{comp} = d$$

$$t_{comm} = 2(t_{startup} + t_{data})$$

$$t_{total} = (2(t_{startup} + t_{data}) + d)(m + n/d - 1)$$

Obviously, as we increase $d$, the data partition, the impact of the communication on the overall time diminishes. But increasing the data partition decreases the parallelism and often increases the execution time. It is left as an exercise to explore the trade-offs of these effects (Problem 5-5).

# Reference

**Text Books and/or Reference Books:**


1. Professional CUDA C Programming – John Cheng, Max Grossman, Ty McKercher, 2014
2. **B.Wilkinson, M.Allen, ”Parallel Programming: Techniques and Applications Using Networked**

Workstations and Parallel Computers”, Pearson Education, 1999

3. I.Foster, ”Designing and building parallel programs”, 2003
4. Parallel Programming in C using OpenMP and MPI – Micheal J Quinn, 2004
5. Introduction to Parallel Programming – Peter S Pacheco, Morgan Kaufmann Publishers, 2011
6. Advanced Computer Architectures: A design approach, Dezso Sima, Terence Fountain, Peter Kacsuk, 2002
7. Parallel Computer Architecture : A hardware/Software Approach, David E Culler, Jaswinder Pal Singh Anoop Gupta, 2011
8. Introduction to Parallel Computing, Ananth Grama, Anshul Gupta, George Karypis, Vipin Kumar, Pearson, 2011

# Thank You

# Speedup Analysis

Dr. Geetha.V
Dept of IT
NITK, Surathkal

# Embarrasingly Parallel Computing Image Scaling - Speedup

# Image Scaling – Tcomp/Tcomm

# Partioning – Adding n data - Speedup

# Partioning – Adding n data – Tcomp/ TComm

# Divide and Conquer – add n data-Speedup

# Divide and Conquer – add n data-Tcomp/Tcomm

# Bucket Sort - Speedup

# Bucket Sort-Tcomp/Tcomm

**National Institute of Technology Karnataka Surathkal**
**Department of Information Technology**

# IT 301 Parallel Computing

## Shared Memory Programming Technique

## OpenMP :*parallel, for*

**Dr. Geetha V**

*Assistant Professor*

*Dept of Information Technology*

*NITK Surathkal*

# Index


- Introduction
  - Shared memory and Distributed Memory
  - OpenMP
- OpenMP
  - Program Structure
  - Directives : *parallel* , *for*
- References

# Course Outline

**Course Plan: Theory:**

**Part A: Parallel Computer Architectures**

Week 1,2,3: ***Introduction to Parallel Computer Architecture:*** Parallel Computing, Parallel architecture, bit level, instruction level , data level and task level parallelism. Instruction level parelllelism: pipelining(Data and control instructions), scalar and superscalar processors, vector processors. Parallel computers and computation.

Week 4,5: Memory Models: UMA, NUMA and COMA. Flynns classification, Cache coherence,

Week 6,7: Amdahl's Law. Performance evaluation, Designing parallel algorithms : Divide and conquer, Load balancing, Pipelining.

Week 8 -11: ***Parallel Programming techniques like Task Parallelism using TBB, TL2, Cilk++ etc. and software transactional memory techniques.***

# Course Outline

**Part B: OpenMP/MPI/CUDA**

Week 1,2,3 : ***Shared Memory Programing Techniques:*** **Introduction to OpenMP : Directives: *parallel, for,*** *sections, task, single, critical, barrier, taskwait, atomic. Clauses: private, shared, firstprivate, lastprivate, reduction, nowait, ordered, schedule, collapse, num_threads, shared, if().*

Week 4,5: ***Distributed Memory programming Techniques:*** MPI: Blocking, Non-blocking.

Week 6,7 : CUDA : OpenCL, Execution models, GPU memory, GPU libraries.

Week 10,11,: ***Introduction to accelerator programming using CUDA/OpenCL and Xeon-phi. Concepts of Heterogeneous programming techniques.***

**Practical:**

Implementation of parallel programs using OpenMP/MPI/CUDA.

**Assignment:** Performance evaluation of parallel algorithms (in group of 2 or 3 members)

# 1. Introduction

- Serial Programming
  - Develop a serial program and Optimize for performance
- Real World scenario:
  - Run multiple programs
  - Large and Complex problems
  - Time consuming
- Solution:
  - Use parallel machines
  - Use Multi-core Machines
- Why Parallel ?
  - Reduce the execution time
  - Run multiple programs

- What is parallel programming?
  - Obtain the same amount of computation with multiple cores or threads at low frequency (Fast)

# 1. Introduction

- Concurrency
  - Condition of a system in which multiple tasks are logically active at the same time ….but they may not necessarily run in parallel
- Parallelism
  - Subset of concurrency
  - Condition of a system in which multiple tasks are active at the same time and run in parallel

# 1. Introduction

- Shared Memory Machines
  - All processors share the same memory
  - The variables can be shared or private
  - Communication via shared memory

Multi-threading

  - Portable, easy to program and use
  - Not very scalable
- OpenMP based Programming

- Distributed Memory Machines
  - Each processor has its own memory
  - The variables are Independent
  - Communication by passing messages (network)

Multi-Processing

  - Difficult to program
  - Scalable
- MPI based Programming

# 1. OpenMP : API

## Open Specification for Multi-Processing (OpenMP)

- Library used to divide computational work in a program and add parallelism to serial program (create threads)
- An Application Program Interface (API) that is used explicitly direct multi-threaded, shared memory parallelism
- API Components
  - Compiler Directives
  - Runtime library routines
  - Environment variables
- Standardization
  - Jointly defined and endorsed by major computer hardware and software vendors

# 1. OpenMP : API

- Open Specification for Multi-Processing (OpenMP)
- History
  - In 1991, Parallel Computing Forum (PCF) group invented a set of directives for specifying loop parallelism in Fortran Programs
  - X3H5, an ANSI subcommittee developed an ANSI standard based on PCF
  - In 1997, the first version of OpenMP for Fortran was defined by OpenMP Architecture Review Board.
  - Binding for C/C++ was introduced later.
  - Version 3.0 is available since 2008.
  - Version 4.0 is available since 2013.
  - Current version is 5.0, released in November 2018

# 1. OpenMP : API

- Architecture

# 1. OpenMP : API

- OpenMP Language Extensions

# 1. OpenMP

## Threads and Process

- A process is an instance of a computer program that is being executed. It contains the program code and its current activity.
- A thread of execution is the smallest unit of processing that can be scheduled by an operating system.
- Differences between threads and processes:
  - A thread is contained inside a process.
  - Multiple threads can exist within the same process and share resources such as memory.
  - The threads of a process share the latter's instructions (code) and its context (values that its variables reference at any given moment).
  - Different processes do not share these resources.

http://en.wikipedia.org/wiki/Process_(computing)

# 1. OpenMP

## Process

- A process contains all the information needed to execute the program
  - Process ID
  - Program code
  - Data on run time stack
  - Global data
  - Data on heap

Each process has its own address space.

- In multitasking, processes are given time slices in a round robin fashion.
  - If computer resources are assigned to another process, the status of the present process must be saved, in order that the execution of the suspended process can be resumed later.

# 1. OpenMP

## Threads

- Thread model is an extension of the process model.
- Each process consists of multiple independent instruction streams (or threads) that are assigned computer resources by some scheduling procedure.
- Threads of a process share the address space of this process.
  - Global variables and all dynamically allocated data objects are accessible by all threads of a process
- Each thread has its own run-time stack, register, program counter.
- Threads can communicate by reading/writing variables in the common address space.

# 1. OpenMP

## OpenMP Programming Model

- Shared memory, thread-based parallelism
  - OpenMP is based on the existence of multiple threads in the shared memory programming paradigm.
  - A shared memory process consists of multiple threads.

- Explicit Parallelism
  - Programmer has full control over parallelization. OpenMP is not an automatic parallel programming model.

- Compiler directive based
  - Most OpenMP parallelism is specified using compiler directives which are embedded in the source code.

# 1. OpenMP

## OpenMP is not

- Necessarily implemented identically by all vendors
- Meant for distributed-memory parallel systems (it is designed for shared address spaced machines)
- Guaranteed to make the most efficient use of shared memory
- Required to check for data dependencies, data conflicts, race conditions, or deadlocks
- Required to check for code sequences
- Meant to cover compiler-generated automatic parallelization and directives to the compiler to assist such parallelization
- Designed to guarantee that input or output to the same file is synchronous when executed in parallel.

# 1. OpenMP

## FORK – JOIN Parallelism

- OpenMP program begin as a single process: the master thread. The master thread executes sequentially until the first parallel region construct is encountered.
- When a parallel region is encountered, master thread
  - Create a group of threads by FORK.
  - Becomes the master of this group of threads and is assigned the thread id 0 within the group.
- The statement in the program that are enclosed by the parallel region construct are then executed in parallel among these threads.
- JOIN: When the threads complete executing the statement in the parallel region construct, they synchronize and terminate, leaving only the master thread.

# 1. OpenMP

- I/O
  - OpenMP does not specify parallel I/O.
  - It is up to the programmer to ensure that I/O is conducted correctly within the context of a multithreaded program.

- Memory Model
  - Threads can “cache” their data and are not required to maintain exact consistency with real memory all the time.

  - When it is critical that all threads view a shared variable identically, the programmer is responsible for ensuring that the variable is updated by all threads as needed. *(flush)*

# 2. OpenMP Programming

```
%%Program hello.c
#include <stdio.h>
#include <omp.h>

int main(void)
{
    #pragma omp parallel
    printf("Hello, world.\n");
    return 0;
}
```

Compiling the program

```
$ gcc -fopenmp hello.c -o hello
```

Output

```
Hello, world.
Hello, world.
```

# 2. OpenMP Programming

- Directives
  - An OpenMP executable directive applies to the succeeding structured block or an OpenMP Construct. A “structured block” is a single statement or a compound statement with a single entry at the top and a single exit at the bottom.
- Clauses
  - Not all the clauses are valid on all directives. The set of clauses that is valid on a particular directive is described with the directive. Most of the clauses accept a comma-separated list of list items. All list items appearing in a clause must be visible.
- Runtime Library Routines
  - Execution environment routines affect and monitor threads, processors, and the parallel environment. Lock routines support synchronization with OpenMP locks. Timing routines support a portable wall clock timer. Prototypes for the runtime library routines are defined in the file “omp.h”.

# 2. OpenMP Programming : *Parallel*

## Directives

- Parallel
- For
- Sections
- Single
- Task
- Master
- Critical
- Barrier
- Taskwait
- Automic
- Flush
- Ordered
- Threadprivate

## Clauses

- Default (shared/none)
- Shared
- Private
- Firstprivate
- Lastprivate
- Reduction
- Copyin
- copyprivate

## Run time variables

- omp_set_num_threads
- omp_get_num_threads
- omp_get_max_threads
- omp_get_thread_num(void)
- ……etc

# 2. OpenMP Programming

- Directives
  - An OpenMP executable directive applies to the succeeding structured block or an OpenMP Construct. A “structured block” is a single statement or a compound statement with a single entry at the top and a single exit at the bottom.
  - They are case sensitive
  - Starts with **#pragma omp**
  - Directives cannot be embedded in embedded within continued statements, and statements cannot be embedded within directives.
  - Only one directive-name can be specified per directive

# 2. OpenMP Programming: Directives

The parallel construct forms a team of threads and starts parallel execution.

**#pragma omp parallel** *[clause[,]clause...] new-line*

*Structured-block*

Clause: **if**(*scalar-expression*)

**num_threads**(*integer-expression*)

**default**(*shared|none*)

**private**(*list*)

**firstprivate**(*list)*

**shared**(*list)*

**copyin**(*list)*

**reduction**(*operator:list)*

**Restrictions**

- A program which branches into or out of a parallel region is non-conforming.
- A program must not depend on any ordering of the evaluations of the clauses of the parallel directive, or on any side effects of the evaluations of the clauses.
- At most one ***if*** clause can appear on the directive.
- At most one ***num_threads*** clause can appear on the directive. The ***num_threads*** expression must evaluate to a positive integer value.

# 2. OpenMP Programming: Directives

**#pragma omp parallel** *[clause[,]clause...] new-line*

*Structured-block*

Clause: **if** (*scalar-expression*)

**num_threads**(*integer-expression*)

**default**(*shared|none*)

**private**(*list*)

**firstprivate**(*list*)

**shared**(*list*)

**copyin**(*list*)

**reduction**(*operator:list*)

- A team of threads is created to execute the parallel region
- A thead which encounters the parallel construct becomes master thread
- The thread id of master is 0
- All threads including master thread executes parallel region
- **omp_get_thread_num()** provides thread id
- There is implied barrier at the end of a parallel region.
- If a thread in a team executing a parallel region encounters another parallel directive, it creates a new team, and that thread is master of new team.

# 2. OpenMP Programming: Directives

**#pragma omp parallel** *[clause[,]clause…] new-line*

*Structured-block*

Clause: **if**(*scalar-expression*)

**num_threads**(*integer-expression*)

**default**(*shared|none*)

**private**(*list*)

**firstprivate**(*list*)

**shared**(*list*)

**copyin**(*list*)

**reduction**(*operator:list*)

- If execution of a thread terminates while inside a parallel region, execution of all threads in all teams terminates.
- The order of termination of threads is unspecified
- All the work done by a team prior to any barrier which the team has passed in the program is guaranteed to be complete.
- The amount of work done by each thread after the last barrier that it passed and before it terminates is unspecified

# 2. OpenMP Programming

```
%%Program hello.c
%%Num_threads() sets nuber of threads

#include <stdio.h>
#include <omp.h>

int main(void)
{
    #pragma omp parallel num_threads(4)
    printf("Hello, world.\n");
    return 0;
}
```

Compiling the program

```
$ gcc -fopenmp hello.c -o hello
```

Output

```
Hello, world.
Hello, world.
Hello, world.
Hello, world.
```

# 2. OpenMP Programming: Directives - Loop

```
#pragma omp parallel

#pragma omp parallel
{
  int threadnum = omp_get_thread_num(),
    numthreads = omp_get_num_threads();

  int low = N*threadnum/numthreads,
    high = N*(threadnum+1)/numthreads;

  for (i=low; i<high; i++)
    // do something with i
}
```

```
# pragma omp parallel for



#pragma omp parallel
#pragma omp for
for (i=0; i<N; i++) {
  // do something with i
}
```

# 2. OpenMP Programming: Directives

A loop Construct specifies that the iterations of loops will be distributed among and executed by the encountering team of threads.

**#pragma omp for** *[clause[,]clause...] new-line*

*for-loops*

Clause: **private**(*list*)

**firstprivate**(*list*)

**lastprivate**(*list*)

**reduction**(*operator:list*)

**schedule**(*kind[,chunk_size]*)

**collapse**(*n*)

***ordered***

***nowait***

Example

```
#pragma omp parallel
  #pragma omp for
  for (i=0; i<N; i++) {
    // do something with i
  }
```

# 2. OpenMP Programming

- *#pragma omp for* Restrictions
  - The values of the loop control expressions of the loop associated with the loop directive must be the same for all the threads in the team.
  - Only a single **schedule** clause can appear on a loop directive.
  - *chunk_size* must be a loop invariant integer expression with a positive value.
  - The value of the *chunk_size* expression must be the same for all threads in the team.
  - When **schedule(runtime)** is specified, chunk_size must not be specified.
  - Only a single **ordered** clause can appear on a loop directive.
  - The **ordered** clause must be present on the loop construct if any **ordered** region ever binds to a loop region arising from the loop construct.
  - The loop iteration variable may not appear in a **threadprivate** directive.
  - The *for-loop* must be a structured block, and in addition, its execution must not be terminated by a **break** statement.
  - The *for-loop* iteration variable var must have a signed integer type.
  - Only a single **nowait** clause can appear on a **for** directive.

# Index


- Introduction
  - Shared memory and Distributed Memory
  - OpenMP
- OpenMP
  - Program Structure
  - Directives : parallel, for
- References

# Reference

**Text Books and/or Reference Books:**

1. Professional CUDA C Programming – John Cheng, Max Grossman, Ty McKercher, 2014

2. B.Wilkinson, M.Allen, ”Parallel Programming: Techniques and Applications Using Networked Workstations and Parallel Computers”, Pearson Education, 1999

3. I.Foster, ”Designing and building parallel programs”, 2003

4. Parallel Programming in C using OpenMP and MPI – Micheal J Quinn, 2004

5. Introduction to Parallel Programming – Peter S Pacheco, Morgan Kaufmann Publishers, 2011

6. Advanced Computer Architectures: A design approach, Dezso Sima, Terence Fountain, Peter Kacsuk, 2002

7. Parallel Computer Architecture : A hardware/Software Approach, David E Culler, Jaswinder Pal Singh Anoop Gupta, 2011 8. Introduction to Parallel Computing, Ananth Grama, Anshul Gupta, George Karypis, Vipin Kumar, Pearson, 2011

# Reference

Acknowledgements

1. Introduction to OpenMP https://www3.nd.edu/~zxu2/acms60212-40212/Lec-12-OpenMP.pdf

2. Introduction to parallel programming for shared memory Machines https://www.youtube.com/watch?v=LL3TAHpxOig

3. OpenMP Application Program Interface Version 2.5 May 2005

4. OpenMP Application Program Interface Version 5.0 November 2018

# Thank You

**National Institute of Technology Karnataka Surathkal**
**Department of Information Technology**

# IT 301 Parallel Computing

## Shared Memory Programming Technique (2)

## OpenMP :*parallel, for Clauses*

**Dr. Geetha V**

*Assistant Professor*

*Dept of Information Technology*

*NITK Surathkal*

# Index

- OpenMP
  - Program Structure
  - Directives : *parallel* , *for*
  - Clauses
    - If
    - Private
    - Shared
    - Default(shared/none)
    - Firstprivate
    - Num_threads
- References

# Course Outline

**Course Plan: Theory:**

**Part A: Parallel Computer Architectures**

Week 1,2,3: ***Introduction to Parallel Computer Architecture:*** Parallel Computing, Parallel architecture, bit level, instruction level , data level and task level parallelism. Instruction level parelllelism: pipelining(Data and control instructions), scalar and superscalar processors, vector processors. Parallel computers and computation.

Week 4,5: Memory Models: UMA, NUMA and COMA. Flynns classification, Cache coherence,

Week 6,7: Amdahl's Law. Performance evaluation, Designing parallel algorithms : Divide and conquer, Load balancing, Pipelining.

Week 8 -11: ***Parallel Programming techniques like Task Parallelism using TBB, TL2, Cilk++ etc. and software transactional memory techniques.***

# Course Outline

**Part B: OpenMP/MPI/CUDA**

Week 1,2,3 : ***Shared Memory Programing Techniques:*** Introduction to OpenMP : Directives: *parallel, for, sections, task, single, critical, barrier, taskwait, atomic. Clauses: private, shared, firstprivate, lastprivate, reduction, nowait, ordered, schedule, collapse, num_threads, if().*

Week 4,5: ***Distributed Memory programming Techniques:*** MPI: Blocking, Non-blocking.

Week 6,7 : CUDA : OpenCL, Execution models, GPU memory, GPU libraries.

Week 10,11,: ***Introduction to accelerator programming using CUDA/OpenCL and Xeon-phi. Concepts of Heterogeneous programming techniques.***

**Practical:**

Implementation of parallel programs using OpenMP/MPI/CUDA.

**Assignment:** Performance evaluation of parallel algorithms (in group of 2 or 3 members)

# 1. Introduction

- Serial Programming
  - Develop a serial program and Optimize for performance
- Real World scenario:
  - Run multiple programs
  - Large and Complex problems
  - Time consuming
- Solution:
  - Use parallel machines
  - Use Multi-core Machines
- Why Parallel ?
  - Reduce the execution time
  - Run multiple programs

- What is parallel programming?
  - Obtain the same amount of computation with multiple cores or threads at low frequency (Fast)

# 1. Introduction

- Concurrency
  - Condition of a system in which multiple tasks are logically active at the same time ….but they may not necessarily run in parallel
- Parallelism
  - Subset of concurrency
  - Condition of a system in which multiple tasks are active at the same time and run in parallel

# 1. Introduction

- Shared Memory Machines
  - All processors share the same memory
  - The variables can be shared or private
  - Communication via shared memory

Multi-threading

  - Portable, easy to program and use
  - Not very scalable
- OpenMP based Programming

- Distributed Memory Machines
  - Each processor has its own memory
  - The variables are Independent
  - Communication by passing messages (network)

Multi-Processing

  - Difficult to program
  - Scalable
- MPI based Programming

# 1. OpenMP : API

## Open Specification for Multi-Processing (OpenMP)

- Library used to divide computational work in a program and add parallelism to serial program (create threads)
- An Application Program Interface (API) that is used explicitly direct multi-threaded, shared memory parallelism
- API Components
  - Compiler Directives
  - Runtime library routines
  - Environment variables
- Standardization
  - Jointly defined and endorsed by major computer hardware and software vendors

# 1. OpenMP

## FORK – JOIN Parallelism

- OpenMP program begin as a single process: the master thread. The master thread executes sequentially until the first parallel region construct is encountered.
- When a parallel region is encountered, master thread
  - Create a group of threads by FORK.
  - Becomes the master of this group of threads and is assigned the thread id 0 within the group.
- The statement in the program that are enclosed by the parallel region construct are then executed in parallel among these threads.
- JOIN: When the threads complete executing the statement in the parallel region construct, they synchronize and terminate, leaving only the master thread.

# 1. OpenMP

- I/O
  - OpenMP does not specify parallel I/O.
  - It is up to the programmer to ensure that I/O is conducted correctly within the context of a multithreaded program.

- Memory Model
  - Threads can “cache” their data and are not required to maintain exact consistency with real memory all the time.

  - When it is critical that all threads view a shared variable identically, the programmer is responsible for ensuring that the variable is updated by all threads as needed. *(flush)*

# 1. OpenMP : API

- Architecture

# 1. OpenMP : API

- OpenMP Language Extensions

# 2. OpenMP Programming

```
%%Program hello.c
#include <stdio.h>
#include <omp.h>

int main(void)
{
    #pragma omp parallel
    printf("Hello, world.\n");
    return 0;
}
```

Compiling the program

```
$ gcc -fopenmp hello.c -o hello
```

Output

```
Hello, world.
Hello, world.
```

# 2. OpenMP Programming

- Directives
  - An OpenMP executable directive applies to the succeeding structured block or an OpenMP Construct. A “structured block” is a single statement or a compound statement with a single entry at the top and a single exit at the bottom.

- Clauses
  - Not all the clauses are valid on all directives. The set of clauses that is valid on a particular directive is described with the directive. Most of the clauses accept a comma-separated list of list items. All list items appearing in a clause must be visible.

- Runtime Library Routines
  - Execution environment routines affect and monitor threads, processors, and the parallel environment. Lock routines support synchronization with OpenMP locks. Timing routines support a portable wall clock timer. Prototypes for the runtime library routines are defined in the file “omp.h”.

# 2. OpenMP Programming : *Parallel*

## Directives

- Parallel
- For
- Sections
- Single
- Task
- Master
- Critical
- Barrier
- Taskwait
- Automic
- Flush
- Ordered
- Threadprivate

## Clauses

- Default (shared/none)
- Shared
- Private
- Firstprivate
- Lastprivate
- Reduction
- Copyin
- copyprivate

## Run time variables

- omp_set_num_threads
- omp_get_num_threads
- omp_get_max_threads
- omp_get_thread_num(void)
- ……etc

# 2. OpenMP Programming

- Directives
  - An OpenMP executable directive applies to the succeeding structured block or an OpenMP Construct. A “structured block” is a single statement or a compound statement with a single entry at the top and a single exit at the bottom.
  - They are case sensitive
  - Starts with #pragma omp
  - Directives cannot be embedded in embedded within continued statements, and statements cannot be embedded within directives.
  - Only one directive-name can be specified per directive

# 2. OpenMP Programming: Directives

The parallel construct forms a team of threads and starts parallel execution.

**#pragma omp parallel** *[clause[,]clause...] new-line*

*Structured-block*

Clause: **if**(*scalar-expression*)

**num_threads**(*integer-expression*)

**default**(*shared|none*)

**private**(*list*)

**firstprivate**(*list)*

**shared**(*list)*

**copyin**(*list)*

**reduction**(*operator:list)*

Restrictions

- A program which branches into or out of a parallel region is non-conforming.
- A program must not depend on any ordering of the evaluations of the clauses of the parallel directive, or on any side effects of the evaluations of the clauses.
- At most one ***if*** clause can appear on the directive.
- At most one ***num_threads*** clause can appear on the directive. The ***num_threads*** expression must evaluate to a positive integer value.

# 2. OpenMP Programming: Directives

**#pragma omp parallel** *[clause[,]clause...] new-line*

*Structured-block*

Clause: **if** (*scalar-expression*)

**num_threads**(*integer-expression*)

**default**(*shared|none*)

**private**(*list*)

**firstprivate**(*list*)

**shared**(*list*)

**copyin**(*list*)

**reduction**(*operator:list*)

- A team of threads is created to execute the parallel region
- A thread which encounters the parallel construct becomes master thread
- The thread id of master is 0
- All threads including master thread executes parallel region
- **omp_get_thread_num()** provides thread id
- There is implied barrier at the end of a parallel region.
- If a thread in a team executing a parallel region encounters another parallel directive, it creates a new team, and that thread is master of new team.

# 2. OpenMP Programming: Directives

**#pragma omp parallel** *[clause[,]clause…] new-line*

*Structured-block*

Clause: **if**(*scalar-expression*)

**num_threads**(*integer-expression*)

**default**(*shared|none*)

**private**(*list*)

**firstprivate**(*list*)

**shared**(*list*)

**copyin**(*list*)

**reduction**(*operator:list*)

- If execution of a thread terminates while inside a parallel region, execution of all threads in all teams terminates.
- The order of termination of threads is unspecified
- All the work done by a team prior to any barrier which the team has passed in the program is guaranteed to be complete.
- The amount of work done by each thread after the last barrier that it passed and before it terminates is unspecified

# 2. OpenMP Programming: Clauses

**#pragma omp parallel** *[clause[,]clause…] new-line*

*Structured-block*

Clause: **if**(*scalar-expression*)

**num_threads**(*integer-expression*)

**default**(*shared|none*)

**private**(*list*)

**firstprivate**(*list*)

**shared**(*list*)

**copyin**(*list*)

**reduction**(*operator:list*)

If Clause

- A structured block is executed in parallel if the evaluation of ***if()*** clause is evaluated as true.
- A missing if clause is equivalent to an if clause that evaluates true.
- At most one ***if ()*** clause can appear on the directive.

# 2. OpenMP Programming : if clause

```
%%Program hello.c
%% if () clause

#include <stdio.h>
#include <omp.h>

int main(void)
{
int par=0;
 #pragma omp parallel  if(par==1)num_threads(4)
   printf("Hello, world.\n");
   return 0;
}
```

What is the result of the program?

Hello, world.

OR

Hello, world.
Hello, world.
Hello, world.
Hello, world.

# 2. OpenMP Programming: clauses num_threads

**#pragma omp parallel** *[clause[,]clause…] new-line*

*Structured-block*

Clause: **if**(*scalar-expression*)

**num_threads**(*integer-expression*)

**default**(*shared|none*)

**private**(*list*)

**firstprivate**(*list*)

**shared**(*list*)

**copyin**(*list*)

**reduction**(*operator:list*)

**num_threads** ( ) Clause

- **num_threads** must evaluate to positive integer.
- It sets number of threads for the execution of parallel region
- Some of the execution environment routines
  - To set number of threads

    **void omp_set_num_threads(int num_threads);**
  - To get number of threads

    **int omp_get_num_threads(void);**
  - To find number of threads which can for a team

    **int omp_get_max_threads(void);**
  - To get thread id

    **int omp_get_thread_num(void);**

# 2. OpenMP Programming : num_threads clause

```
%%Program hello.c
%% if () clause

#include <stdio.h>
#include <omp.h>

int main(void)
{
int par=0;
 #pragma omp parallel  if(par==1) num_threads(6)
   printf("Hello, world.\n");
   return 0;
}
```

What is the result of the program?

Hello, world.

OR

Hello, world.
Hello, world.
Hello, world.
Hello, world.
Hello, world.
Hello, world.

# 2. OpenMP Programming: num_threads

```
%%Program hello.c
%% if () clause

#include <stdio.h>
#include <omp.h>

int main(void)
{
int par=1;
 #pragma omp parallel  if(par==1) num_threads(6)
   printf("Hello, world.\n");
   return 0;
}
```

What is the result of the program?

Hello, world.

OR

Hello, world.
Hello, world.
Hello, world.
Hello, world.
Hello, world.
Hello, world.

# 2. OpenMP Programming: Clauses

**#pragma omp parallel** *[clause[,]clause…] new-line*

*Structured-block*

Clause: **if**(*scalar-expression*)

**num_threads**(*integer-expression*)

**default**(*shared|none*)

**private**(*list*)

**firstprivate**(*list*)

**shared**(*list*)

**copyin**(*list*)

**reduction**(*operator:list*)

**Default(shared/none) , shared(list) Clause**

- **default (shared)** clause causes all variables referenced in the construct which have implicitly determined sharing attributes to be shared.
- **default(none)** clause requires that each variable which is referenced in the construct, and that does not have a predetermined sharing attribute, must have its sharing attribute explicitly determined by being listed in a data sharing attribute clause
- **shared(list)** : One or more list items must be shared among all the threads in a team.

# 2. OpenMP Programming: Directives

**#pragma omp parallel** *[clause[,]clause…] new-line*

*Structured-block*

Clause: **if**(*scalar-expression*)

**num_threads**(*integer-expression*)

**default**(*shared|none*)

**private**(*list*)

**firstprivate**(*list*)

**shared**(*list*)

**copyin**(*list*)

**reduction**(*operator:list*)

**private (list) Clause**

- **private (list)** clause declares one or more list items must be private to a thread.
- A list item that appears in the **reduction** clause of a parallel construct must not appear in a **private** clause on a work-sharing construct.

# 2. OpenMP Programming: Clauses – default(shared)

```
// N: total number of iterations
#pragma omp parallel default(shared)
private(threadnum, numthreads, low,high, i)
{
  int threadnum = omp_get_thread_num(),
    numthreads = omp_get_num_threads();

  int low = N*threadnum/numthreads,
    high = N*(threadnum+1)/numthreads;

  for (i=low; i<high; i++)
    a[i]=b[i]+c[i]
    }
```

Shared : A, B, C

Other variables are default

# 2. OpenMP Programming: Directives – Default(none)

```
// N: total number of iterations
#pragma omp parallel default(none)
 shared(A,B,C,N) private(threadnum, numthreads,
low,high, i)
{
  int threadnum = omp_get_thread_num(),
    numthreads = omp_get_num_threads();

  int low = N*threadnum/numthreads,
    high = N*(threadnum+1)/numthreads;

  for (i=low; i<high; i++)
    a[i]=b[i]+c[i]
    }
```

Shared : A, B, C

Other variables are default

# 2. OpenMP Programming: Directives - shared

```c
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>

int main (void) {
int x=0;
#pragma omp parallel shared(x)
    {
    int tid=omp_get_thread_num();
    x=x+1;
    printf("Thread [%d] value of x is %d \n",tid,x);
    }


return 0;
}
```

```
Thread [1] value of x is 1
Thread [0] value of x is 2
Thread [2] value of x is 3
Thread [3] value of x is 4
```

```
Thread [0] value of x is 2
Thread [1] value of x is 1
Thread [2] value of x is 3
Thread [3] value of x is 4
```

Since the x is shared, the change in one thread is visible to all other threads too.

# 2. OpenMP Programming: Directives - private

```c
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>

int main (void) {
int x=0;
printf("x value ouside parallel:%d\n",x);
#pragma omp parallel private(x)
    {
    x=10;
    int tid=omp_get_thread_num();
    x=x+1;
    printf("Thread [%d] value of x is %d \n",tid,x);
    }


return 0;
}
```

```
x value ouside parallel:0
Thread [0] value of x is 11
Thread [2] value of x is 11
Thread [1] value of x is 11
Thread [3] value of x is 11
```

The variable x is private here . So the change of value performed in one thread is not visible to other threads.

# 2. OpenMP Programming: Directives - Private

```c
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>
int main (void) {
int x=10, tid;
printf("x value ouside parallel:%d\n",x);
#pragma omp parallel  num_threads(4) private(x) private(tid)
    {

    int tid=omp_get_thread_num();
    printf("\n 1. Thread [%d] value of x is %d \n",tid,x);
    x=15;
    printf("\n 2. Thread [%d] value of x is %d \n",tid,x);
    x=x+1;
    printf("\n 3. Thread [%d] value of x is %d \n",tid,x);
    }
return 0;
}
```

```
x value ouside parallel:10

 1. Thread [1] value of x is 6683504

 2. Thread [1] value of x is 15

 3. Thread [1] value of x is 16

 1. Thread [3] value of x is 17143736

 2. Thread [3] value of x is 15

 3. Thread [3] value of x is 16

 1. Thread [2] value of x is 6683600

 2. Thread [2] value of x is 15

 3. Thread [2] value of x is 16

 1. Thread [0] value of x is 0

 2. Thread [0] value of x is 15

 3. Thread [0] value of x is 16
```

The variable x is private here. The value of x is 10 before parallel region. Since x is private, the value 10 is not reflected in threads. When the parallel region is entered, the x contains some garbage value as it is not initialed .

# 2. OpenMP Programming: Directives - Shared

```c
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>
int main (void) {
int x=10, tid;
printf("x value ouside parallel:%d\n",x);
#pragma omp parallel  num_threads(4) shared(x) private(tid)
    {

    int tid=omp_get_thread_num();
    printf("\n 1. Thread [%d] value of x is %d \n",tid,x);
    x=15;
    printf("\n 2. Thread [%d] value of x is %d \n",tid,x);
    x=x+1;
    printf("\n 3. Thread [%d] value of x is %d \n",tid,x);
    }
return 0;
}
```

```
x value ouside parallel:10

 1. Thread [2] value of x is 10

 2. Thread [2] value of x is 15

 3. Thread [2] value of x is 16

 1. Thread [3] value of x is 10

 2. Thread [3] value of x is 15

 3. Thread [3] value of x is 16

 1. Thread [1] value of x is 10

 2. Thread [1] value of x is 15

 3. Thread [1] value of x is 16

 1. Thread [0] value of x is 10

 2. Thread [0] value of x is 15

 3. Thread [0] value of x is 16
```

X is shared. X is assigned value 15 inside parallel region.
Each thread is assigning value as 15. And updating it with x=x+1; But update is not getting reflected in other threads.

# 2. OpenMP Programming: Directives - Shared

```c
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>
int main (void) {
int x=10, tid;
printf("x value ouside parallel:%d\n",x);
#pragma omp parallel  num_threads(4) shared(x) private(tid)
    {

    int tid=omp_get_thread_num();
    printf("\n 1. Thread [%d] value of x is %d \n",tid,x);
    //x=15;

    printf("\n 2. Thread [%d] value of x is %d \n",tid,x);
    x=x+1;

    printf("\n 3. Thread [%d] value of x is %d \n",tid,x);
    }
return 0;
}
```

```
x value ouside parallel:10

 1. Thread [1] value of x is 10

 2. Thread [1] value of x is 10

 3. Thread [1] value of x is 11

 1. Thread [3] value of x is 10

 2. Thread [3] value of x is 11

 3. Thread [3] value of x is 12

 1. Thread [0] value of x is 10

 2. Thread [0] value of x is 12

 3. Thread [0] value of x is 13

 1. Thread [2] value of x is 10

 2. Thread [2] value of x is 13

 3. Thread [2] value of x is 14
```

x is shared. No assignment in the parallel region.
Update is getting reflected in all other threads.
Synchronization is job of programmer.

# 2. OpenMP Programming: Directives

**#pragma omp parallel** *[clause[,]clause…] new-line*

*Structured-block*

Clause: **if**(*scalar-expression*)

**num_threads**(*integer-expression*)

**default**(*shared|none*)

**private**(*list*)

**firstprivate**(*list*)

**shared**(*list*)

**copyin**(*list*)

**reduction**(*operator:list*)

**firstprivate (list) Clause**

- **firstprivate (list)** clause declares one or more list items to be private to a thread and initializes each of them with that the corresponding original item has when the construct is encountered.
- For a **firstprivate** clause on a **parallel** construct, the initial value of the new item is the value of the original list item that exists immediately prior to the **parallel** construct for the thread that encounters the construct.

# 2. OpenMP Programming: Directives - Private

```c
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>
int main (void) {
int x=10, tid;
printf("x value ouside parallel:%d\n",x);
#pragma omp parallel  num_threads(4) private(x) private(tid)
    {

    int tid=omp_get_thread_num();
    printf("\n 1. Thread [%d] value of x is %d \n",tid,x);
    x=15;
    printf("\n 2. Thread [%d] value of x is %d \n",tid,x);
    x=x+1;
    printf("\n 3. Thread [%d] value of x is %d \n",tid,x);
    }
return 0;
}
```

```
x value ouside parallel:10

 1. Thread [1] value of x is 6683504

 2. Thread [1] value of x is 15

 3. Thread [1] value of x is 16

 1. Thread [3] value of x is 17143736

 2. Thread [3] value of x is 15

 3. Thread [3] value of x is 16

 1. Thread [2] value of x is 6683600

 2. Thread [2] value of x is 15

 3. Thread [2] value of x is 16

 1. Thread [0] value of x is 0

 2. Thread [0] value of x is 15

 3. Thread [0] value of x is 16
```

X is private. While entering parallel region the x gets assigned with some value.

# 2. OpenMP Programming: Directives - Private

```c
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>
int main (void) {
int x=10, tid;
printf("x value ouside parallel:%d\n",x);
#pragma omp parallel  num_threads(4) private(x) private(tid)
    {

    int tid=omp_get_thread_num();
    printf("\n 1. Thread [%d] value of x is %d \n",tid,x);

    x=x+1;

    printf("\n 3. Thread [%d] value of x is %d \n",tid,x);
    }
return 0;
}
```

```
x value ouside parallel:10

 1. Thread [2] value of x is 6683600

 3. Thread [2] value of x is 6683601

 1. Thread [0] value of x is 0

 3. Thread [0] value of x is 1

 1. Thread [1] value of x is 6683504

 3. Thread [1] value of x is 6683505

 1. Thread [3] value of x is 14195704

 3. Thread [3] value of x is 14195705
```

Shared x. x is already with some garbage value. X is private to each thread.

# 2. OpenMP Programming: Directives - Private

```c
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>
int main (void) {
int x=10, tid;
printf("x value ouside parallel:%d\n",x);
#pragma omp parallel  num_threads(4) firstprivate(x) private(tid)
    {

    int tid=omp_get_thread_num();
    printf("\n 1. Thread [%d] value of x is %d \n",tid,x);

    x=x+1;

    printf("\n 3. Thread [%d] value of x is %d \n",tid,x);
    }
return 0;
}
```

```
x value ouside parallel:10

 1. Thread [2] value of x is 10

 3. Thread [2] value of x is 11

 1. Thread [1] value of x is 10

 3. Thread [1] value of x is 11

 1. Thread [3] value of x is 10

 3. Thread [3] value of x is 11

 1. Thread [0] value of x is 10

 3. Thread [0] value of x is 11
```

First private works as follows.

1. assign the value as in main thread before parallel region.
2. thread is private once it enters to parallel region.

# 2. OpenMP Programming: Directives

A loop Construct specifies that the iterations of loops will be distributed among and executed by the encountering team of threads.

**#pragma omp for** *[clause[,]clause...] new-line*

*for-loops*

Clause: **private**(*list*)

**firstprivate**(*list*)

**lastprivate**(*list*)

**reduction**(*operator:list*)

**schedule**(*kind[,chunk_size]*)

**collapse**(*n*)

***ordered***

***nowait***

Example

```
#pragma omp parallel
  #pragma omp for
  for (i=0; i<N; i++) {
    // do something with i
  }
```

# Index

- OpenMP
  - Program Structure
  - Directives : *parallel* , *for*
  - Clauses
    - If
    - Private
    - Shared
    - Default(shared/none)
    - Firstprivate
    - Num_threads

# Reference

**Text Books and/or Reference Books:**

1. Professional CUDA C Programming – John Cheng, Max Grossman, Ty McKercher, 2014

2. B.Wilkinson, M.Allen, ”Parallel Programming: Techniques and Applications Using Networked Workstations and Parallel Computers”, Pearson Education, 1999

3. I.Foster, ”Designing and building parallel programs”, 2003

4. Parallel Programming in C using OpenMP and MPI – Micheal J Quinn, 2004

5. Introduction to Parallel Programming – Peter S Pacheco, Morgan Kaufmann Publishers, 2011

6. Advanced Computer Architectures: A design approach, Dezso Sima, Terence Fountain, Peter Kacsuk, 2002

7. Parallel Computer Architecture : A hardware/Software Approach, David E Culler, Jaswinder Pal Singh Anoop Gupta, 2011 8. Introduction to Parallel Computing, Ananth Grama, Anshul Gupta, George Karypis, Vipin Kumar, Pearson, 2011

# Reference

Acknowledgements

1. Introduction to OpenMP https://www3.nd.edu/~zxu2/acms60212-40212/Lec-12-OpenMP.pdf

2. Introduction to parallel programming for shared memory Machines https://www.youtube.com/watch?v=LL3TAHpxOig

3. OpenMP Application Program Interface Version 2.5 May 2005

4. OpenMP Application Program Interface Version 5.0 November 2018

# Thank You

**National Institute of Technology Karnataka Surathkal**
**Department of Information Technology**

# IT 301 Parallel Computing

## Shared Memory Programming Technique (3)

## OpenMP :*parallel, for –reduction, master*

**Dr. Geetha V**

*Assistant Professor*

*Dept of Information Technology*

*NITK Surathkal*

# Index


- OpenMP
  - Program Structure
  - Directives : master
  - Clauses
    - Reduction
    - lastprivate
- References

# Course Outline

**Course Plan: Theory:**

**Part A: Parallel Computer Architectures**

Week 1,2,3: ***Introduction to Parallel Computer Architecture:*** Parallel Computing, Parallel architecture, bit level, instruction level , data level and task level parallelism. Instruction level parelllelism: pipelining(Data and control instructions), scalar and superscalar processors, vector processors. Parallel computers and computation.

Week 4,5: Memory Models: UMA, NUMA and COMA. Flynns classification, Cache coherence,

Week 6,7: Amdahl's Law. Performance evaluation, Designing parallel algorithms : Divide and conquer, Load balancing, Pipelining.

Week 8 -11: ***Parallel Programming techniques like Task Parallelism using TBB, TL2, Cilk++ etc. and software transactional memory techniques.***

# Course Outline

**Part B: OpenMP/MPI/CUDA**

Week 1,2,3 : ***Shared Memory Programing Techniques:*** Introduction to OpenMP : Directives: *parallel, for, sections, task, master, single, critical, barrier, taskwait, atomic. Clauses: private, shared, firstprivate, lastprivate, reduction, nowait, ordered, schedule, collapse, num_threads, if().*

Week 4,5: ***Distributed Memory programming Techniques:*** MPI: Blocking, Non-blocking.

Week 6,7 : CUDA : OpenCL, Execution models, GPU memory, GPU libraries.

Week 10,11,: ***Introduction to accelerator programming using CUDA/OpenCL and Xeon-phi. Concepts of Heterogeneous programming techniques.***

**Practical:**

Implementation of parallel programs using OpenMP/MPI/CUDA.

**Assignment:** Performance evaluation of parallel algorithms (in group of 2 or 3 members)

# 1. OpenMP

## FORK – JOIN Parallelism

- OpenMP program begin as a single process: the master thread. The master thread executes sequentially until the first parallel region construct is encountered.
- When a parallel region is encountered, master thread
  - Create a group of threads by FORK.
  - Becomes the master of this group of threads and is assigned the thread id 0 within the group.
- The statement in the program that are enclosed by the parallel region construct are then executed in parallel among these threads.
- JOIN: When the threads complete executing the statement in the parallel region construct, they synchronize and terminate, leaving only the master thread.

# 2. OpenMP Programming: Directives

**#pragma omp parallel** *[clause[,]clause…] new-line*

*Structured-block*

Clause: **if**(*scalar-expression*)

**num_threads**(*integer-expression*)

**default**(*shared|none*)

**private**(*list*)

**firstprivate**(*list)*

**shared**(*list)*

**copyin**(*list)*

**reduction**(*operator:list)*

# 2. OpenMP Programming: Clauses

**#pragma omp parallel** *[clause[,]clause…] new-line*

*Structured-block*

Clause: **if**(*scalar-expression*)

**num_threads**(*integer-expression*)

**default**(*shared|none*)

**private**(*list*)

**firstprivate**(*list*)

**shared**(*list*)

**copyin**(*list*)

**reduction**(*operator:list*)

**Default(shared/none) , shared(list) Clause**

- **if(scalar_expression):** if true execute in parallel
- **num_threads(int):** set number of threads
- **default (shared)** clause causes all variables referenced in the construct which have implicitly determined sharing attributes to be shared.
- **default(none)** clause requires that each variable which is referenced in the construct, and that does not have a predetermined sharing attribute, must have its sharing attribute explicitly determined by being listed in a data sharing attribute clause
- **shared(list)** : One or more list items must be shared among all the threads in a team.
- **private (list)** clause declares one or more list items must be private to a thread.
- **firstprivate (list)** clause declares one or more list items to be private to a thread and initializes each of them with that the corresponding original item has when the construct is encountered.

# 2. OpenMP Programming: Directives - Shared

```c
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>
int main (void) {
int x=10, tid;
printf("x value ouside parallel:%d\n",x);
#pragma omp parallel  num_threads(4) shared(x) private(tid)
    {

    int tid=omp_get_thread_num();
    printf("\n 1. Thread [%d] value of x is %d \n",tid,x);
    x=15;
    printf("\n 2. Thread [%d] value of x is %d \n",tid,x);
    x=x+1;
    printf("\n 3. Thread [%d] value of x is %d \n",tid,x);
    }
return 0;
}
```

```
x value ouside parallel:10

 1. Thread [2] value of x is 10

 2. Thread [2] value of x is 15

 3. Thread [2] value of x is 16

 1. Thread [3] value of x is 10

 2. Thread [3] value of x is 15

 3. Thread [3] value of x is 16

 1. Thread [1] value of x is 10

 2. Thread [1] value of x is 15

 3. Thread [1] value of x is 16

 1. Thread [0] value of x is 10

 2. Thread [0] value of x is 15

 3. Thread [0] value of x is 16
```

X is shared. X is assigned value 15 inside parallel region.
Each thread is assigning value as 15. And updating it with x=x+1; But update is not getting reflected in other threads.

# 2. OpenMP Programming: Directives - Shared

```c
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>
int main (void) {
int x=10, tid;
printf("x value ouside parallel:%d\n",x);
#pragma omp parallel  num_threads(4) shared(x) private(tid)
    {

    int tid=omp_get_thread_num();
    printf("\n 1. Thread [%d] value of x is %d \n",tid,x);
    //x=15;

    printf("\n 2. Thread [%d] value of x is %d \n",tid,x);
    x=x+1;

    printf("\n 3. Thread [%d] value of x is %d \n",tid,x);
    }
return 0;
}
```

```
x value ouside parallel:10

 1. Thread [1] value of x is 10

 2. Thread [1] value of x is 10

 3. Thread [1] value of x is 11

 1. Thread [3] value of x is 10

 2. Thread [3] value of x is 11

 3. Thread [3] value of x is 12

 1. Thread [0] value of x is 10

 2. Thread [0] value of x is 12

 3. Thread [0] value of x is 13

 1. Thread [2] value of x is 10

 2. Thread [2] value of x is 13

 3. Thread [2] value of x is 14
```

x is shared. No assignment in the parallel region. Update is getting reflected in all other threads. Synchronization is job of programmer.

# 2. OpenMP Programming: Directives - master

```c
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>
int main (void) {
int x=10, tid;
printf("x value ouside parallel:%d\n",x);
#pragma omp parallel  num_threads(4) shared(x) private(tid)
    {

    int tid=omp_get_thread_num();
    printf("\n 1. Thread [%d] value of x is %d \n",tid,x);
    #pragma omp master
    {
        x=15;
    }

    printf("\n 2. Thread [%d] value of x is %d \n",tid,x);
    x=x+1;

    printf("\n 3. Thread [%d] value of x is %d \n",tid,x);
    }
return 0;
}
```

```
x value ouside parallel:10

 1. Thread [0] value of x is 10

 2. Thread [0] value of x is 15

 3. Thread [0] value of x is 16

 1. Thread [3] value of x is 10

 2. Thread [3] value of x is 16

 3. Thread [3] value of x is 17

 1. Thread [1] value of x is 10

 2. Thread [1] value of x is 17

 3. Thread [1] value of x is 18

 1. Thread [2] value of x is 10

 2. Thread [2] value of x is 18

 3. Thread [2] value of x is 19
```

x is shared. Assignment statement is done only by master.

# 2. OpenMP Programming: Directives - master

The master construct specifies a structured block that is executed by the master thread of the team.

**#pragma omp master** *new-line*

*Structured block*

- A master region binds to the innermost enclosing parallel region.
- Only the master thread executes the structured block
- There is no implied barrier on entry or exit, for master construct. So other threads need not fork or join.

```
#pragma omp master
  {
      Structured block
}
```

Consider a program for adding sum of elements in an array.

```c
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>
int main (void) {
int tid,p,a[50],sum[20],dsum,finalsum;
int i, low, high;
int n=20;

//initialise
for(i=0;i<20;i++)
{
  a[i]=i;
  dsum=dsum+i;
}
printf("\n 1. dsum=%d \n",dsum);
```

```c
#pragma omp parallel  num_threads(4) default(shared) private(tid,low,high,i)
   {
   p=omp_get_num_threads();
   int tid=omp_get_thread_num();
   //assign the iterations to threads
   low=n*tid/p;
   high=n*(tid+1)/p;
   printf("\n 2. Thread [%d] low=%d high=%d \n",tid,low,high);

   //find partial sum
   sum[tid]=0;
   for(i=low;i<high;i++)
       sum[tid]=sum[tid]+a[i];

 printf("3: partial sum [%d] = %d \n",tid,sum[tid]);

} //close parallel
```

```c
finalsum=0;
  //add all partial sum
  for(i=0;i<p;i++)
    finalsum=finalsum+sum[i];

   printf("\n 4. finalsum= %d \n",finalsum);

   return 0;
}
```

The parallel region assigns iterations to each thread .
Partial sum is calculated in each thread
Partial sum is stored in an array
Master thread computes final sum

Consider a program for adding sum of elements in an array.

```c
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>
int main (void) {
int tid,p,a[50],sum[20],dsum,finalsum;
int i, low, high;
int n=20;

//initialise
for(i=0;i<20;i++)
{
  a[i]=i;
  dsum=dsum+i;
}
printf("\n 1. dsum=%d \n",dsum);
#pragma omp parallel  num_threads(4) default(shared) private(tid,low,high,i)
   {
   p=omp_get_num_threads();
   int tid=omp_get_thread_num();
   //assign the iterations to threads
   low=n*tid/p;
   high=n*(tid+1)/p;
   printf("\n 2. Thread [%d] low=%d high=%d \n",tid,low,high);

   //find partial sum
   sum[tid]=0;
   for(i=low;i<high;i++)
       sum[tid]=sum[tid]+a[i];

 printf("3: partial sum [%d] = %d \n",tid,sum[tid]);

} //close parallel
```

```c
finalsum=0;
   //add all partial sum
   for(i=0;i<p;i++)
       finalsum=finalsum+sum[i];

    printf("\n 4. finalsum= %d \n",finalsum);

    return 0;
}
```

```
 1. dsum=190

 2. Thread [2] low=10 high=15

 2. Thread [3] low=15 high=20
3: partial sum [3] = 85
3: partial sum [2] = 60

 2. Thread [1] low=5 high=10
3: partial sum [1] = 35

 2. Thread [0] low=0 high=5
3: partial sum [0] = 10

 4. finalsum= 190

--------------------------------
Process exited after 0.05116 seconds
```

# 2. OpenMP Programming: Clauses

**#pragma omp parallel** *[clause[,]clause…] new-line*

*Structured-block*

Clause: **if**(*scalar-expression*)

**num_threads**(*integer-expression*)

**default**(*shared|none*)

**private**(*list*)

**firstprivate**(*list*)

**shared**(*list*)

**copyin**(*list*)

**reduction**(*operator:list*)

- Reduction(operator: list)
- The **reduction** clause specifies an operator and one or more list items.
- For each list item, a private copy is created on each thread, and is initialized appropriately for the operator.
- After the end of the region, the original list item is updated with the values of the private copies using the specified operator.
- Initialization value depend on data type of the **reduction** variable.

# 2. OpenMP Programming: Clauses

**#pragma omp parallel** *[clause[,]clause...] new-line*

*Structured-block*

Clause: **if**(*scalar-expression*)

**num_threads**(*integer-expression*)

**default**(*shared|none*)

**private**(*list*)

**firstprivate**(*list*)

**shared**(*list*)

**copyin**(*list*)

**reduction**(*operator:list*)

- Reduction(operator: list)
- Initialization value depend on data type of the reduction variable.

| Operator | Initialization value |
|---|---|
| + | 0 |
| * | 1 |
| - | 0 |
| & | ~0 |
| \| | 0 |
| ^ | 0 |
| && | 1 |
| \|\| | 0 |

# Consider a program for adding sum of elements in an array: with reduction

```c
#include <stdlib.h>
#include <omp.h>
int main (void) {
int tid,p,a[50],sum=0,dsum,finalsum;
int i, low, high;
int n=20;

//initialise
for(i=0;i<20;i++)
{
  a[i]=i;
  dsum=dsum+i;
}
printf("\n 1. dsum=%d \n",dsum);
#pragma omp parallel  num_threads(4) default(shared) private(tid,low,high,i) reduction(+:sum)
    {
    p=omp_get_num_threads();
    int tid=omp_get_thread_num();
    //assign the iterations to threads
    low=n*tid/p;
    high=n*(tid+1)/p;
    printf("\n 2. Thread [%d] low=%d high=%d \n",tid,low,high);

    //find partial sum
    for(i=low;i<high;i++)
        sum=sum+a[i];
} //close parallel

  printf("\n 4. finalsum= %d \n",sum);
    return 0;
}
```

```
 1. dsum=190

 2. Thread [2] low=10 high=15

 2. Thread [3] low=15 high=20
3: partial sum [3] = 85
3: partial sum [2] = 60

 2. Thread [1] low=5 high=10
3: partial sum [1] = 35

 2. Thread [0] low=0 high=5
3: partial sum [0] = 10

 4. finalsum= 190
```

# 2. OpenMP Programming: Clauses

**#pragma omp parallel** *[clause[,]clause…] new-line*

*Structured-block*

Clause: **if**(*scalar-expression*)

**num_threads**(*integer-expression*)

**default**(*shared|none*)

**private**(*list*)

**firstprivate**(*list*)

**shared**(*list*)

**copyin**(*list*)

**reduction**(*operator:list*)

- Reduction(operator: list)
- Used for some form of recurrence calculations
- The type of a list item that appears in a reduction clause must be valid for the **reduction** operator.
- Aggregate types(including arrays), pointers types and reference types may not appear in a **reduction** clause
- A variable must appear in a **reduction** clause must not be const-qualified
- The operator specified in a **reduction** clause cannot be overloaded with respect to the variables that appear in that clause.

# 2. OpenMP Programming: Clauses

**#pragma omp for** *[clause[,]clause…] new-line*

*for-loops*

Clause: **private(*list*)**

**firstprivate(*list*)**

**lastprivate(*list*)**

**reduction(*operator:list*)**

**schedule(*kind[,chunk_size]*)**

**collapse(*n*)**

***ordered***

***nowait***

```c
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>
int main (void) {
int n=20, dsum=0, tid,i,a[20],sum=0;

for(i=0;i<n;i++)
{
a[i]=i;
dsum=dsum+i;
}
printf("dsum=%d\n",dsum);
#pragma omp parallel  num_threads(4)
    {
    int tid=omp_get_thread_num();
    #pragma omp for private(i) schedule(static, 5) reduction(+:sum)
    for(i=0;i<n;i++)
     sum=sum+a[i];
}
    printf("\n sum= %d \n",sum);
return 0;
}
```

# 2. OpenMP Programming: Clauses

```c
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>
int main (void) {
int n=20, dsum=0, tid,i,a[20],sum=0;

for(i=0;i<n;i++)
{
a[i]=i;
dsum=dsum+i;
}
printf("dsum=%d\n",dsum);
#pragma omp parallel  num_threads(4)
    {
    int tid=omp_get_thread_num();
    #pragma omp for private(i) schedule(static, 5) reduction(+:sum)
    for(i=0;i<n;i++)
     sum=sum+a[i];
}
    printf("\n sum= %d \n",sum);
return 0;
}
```

```
dsum=190

 sum= 190


--------------------------------
Process exited after 0.02466 seconds
Press any key to continue
```

# 2. OpenMP Programming: Clauses

```c
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>
int main (void) {
int n=10, dsum=0, tid,i,a[10],sum=0, count=0;

for(i=0;i<n;i++)
{
a[i]=i;
}
#pragma omp parallel  num_threads(2) default(shared)
    {
    int tid=omp_get_thread_num();
    count=0;
    #pragma omp for schedule(static, 5) private(count)
    for(i=0;i<n;i++)
     {
     a[i]=a[i]+5;
     if(a[i]%2==0) count++;
     printf("tid[%d] a[%d]=%d count %d\n",tid,i,a[i],count);
}
}
return 0;
}
```

```
tid[0] a[0]=5 count 4199876
tid[0] a[1]=6 count 4199877
tid[0] a[2]=7 count 4199877
tid[0] a[3]=8 count 4199878
tid[0] a[4]=9 count 4199878
tid[1] a[5]=10 count 1790492427
tid[1] a[6]=11 count 1790492427
tid[1] a[7]=12 count 1790492428
tid[1] a[8]=13 count 1790492428
tid[1] a[9]=14 count 1790492429
```

# 2. OpenMP Programming: Clauses

```c
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>
int main (void) {
int n=10, dsum=0, tid,i,a[10],sum=0, count=0;

for(i=0;i<n;i++)
{
a[i]=i;
}
#pragma omp parallel  num_threads(2) default(shared)
    {
    int tid=omp_get_thread_num();
    count=0;
    #pragma omp for schedule(static, 5) firstprivate(count)
    for(i=0;i<n;i++)
     {
     a[i]=a[i]+5;
     if(a[i]%2==0) count++;
     printf("tid[%d] a[%d]=%d count %d\n",tid,i,a[i],count);
}
}
return 0;
}
```

```
tid[0] a[0]=5 count 0
tid[0] a[1]=6 count 1
tid[0] a[2]=7 count 1
tid[0] a[3]=8 count 2
tid[0] a[4]=9 count 2
tid[1] a[5]=10 count 1
tid[1] a[6]=11 count 1
tid[1] a[7]=12 count 2
tid[1] a[8]=13 count 2
tid[1] a[9]=14 count 3
```

# 2. OpenMP Programming: Clauses

```c
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>
int main (void) {
int i;
int x;
x=100;
printf("X value before parallel region %d\n",x);
#pragma omp parallel for num_threads(2) private(x)
for(i=0;i<=10;i++){
x=x+i;
printf("Thread number: %d     x: %d\n",omp_get_thread_num(),x);
}
printf("x is %d\n", x);

return 0;
}
```

```
X value before parallel region 100
Thread number: 0     x: 8
Thread number: 0     x: 9
Thread number: 1     x: 2067342
Thread number: 1     x: 2067349
Thread number: 1     x: 2067357
Thread number: 1     x: 2067366
Thread number: 1     x: 2067376
Thread number: 0     x: 11
Thread number: 0     x: 14
Thread number: 0     x: 18
Thread number: 0     x: 23
x is 100
```

# 2. OpenMP Programming: Clauses

```c
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>
int main (void) {
int i;
int x;
x=100;
printf("X value before parallel region %d\n",x);
#pragma omp parallel for num_threads(2) firstprivate(x)
for(i=0;i<=10;i++){
x=x+i;
printf("Thread number: %d     x: %d\n",omp_get_thread_num(),x);
}
printf("x is %d\n", x);

return 0;
}
```

```
X value before parallel region 100
Thread number: 0     x: 100
Thread number: 0     x: 101
Thread number: 0     x: 103
Thread number: 0     x: 106
Thread number: 0     x: 110
Thread number: 0     x: 115
Thread number: 1     x: 106
Thread number: 1     x: 113
Thread number: 1     x: 121
Thread number: 1     x: 130
Thread number: 1     x: 140
x is 100
```

# 2. OpenMP Programming: Clauses

```c
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>
int main (void) {
int i;
int x;
x=100;
printf("X value before parallel region %d\n",x);
#pragma omp parallel for num_threads(2) firstprivate(x) lastprivate(x)
for(i=0;i<=10;i++){
x=x+i;
printf("Thread number: %d     x: %d\n",omp_get_thread_num(),x);
}
printf("x is %d\n", x);

return 0;
}
```

```
X value before parallel region 100
Thread number: 0     x: 100
Thread number: 1     x: 106
Thread number: 1     x: 113
Thread number: 0     x: 101
Thread number: 1     x: 121
Thread number: 1     x: 130
Thread number: 1     x: 140
Thread number: 0     x: 103
Thread number: 0     x: 106
Thread number: 0     x: 110
Thread number: 0     x: 115
x is 140
```

# Index


- OpenMP
  - Program Structure
  - Directives : master
  - Clauses
    - Reduction
    - Lastprivate
- References

# Reference

**Text Books and/or Reference Books:**

1. Professional CUDA C Programming – John Cheng, Max Grossman, Ty McKercher, 2014

2. B.Wilkinson, M.Allen, ”Parallel Programming: Techniques and Applications Using Networked Workstations and Parallel Computers”, Pearson Education, 1999

3. I.Foster, ”Designing and building parallel programs”, 2003

4. Parallel Programming in C using OpenMP and MPI – Micheal J Quinn, 2004

5. Introduction to Parallel Programming – Peter S Pacheco, Morgan Kaufmann Publishers, 2011

6. Advanced Computer Architectures: A design approach, Dezso Sima, Terence Fountain, Peter Kacsuk, 2002

7. Parallel Computer Architecture : A hardware/Software Approach, David E Culler, Jaswinder Pal Singh Anoop Gupta, 2011 8. Introduction to Parallel Computing, Ananth Grama, Anshul Gupta, George Karypis, Vipin Kumar, Pearson, 2011

# Reference

Acknowledgements

1. Introduction to OpenMP https://www3.nd.edu/~zxu2/acms60212-40212/Lec-12-OpenMP.pdf

2. Introduction to parallel programming for shared memory Machines https://www.youtube.com/watch?v=LL3TAHpxOig

3. OpenMP Application Program Interface Version 2.5 May 2005

4. OpenMP Application Program Interface Version 5.0 November 2018

# Thank You

**National Institute of Technology Karnataka Surathkal**
**Department of Information Technology**

# IT 301 Parallel Computing

## Shared Memory Programming Technique (4)

## OpenMP : *for –schedule*

**Dr. Geetha V**

*Assistant Professor*

*Dept of Information Technology*

*NITK Surathkal*

# Index

- OpenMP
  - Directives : if, for
  - Clauses
    - Schedule
      - Static
      - Dynamic
      - Guided
      - Runtime
- References

# Course Outline

**Course Plan: Theory:**

**Part A: Parallel Computer Architectures**

Week 1,2,3: ***Introduction to Parallel Computer Architecture:*** Parallel Computing, Parallel architecture, bit level, instruction level , data level and task level parallelism. Instruction level parelllelism: pipelining(Data and control instructions), scalar and superscalar processors, vector processors. Parallel computers and computation.

Week 4,5: Memory Models: UMA, NUMA and COMA. Flynns classification, Cache coherence,

Week 6,7: Amdahl's Law. Performance evaluation, Designing parallel algorithms : Divide and conquer, Load balancing, Pipelining.

Week 8 -11: ***Parallel Programming techniques like Task Parallelism using TBB, TL2, Cilk++ etc. and software transactional memory techniques.***

# Course Outline

**Part B: OpenMP/MPI/CUDA**

Week 1,2,3 : ***Shared Memory Programing Techniques:*** Introduction to OpenMP : Directives: *parallel, for, sections, task, master, single, critical, barrier, taskwait, atomic. Clauses: private, shared, firstprivate, lastprivate, reduction, nowait, ordered, schedule, collapse, num_threads, if().*

Week 4,5: ***Distributed Memory programming Techniques:*** MPI: Blocking, Non-blocking.

Week 6,7 : CUDA : OpenCL, Execution models, GPU memory, GPU libraries.

Week 10,11,: ***Introduction to accelerator programming using CUDA/OpenCL and Xeon-phi. Concepts of Heterogeneous programming techniques.***

**Practical:**

Implementation of parallel programs using OpenMP/MPI/CUDA.

**Assignment:** Performance evaluation of parallel algorithms (in group of 2 or 3 members)

# 1. OpenMP

## FORK – JOIN Parallelism

- OpenMP program begin as a single process: the master thread. The master thread executes sequentially until the first parallel region construct is encountered.
- When a parallel region is encountered, master thread
  - Create a group of threads by FORK.
  - Becomes the master of this group of threads and is assigned the thread id 0 within the group.
- The statement in the program that are enclosed by the parallel region construct are then executed in parallel among these threads.
- JOIN: When the threads complete executing the statement in the parallel region construct, they synchronize and terminate, leaving only the master thread.

# 2. OpenMP Programming: Directives : Parallel, For

**#pragma omp parallel** *[clause[,]clause…] new-line*

*Structured-block*

Clause: **if**(*scalar-expression*)

**num_threads**(*integer-expression*)

**default**(*shared|none*)

**private**(*list*)

**firstprivate**(*list*)

**shared**(*list*)

**copyin**(*list*)

**reduction**(*operator:list*)

**#pragma omp for** *[clause[,]clause…] new-line*

*for-loops*

Clause: **private**(*list*)

**firstprivate**(*list*)

**lastprivate**(*list*)

**reduction**(*operator:list*)

**schedule**(*kind[,chunk_size]*)

**collapse**(*n*)

***ordered***

***nowait***

# 2. OpenMP Programming: Clauses : Schedule

**#pragma omp for** *[clause[,]clause...] new-line*

*for-loops*

Clause: **private**(*list*)

**firstprivate**(*list*)

**lastprivate**(*list*)

**reduction**(*operator:list*)

**schedule**(*kind[,chunk_size]*)

**collapse**(*n*)

***ordered***

***nowait***

**Schedule(kind[,chunksize]) Clause**

- Schedule clause specifies how iteration of the loop are divided into contiguous non-empty subsets, called chunks, and how these chunks are assigned among threads of the team.
- Kind: It has following kind.
  - Static
  - Dynamic
  - Guided
  - runtime

# 2. OpenMP Programming: *schedule(static, chunk_size)*

**#pragma omp for** *[clause[,]clause...] new-line*

*for-loops*

Clause: **private**(*list*)

**firstprivate**(*list*)

**lastprivate**(*list*)

**reduction**(*operator:list*)

**schedule**(*kind[,chunk_size]*)

**collapse**(*n*)

***ordered***

***nowait***

**Schedule(static, chunksize]) Clause**

- Iterations are divided into chunk of size chunk_size.
- Chunks are statically assigned to threads in round robin fashion in the order of thread number
- Last chunk to be assigned may have smaller number of iterations.
- When no chunk size is specified, iterations/threads
- Example: 28 iteration, threads= 4
- Schedule(static, 5)

| thread0 | thread1 | thread2 | Thread3 | thread0 | thread1 |
|---|---|---|---|---|---|
| 0-4 | 5-9 | 10-14 | 15-19 | 20-24 | 25-27 |

# 2. OpenMP Programming: *schedule(dynamic, chunk_size)*

**#pragma omp for** *[clause[,]clause...] new-line*

*for-loops*

Clause: **private**(*list*)

**firstprivate**(*list*)

**lastprivate**(*list*)

**reduction**(*operator:list*)

**schedule**(*kind[,chunk_size]*)

**collapse**(*n*)

***ordered***

***nowait***

**Schedule(Dynamic, chunksize]) Clause**

- Iterations are assigned to threads in chunksize as the threads request them.
- Thread executes the chunk of iteration and then requests another chunk, until all iterations are complete.
- Each chunk contains chunksize except for the last chunk assigned.
- Example: 28 iteration, threads= 4
- Schedule(dynamic, 5)

| thread1 | thread3 | thread0 | thread2 | thread1 | thread2 |
|---|---|---|---|---|---|
| 0-4 | 5-9 | 10-14 | 15-19 | 20-24 | 25-27 |

# 2. OpenMP Programming: *schedule(guided, chunk_size)*

**Schedule(Guided, chunksize]) Clause**

- Iterations are assigned to threads of chunksize as the threads request them.
- Thread executes the chunk of iteration and then requests another chunk, until all iterations are complete.
- Chunk = remaining iterations / #threads
- Chunk size determines the minimum size of chunk , except lst chunk.
- Default value of chunk_size =1

- Chunk = remaining iterations / #threads
- Example: 28 iteration, threads= 4
- Schedule(guided,3)
- 28/4 = 7 [remaining = 28-7=21]
- 21/4=5.2 => 6 [remaining =21-6 = 15]
- 15/4=3.7=>4 [remaining = 15-4 = 11]
- 11/4=2.7 =>3 [ remaining =11-3=8]
- 8/4 = 2 [ min is chunk size 3 . So assign 3:]
- [remaining =8-3 = 5]
- 5/4 = 1 [ min = 3: remaining : 2]
- 2<=3 , so last chunk = 2

| thread2 | thread1 | thread0 | thread3 | thread2 | thread2 | thread2 |
|---|---|---|---|---|---|---|
| 0-6 (7) | 7-12(6) | 13-16(4) | 17-19(3) | 20-22 (3) | 23-25(3) | 26-27(2) |

# 2. OpenMP Programming: *schedule(runtime)*

**#pragma omp for** *[clause[,]clause...] new-line*

*for-loops*

Clause: **private**(*list*)

**firstprivate**(*list*)

**lastprivate**(*list*)

**reduction**(*operator:list*)

**schedule**(*kind[,chunk_size]*)

**collapse**(*n*)

***ordered***

***nowait***

**Schedule(runtime) Clause**

- The decision regarding scheduling is defered until run time, and the schedule and chunk size are taken from the *run-sched-var* control variable.

# 2. OpenMP Programming: *schedule(static, chunk_size)*

```c
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>
int main (void) {
int i;
#pragma omp parallel for num_threads(4) schedule(static,5)
for(i=0;i<28;i++){
printf("Thread number: %d : %d\n",omp_get_thread_num(),i);
}
return 0;
}
```

| thread0 | thread1 | thread2 | Thread3 | thread0 | thread1 |
|---|---|---|---|---|---|
| 0-4 | 5-9 | 10-14 | 15-19 | 20-24 | 25-27 |

```
Thread number: 1 : 5
Thread number: 1 : 6
Thread number: 1 : 7
Thread number: 1 : 8
Thread number: 1 : 9
Thread number: 1 : 25
Thread number: 1 : 26
Thread number: 1 : 27
Thread number: 0 : 0
Thread number: 0 : 1
Thread number: 0 : 2
Thread number: 0 : 3
Thread number: 0 : 4
Thread number: 0 : 20
Thread number: 0 : 21
Thread number: 0 : 22
Thread number: 0 : 23
Thread number: 0 : 24
Thread number: 2 : 10
Thread number: 2 : 11
Thread number: 2 : 12
Thread number: 2 : 13
Thread number: 2 : 14
Thread number: 3 : 15
Thread number: 3 : 16
Thread number: 3 : 17
Thread number: 3 : 18
Thread number: 3 : 19
```

# 2. OpenMP Programming: *schedule(dynamic, chunk_size)*

```c
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>
int main (void) {
int i;
#pragma omp parallel for num_threads(4) schedule(dynamic,5)
for(i=0;i<28;i++){
printf("Thread number: %d : %d\n",omp_get_thread_num(),i);
}
return 0;
}
```

| thread2 | thread1 | thread0 | thread3 | thread1 | thread3 |
|---|---|---|---|---|---|
| 0-4 | 5-9 | 10-14 | 15-19 | 20-24 | 25-27 |

```
Thread number: 2 : 0
Thread number: 1 : 5
Thread number: 1 : 6
Thread number: 1 : 7
Thread number: 1 : 8
Thread number: 1 : 9
Thread number: 3 : 15
Thread number: 3 : 16
Thread number: 3 : 17
Thread number: 3 : 18
Thread number: 3 : 19
Thread number: 3 : 25
Thread number: 3 : 26
Thread number: 3 : 27
Thread number: 0 : 10
Thread number: 0 : 11
Thread number: 0 : 12
Thread number: 0 : 13
Thread number: 0 : 14
Thread number: 1 : 20
Thread number: 1 : 21
Thread number: 1 : 22
Thread number: 1 : 23
Thread number: 1 : 24
Thread number: 2 : 1
Thread number: 2 : 2
Thread number: 2 : 3
Thread number: 2 : 4
```

# 2. OpenMP Programming: *schedule(guided, chunk_size)*

```c
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>
int main (void) {
int i;
#pragma omp parallel for num_threads(4) schedule(guided,3)
for(i=0;i<28;i++){
printf("Thread number: %d : %d\n",omp_get_thread_num(),i);
}
return 0;
}
```

```
Thread number: 2 : 0
Thread number: 2 : 1
Thread number: 2 : 2
Thread number: 2 : 3
Thread number: 2 : 4
Thread number: 2 : 5
Thread number: 2 : 6
Thread number: 2 : 20
Thread number: 2 : 21
Thread number: 2 : 22
Thread number: 2 : 23
Thread number: 2 : 24
Thread number: 2 : 25
Thread number: 2 : 26
Thread number: 2 : 27
Thread number: 3 : 17
Thread number: 3 : 18
Thread number: 3 : 19
Thread number: 1 : 7
Thread number: 1 : 8
Thread number: 1 : 9
Thread number: 1 : 10
Thread number: 1 : 11
Thread number: 1 : 12
Thread number: 0 : 13
Thread number: 0 : 14
Thread number: 0 : 15
Thread number: 0 : 16
```

| thread2 | thread1 | thread0 | thread3 | thread2 | thread2 | thread2 |
|---|---|---|---|---|---|---|
| 0-6 (7) | 7-12(6) | 13-16(4) | 17-19(3) | 20-22 (3) | 23-25(3) | 26-27(2) |

# 2. OpenMP Programming: *schedule(runtime)*

```c
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>
int main (void) {
int i;
#pragma omp parallel for num_threads(4) schedule(runtime)
for(i=0;i<28;i++){
printf("Thread number: %d : %d\n",omp_get_thread_num(),i);
}
return 0;
}
```

| thread1 | thread0 | thread3 | Thread2 | thread0 |
|---|---|---|---|---|
| 0 | 2 | 3 | 4-6 | 7-27 |

```
Thread number: 2 : 1
Thread number: 2 : 4
Thread number: 2 : 5
Thread number: 0 : 2
Thread number: 0 : 7
Thread number: 0 : 8
Thread number: 0 : 9
Thread number: 0 : 10
Thread number: 0 : 11
Thread number: 0 : 12
Thread number: 0 : 13
Thread number: 0 : 14
Thread number: 0 : 15
Thread number: 0 : 16
Thread number: 0 : 17
Thread number: 0 : 18
Thread number: 0 : 19
Thread number: 0 : 20
Thread number: 0 : 21
Thread number: 0 : 22
Thread number: 0 : 23
Thread number: 0 : 24
Thread number: 0 : 25
Thread number: 0 : 26
Thread number: 2 : 6
Thread number: 3 : 3
Thread number: 1 : 0
Thread number: 0 : 27
```

# 2. OpenMP Programming: *schedule(runtime)*

```c
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>
int main (void) {
int i;
#pragma omp parallel for num_threads(4) schedule(runtime)
for(i=0;i<28;i++){
printf("Thread number: %d : %d\n",omp_get_thread_num(),i);
}
return 0;
}
```

```
Thread number: 1 : 0
Thread number: 1 : 4
Thread number: 1 : 5
Thread number: 1 : 6
Thread number: 1 : 7
Thread number: 1 : 8
Thread number: 1 : 9
Thread number: 1 : 10
Thread number: 1 : 11
Thread number: 1 : 12
Thread number: 1 : 13
Thread number: 1 : 14
Thread number: 1 : 15
Thread number: 2 : 1
Thread number: 1 : 16
Thread number: 0 : 2
Thread number: 0 : 19
Thread number: 0 : 20
Thread number: 0 : 21
Thread number: 0 : 22
Thread number: 0 : 23
Thread number: 2 : 17
Thread number: 2 : 25
Thread number: 2 : 26
Thread number: 1 : 18
Thread number: 0 : 24
Thread number: 2 : 27
Thread number: 3 : 3
```

# Index

- OpenMP
  - Directives : if, for
  - Clauses
    - Schedule
      - Static
      - Dynamic
      - Guided
      - Runtime
- References

# Reference

**Text Books and/or Reference Books:**

1. Professional CUDA C Programming – John Cheng, Max Grossman, Ty McKercher, 2014

2. B.Wilkinson, M.Allen, ”Parallel Programming: Techniques and Applications Using Networked Workstations and Parallel Computers”, Pearson Education, 1999

3. I.Foster, ”Designing and building parallel programs”, 2003

4. Parallel Programming in C using OpenMP and MPI – Micheal J Quinn, 2004

5. Introduction to Parallel Programming – Peter S Pacheco, Morgan Kaufmann Publishers, 2011

6. Advanced Computer Architectures: A design approach, Dezso Sima, Terence Fountain, Peter Kacsuk, 2002

7. Parallel Computer Architecture : A hardware/Software Approach, David E Culler, Jaswinder Pal Singh Anoop Gupta, 2011 8. Introduction to Parallel Computing, Ananth Grama, Anshul Gupta, George Karypis, Vipin Kumar, Pearson, 2011

# Reference

Acknowledgements

1. Introduction to OpenMP https://www3.nd.edu/~zxu2/acms60212-40212/Lec-12-OpenMP.pdf

2. Introduction to parallel programming for shared memory Machines https://www.youtube.com/watch?v=LL3TAHpxOig

3. OpenMP Application Program Interface Version 2.5 May 2005

4. OpenMP Application Program Interface Version 5.0 November 2018

# Thank You

**National Institute of Technology Karnataka Surathkal**
**Department of Information Technology**

# IT 301 Parallel Computing

## Shared Memory Programming Technique (5)

## OpenMP : *Synchronization and other clauses*

**Dr. Geetha V**

*Assistant Professor*

*Dept of Information Technology*

*NITK Surathkal*

# Index

- OpenMP
  - Directives : if, for
  - Clauses
    - Master
    - Single
    - Barrier
    - Atomic
    - critical
    - Nowait
    - Ordered
- References

# Course Outline

**Course Plan: Theory:**

**Part A: Parallel Computer Architectures**

Week 1,2,3: ***Introduction to Parallel Computer Architecture:*** Parallel Computing, Parallel architecture, bit level, instruction level , data level and task level parallelism. Instruction level parelllelism: pipelining(Data and control instructions), scalar and superscalar processors, vector processors. Parallel computers and computation.

Week 4,5: Memory Models: UMA, NUMA and COMA. Flynns classification, Cache coherence,

Week 6,7: Amdahl's Law. Performance evaluation, Designing parallel algorithms : Divide and conquer, Load balancing, Pipelining.

Week 8 -11: ***Parallel Programming techniques like Task Parallelism using TBB, TL2, Cilk++ etc. and software transactional memory techniques.***

# Course Outline

**Part B: OpenMP/MPI/CUDA**

Week 1,2,3 : ***Shared Memory Programing Techniques:*** Introduction to OpenMP : Directives: *parallel, for, sections, task, master, single, critical, barrier, taskwait, atomic*. *Clauses: private, shared, firstprivate, lastprivate, reduction, nowait, ordered, schedule, collapse, num_threads, if(), threadprivate, copyin, copyprivate*

Week 4,5: ***Distributed Memory programming Techniques:*** MPI: Blocking, Non-blocking.

Week 6,7 : CUDA : OpenCL, Execution models, GPU memory, GPU libraries.

Week 10,11,: ***Introduction to accelerator programming using CUDA/OpenCL and Xeon-phi. Concepts of Heterogeneous programming techniques.***

**Practical:**

Implementation of parallel programs using OpenMP/MPI/CUDA.

**Assignment:** Performance evaluation of parallel algorithms (in group of 2 or 3 members)

# 1. OpenMP

## FORK – JOIN Parallelism

- OpenMP program begin as a single process: the master thread. The master thread executes sequentially until the first parallel region construct is encountered.
- When a parallel region is encountered, master thread
  - Create a group of threads by FORK.
  - Becomes the master of this group of threads and is assigned the thread id 0 within the group.
- The statement in the program that are enclosed by the parallel region construct are then executed in parallel among these threads.
- JOIN: When the threads complete executing the statement in the parallel region construct, they synchronize and terminate, leaving only the master thread.

# 2. OpenMP Programming: Directives : Parallel, For

**#pragma omp parallel** *[clause[,]clause...] new-line*

*Structured-block*

Clause: **if**(*scalar-expression*)

**num_threads**(*integer-expression*)

**default**(*shared|none*)

**private**(*list*)

**firstprivate**(*list*)

**shared**(*list*)

**copyin**(*list*)

**reduction**(*operator:list*)

**#pragma omp for** *[clause[,]clause...] new-line*

*for-loops*

Clause: **private**(*list*)

**firstprivate**(*list*)

**lastprivate**(*list*)

**reduction**(*operator:list*)

**schedule**(*kind[,chunk_size]*)

**collapse**(*n*)

***ordered***

***nowait***

# 2. OpenMP Programming: Clauses : master

```
#pragma omp master
        Structured block
```

**Master :**

- It specifies a structured block that is executed by the master thread of the team
- Other threads in the team do not execute the associated structured block.
- There is no implied barrier either on entry to, or exit from, the master construct.

# 2. OpenMP Programming: Clauses : single

```
#pragma omp single [clause [[,] clause...]
Structured block
Clause:
    Private(list)
    Firstprivate(list)
    Copyprivate(list)
    nowait
```

**Single :**

- It specifies that the associated structured block is executed by only one thread in the team (not necessarily the master thread).
- The other threads in the team do not execute the block, and wait at an implicit barrier at the end of single construct, unless nowait clause is specified
- *nowait* : The other threads need not wait for synchronization point.
- *Copyprivate* clause must not be used with the *nowait* clause

# 2. OpenMP Programming: Clauses : critical

```
#pragma omp critical [(name)] new-line
Structured block
```

**critical**

- The critical construct restricts execution of the associated structured block to a single thread at a time.
- An optional name may be used to identify the critical construct. All critical constructs without a name are considered to have the same unspecified name.
- A thread waits at the beginning of a critical region until no other thread is executing a critical region with the same name.
- The critical construct enforces exclusive access with respect to all critical constructs with the same name in all threads, not just in the current team.

# 2. OpenMP Programming: Construct: barrier

```
#pragma omp barrier new-line
```

**barrier**

- The barrier construct specifies an explicit barrier at the point at which the construct appears.
- The barrier directive may only be placed in the program at a position where ignoring or deleting the directive would result in a program with correct syntax.
- All of the threads of the team executing the binding parallel region must execute the barrier region before any are allowed to continue execution beyond the barrier.
- Each barrier region must be encountered by all threads in a team or by none at all.
- The sequence of work-sharing regions and barrier regions encountered must be the same for every thread in a team.

# 2. OpenMP Programming: Construct: atomic

```
#pragma omp atomic new-line
Expression-stmt
```

**atomic**

- The atomic construct ensures that a specific storage location is updated atomically, rather than exposing it to the possibility of multiple, simultaneous writing threads.
- Expression-stmt is an expression statement with one of the following forms:
  - x binop=expr
  - x++
  - ++x
  - x--
  - --x
- Binop is not an overloaded operator and is one of +,*,-,/,&,^,|,<<, or >>.
- Only the load and store of the object designated by x are atomic; the evaluation of expr is not atomic.

# 2. OpenMP Programming: Construct: flush

```
#pragma omp flush [(list)] new-line
```

**flush**

- The flush construct executes the OpenMP flush operation.
- This operation makes a thread's temporary view of memory consistent with memory and enforces an order on the memory operations of the variables explicitly specified or implied.
- The flush construct with a list applies the flush operation to the items in the list, and does not return until the operation is complete for all specified list items.
- A flush construct without a list, executed on a given thread, operates as if the whole thread-visible data state of the program, as defined by the base language, is flushed.
- If a pointer is present in the list, the pointer itself is flushed, not the object to which the pointer refers.

# 2. OpenMP Programming: Construct: flush

```
#pragma omp flush [(list)] new-line
```

**flush**

A flush region without a list is implied at the following locations:

- During a barrier region
- At entry to and exit from parallel, critical and ordered regions.
- At exit from work-sharing regions, unless a *nowait* is present.
- At entry to and exit from combined parallel work-sharing regions
- During set and unset lock, test, nest lock etc.

A flush region with a list is implied at the following locations:

At entry and exit from atomic regions, where the list contains only the object updated in the atomic construct.

# 2. OpenMP Programming: Construct: ordered

```
#pragma omp ordered new-line
Structured block
```

**Ordered**

The ordered construct specifies a structured block in a loop region which will be executed in the order of the loop iterations.

This sequentializes and orders the code within an ordered region while allowing code outside the region to run in parallel.

When a thread executing other than first iteration, encounters an ordered region, it waits at the beginning of that ordered region until each of the previous iterations that contains an ordered region has completed the ordered region.

# 2. OpenMP Programming: Construct: ordered

```
#pragma omp ordered new-line
Structured block
```

**Ordered**

Restrictions to the ordered construct:

- The loop region to which an ordered region binds must have an ordered clause specified on the corresponding loop (or parallel loop) construct.
- During execution of an iteration of a loop within a loop region, the executing thread must not execute more than one ordered region which binds to the same loop region.

# 2. OpenMP Programming: Clause: nowait

nowait

**nowait**

It there are multiple independent loops within a parallel region, then *nowait* can be used to avoid the implied barrier at the end of the loop construct.

# 3. OpenMP Programming: Examples

```c
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>
int main (void) {
int x=10, tid;
printf("x value ouside parallel:%d\n",x);
#pragma omp parallel  num_threads(4) shared(x) private(tid)
    {

    int tid=omp_get_thread_num();
    #pragma omp master
    {
        x=15;
        printf("\n 1. Thread [%d] value of x is %d \n",tid,x);
    }
    x=x+1;

    printf("\n 2. Thread [%d] value of x is %d \n",tid,x);
    }
return 0;
}
```

```
x value ouside parallel:10

 2. Thread [2] value of x is 11

 2. Thread [3] value of x is 16

 1. Thread [0] value of x is 15

 2. Thread [0] value of x is 17

 2. Thread [1] value of x is 12
```

# 3. OpenMP Programming: Examples

```c
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>
int main (void) {
int x=10, tid;
printf("x value ouside parallel:%d\n",x);
#pragma omp parallel  num_threads(4) shared(x) private(tid)
    {

    int tid=omp_get_thread_num();
    #pragma omp single
    {
        x=15;
        printf("\n 1. Thread [%d] value of x is %d \n",tid,x);
    }
    x=x+1;

    printf("\n 2. Thread [%d] value of x is %d \n",tid,x);
    }
return 0;
}
```

```
x value ouside parallel:10

 1. Thread [1] value of x is 15

 2. Thread [0] value of x is 16

 2. Thread [2] value of x is 17

 2. Thread [3] value of x is 18

 2. Thread [1] value of x is 19
```

```
x value ouside parallel:10

 1. Thread [2] value of x is 15

 2. Thread [1] value of x is 16

 2. Thread [0] value of x is 17

 2. Thread [3] value of x is 18

 2. Thread [2] value of x is 19
```

# 3. OpenMP Programming: Examples

```c
int main (void) {
int a[5], i;
    #pragma omp parallel
    {
        #pragma omp for
        for (i = 0; i < 5; i++)
            a[i] = i * i;

        #pragma omp master
        {
            printf("tid of master %d\n",omp_get_thread_num());
            for (i = 0; i < 5; i++)
                printf("a[%d] = %d\n", i, a[i]);
        }
        // Wait.
        //#pragma omp barrier
        printf("tid %d \n",omp_get_thread_num());
           // Continue with the computation.
        #pragma omp for
        for (i=0;i<5;i++)
            a[i]+=i;

         #pragma omp single
        {
            printf("tid of single %d\n",omp_get_thread_num());
            for (i = 0; i < 5; i++)
                printf("a[%d] = %d\n", i, a[i]);
        }
    }
return 0;
}
```

```
tid 2
tid 1
tid of master 0
a[0] = 0
a[1] = 1
a[2] = 6
a[3] = 12
a[4] = 16
tid 0
tid 3
tid of single 2
a[0] = 0
a[1] = 2
a[2] = 6
a[3] = 12
a[4] = 20
```

# 3. OpenMP Programming: Examples

```c
int main (void) {
int a[5], i;
    #pragma omp parallel
    {
        #pragma omp for
        for (i = 0; i < 5; i++)
            a[i] = i * i;

        #pragma omp master
        {
            printf("tid of master %d\n",omp_get_thread_num());
            for (i = 0; i < 5; i++)
                printf("a[%d] = %d\n", i, a[i]);
        }
        // Wait.
        #pragma omp barrier
        printf("tid %d \n",omp_get_thread_num());
           // Continue with the computation.
        #pragma omp for
        for (i=0;i<5;i++)
            a[i]+=i;

         #pragma omp single
        {
            printf("tid of single %d\n",omp_get_thread_num());
            for (i = 0; i < 5; i++)
                printf("a[%d] = %d\n", i, a[i]);
        }
    }
return 0;
}
```

```
tid of master 0
a[0] = 0
a[1] = 1
a[2] = 4
a[3] = 9
a[4] = 16
tid 1
tid 3
tid 2
tid 0
tid of single 1
a[0] = 0
a[1] = 2
a[2] = 6
a[3] = 12
a[4] = 20
```

# 3. OpenMP Programming: Examples

```c
#include<stdio.h>
#include<omp.h>
int main(void)
{
int i,n,a[50],b[50],sum;
double t1,t2;
printf("Enter the value of n");
scanf("%d",&n);
t1=omp_get_wtime();
#pragma omp parallel num_threads(4)
{
int id=omp_get_thread_num();
#pragma omp for ordered reduction(+:sum)
for(i=0;i<n;i++)
{
printf("Thread %d: value of i : %d\n",id,i);
sum=sum+i;
#pragma omp ordered
{
b[i]=i+1;
printf("b[%d] value is %d in ORDER\n",i,b[i]);
}
}
}
t2=omp_get_wtime();
printf("Time taken is %f",t2-t1);
return 0;
}
```

```
Enter the value of n5
Thread 3: value of i : 4
Thread 1: value of i : 2
Thread 2: value of i : 3
Thread 0: value of i : 0
b[0] value is 1 in ORDER
Thread 0: value of i : 1
b[1] value is 2 in ORDER
b[2] value is 3 in ORDER
b[3] value is 4 in ORDER
b[4] value is 5 in ORDER
Time taken is 0.001000
```

# Index

- OpenMP
  - Directives : if, for
  - Clauses
    - Master
    - Single
    - Barrier
    - Atomic
    - critical
    - Nowait
    - Ordered
- References

# Reference

**Text Books and/or Reference Books:**

1. Professional CUDA C Programming – John Cheng, Max Grossman, Ty McKercher, 2014

2. B.Wilkinson, M.Allen, ”Parallel Programming: Techniques and Applications Using Networked Workstations and Parallel Computers”, Pearson Education, 1999

3. I.Foster, ”Designing and building parallel programs”, 2003

4. Parallel Programming in C using OpenMP and MPI – Micheal J Quinn, 2004

5. Introduction to Parallel Programming – Peter S Pacheco, Morgan Kaufmann Publishers, 2011

6. Advanced Computer Architectures: A design approach, Dezso Sima, Terence Fountain, Peter Kacsuk, 2002

7. Parallel Computer Architecture : A hardware/Software Approach, David E Culler, Jaswinder Pal Singh Anoop Gupta, 2011 8. Introduction to Parallel Computing, Ananth Grama, Anshul Gupta, George Karypis, Vipin Kumar, Pearson, 2011

# Reference

Acknowledgements

1. Introduction to OpenMP https://www3.nd.edu/~zxu2/acms60212-40212/Lec-12-OpenMP.pdf

2. Introduction to parallel programming for shared memory Machines https://www.youtube.com/watch?v=LL3TAHpxOig

3. OpenMP Application Program Interface Version 2.5 May 2005

4. OpenMP Application Program Interface Version 5.0 November 2018

# Thank You

**National Institute of Technology Karnataka Surathkal**
**Department of Information Technology**

# IT 301 Parallel Computing

## Shared Memory Programming Technique (6)

## OpenMP : *sections, threadprivate, collapse*

**Dr. Geetha V**

*Assistant Professor*

*Dept of Information Technology*

*NITK Surathkal*

# Index

- OpenMP
  - Directives : if, for, master, single, Barrier, atomic, critical,
  - Directives
    - Sections
  - Clauses
    - Threadprivate
    - Collapse
    - Threadwait
    - Copyin, copyprivate
- References

# Course Outline

**Course Plan: Theory:**

**Part A: Parallel Computer Architectures**

Week 1,2,3: ***Introduction to Parallel Computer Architecture:*** Parallel Computing, Parallel architecture, bit level, instruction level , data level and task level parallelism. Instruction level parelllelism: pipelining(Data and control instructions), scalar and superscalar processors, vector processors. Parallel computers and computation.

Week 4,5: Memory Models: UMA, NUMA and COMA. Flynns classification, Cache coherence,

Week 6,7: Amdahl's Law. Performance evaluation, Designing parallel algorithms : Divide and conquer, Load balancing, Pipelining.

Week 8 -11: ***Parallel Programming techniques like Task Parallelism using TBB, TL2, Cilk++ etc. and software transactional memory techniques.***

# Course Outline

**Part B: OpenMP/MPI/CUDA**

Week 1,2,3 : ***Shared Memory Programing Techniques:*** Introduction to OpenMP : Directives: *parallel, for, sections, task, master, single, critical, barrier, taskwait, atomic. Clauses: private, shared, firstprivate, lastprivate, reduction, nowait, ordered, schedule, collapse, num_threads, if(), threadprivate, copyin, copyprivate*

Week 4,5: ***Distributed Memory programming Techniques:*** MPI: Blocking, Non-blocking.

Week 6,7 : CUDA : OpenCL, Execution models, GPU memory, GPU libraries.

Week 10,11,: ***Introduction to accelerator programming using CUDA/OpenCL and Xeon-phi. Concepts of Heterogeneous programming techniques.***

**Practical:**

Implementation of parallel programs using OpenMP/MPI/CUDA.

**Assignment:** Performance evaluation of parallel algorithms (in group of 2 or 3 members)

# 1. OpenMP

## FORK – JOIN Parallelism

- OpenMP program begin as a single process: the master thread. The master thread executes sequentially until the first parallel region construct is encountered.
- When a parallel region is encountered, master thread
  - Create a group of threads by FORK.
  - Becomes the master of this group of threads and is assigned the thread id 0 within the group.
- The statement in the program that are enclosed by the parallel region construct are then executed in parallel among these threads.
- JOIN: When the threads complete executing the statement in the parallel region construct, they synchronize and terminate, leaving only the master thread.

# 2. OpenMP Programming: Worksharing

Work sharing constructs

- Loop constructs
- Section construct
- Single construct

# 2. OpenMP Programming: Section

```
#pragma omp sections [clause,….]
{
 [#pragma omp section new-line
   Structured block]
 [#pragma omp section new-line
  Structured-block]
```

**Clauses**

Private(list)

Firstprivate(list)

Lastprivate(list)

Reduction(operator:list)

nowait

**Section :**

- It is a non-iterative work-sharing construct that contains a set of structured blocks that are to be divided among, and executed by, the threads in a team.
- Each structured block is executed by one of the threads in the team.
- There is an implicit barrier at the end of sections construct, unless a *nowait* clause is specified.
- Only a single *nowait* clause can appear on a sections directive.

# 2. OpenMP Programming: Collapse

```
#pragma omp for schedule(static, n)
collapse(2)
for(i=0; i<imax; i++) {
  for(j=0; j<jmax;j ++)
      a[i][j] = b[i][j] + c[i][j]
}
```

**Collapse :**

- It increases the total number of iterations that will be partitioned across the available number of OMP threads by reducing the granularity of work to be done by each thread.
- If the amount of work to be done by each thread is non-trivial (after collapsing is applied), this may improve the parallel scalability of the OMP applications.

# 2. OpenMP Programming: threadprivate

```
#pragma omp threadprivate(list)
```

**Threadprivate**

- Each thread is allowed to have its own temporary view of the shared memory.
- Each thread also has access to another type of memory that must not be accessed by other threads , called threadrivate memory
- **Shared variable**: each thread refers to the original variable.
- **Private variable:** Current thread's private version of the original variable.
- **Threadprivate**: variable appearing in threadprivae directives are threadprivate.

# 2. OpenMP Programming: threadprivate

```
#pragma omp threadprivate(list)
```

**Threadprivate**

- It specifies that named global-lifetime objects are replicated, with each thread having is own copy.
- Each copy of the *threadprivate* object is initialized once.
- A thread may not reference another thread's copy of a *threadprivate* object.
- A *threadprivate* object must not appear in any clause except the *copyin*, *copyprivate*, *schedule*, *num_threads*, and *if* clauses.
- The list is a comma-separated list of file-scope, names-scope, or static block-scope variables that do not have incomplete types.

# 2. OpenMP Programming: threadprivate

```
#pragma omp threadprivate(list)
```

**Private vs Threadprivate**

- Private variable scope is defined for only specific parallel region. *Threadprivate* is variable scope is declared across the parallel regions.
- *Firstprivate()* is used to copy the values from original variable. *Copyin()* is used to copy the values while entering into parallel region first time.
- Private variables are stored on stack most of the time. *Threadprivate* variables are stored in heap or thread local storage.

# 2. OpenMP Programming: threadprivate

```
#pragma omp threadprivate(list)

Copyin(list)
Copyprivate(list)
```

**Data Copying clauses**

- These clauses support the copying of data values from private or *threadprivate* variables on one implicit task or thread to the corresponding variables on other implicit tasks or threads in the team.
- **Copyin(list):** Copies the value of the master thread's *threadprivate* variable to the *threadprivate* variable of each other member of the team executing the parallel region.
- **Copyprivate (list)** : Broadcasts a value from the data environment of one implicit task to the data environments of the other implicit tasks belonging to the parallel region.

# Index

- OpenMP
  - Directives : if, for, master, single, Barrier, atomic, critical,
  - Directives
    - Sections
  - Clauses
    - Threadprivate
    - Collapse
    - Threadwait
    - Copyin, copyprivate
- References

# Reference

**Text Books and/or Reference Books:**

1. Professional CUDA C Programming – John Cheng, Max Grossman, Ty McKercher, 2014

2. B.Wilkinson, M.Allen, ”Parallel Programming: Techniques and Applications Using Networked Workstations and Parallel Computers”, Pearson Education, 1999

3. I.Foster, ”Designing and building parallel programs”, 2003

4. Parallel Programming in C using OpenMP and MPI – Micheal J Quinn, 2004

5. Introduction to Parallel Programming – Peter S Pacheco, Morgan Kaufmann Publishers, 2011

6. Advanced Computer Architectures: A design approach, Dezso Sima, Terence Fountain, Peter Kacsuk, 2002

7. Parallel Computer Architecture : A hardware/Software Approach, David E Culler, Jaswinder Pal Singh Anoop Gupta, 2011 8. Introduction to Parallel Computing, Ananth Grama, Anshul Gupta, George Karypis, Vipin Kumar, Pearson, 2011

# Reference

Acknowledgements

1. Introduction to OpenMP https://www3.nd.edu/~zxu2/acms60212-40212/Lec-12-OpenMP.pdf

2. Introduction to parallel programming for shared memory Machines https://www.youtube.com/watch?v=LL3TAHpxOig

3. OpenMP Application Program Interface Version 2.5 May 2005

4. OpenMP Application Program Interface Version 5.0 November 2018

# Thank You

**National Institute of Technology Karnataka Surathkal**
**Department of Information Technology**

# IT 301 Parallel Computing

## Shared Memory Programming Technique (7)

## OpenMP : *Task, taskwait*

**Dr. Geetha V**

*Assistant Professor*

*Dept of Information Technology*

*NITK Surathkal*

# Index

- OpenMP
  - Directives : Task
- References

# Course Outline

**Course Plan: Theory:**

**Part A: Parallel Computer Architectures**

Week 1,2,3: ***Introduction to Parallel Computer Architecture:*** Parallel Computing, Parallel architecture, bit level, instruction level , data level and task level parallelism. Instruction level parelllelism: pipelining(Data and control instructions), scalar and superscalar processors, vector processors. Parallel computers and computation.

Week 4,5: Memory Models: UMA, NUMA and COMA. Flynns classification, Cache coherence,

Week 6,7: Amdahl's Law. Performance evaluation, Designing parallel algorithms : Divide and conquer, Load balancing, Pipelining.

Week 8 -11: ***Parallel Programming techniques like Task Parallelism using TBB, TL2, Cilk++ etc. and software transactional memory techniques.***

# Course Outline

**Part B: OpenMP/MPI/CUDA**

Week 1,2,3 : ***Shared Memory Programing Techniques:*** Introduction to OpenMP : Directives: *parallel, for, sections, task, master, single, critical, barrier, taskwait, atomic. Clauses: private, shared, firstprivate, lastprivate, reduction, nowait, ordered, schedule, collapse, num_threads, if(), threadprivate, copyin, copyprivate*

Week 4,5: ***Distributed Memory programming Techniques:*** MPI: Blocking, Non-blocking.

Week 6,7 : CUDA : OpenCL, Execution models, GPU memory, GPU libraries.

Week 10,11,: ***Introduction to accelerator programming using CUDA/OpenCL and Xeon-phi. Concepts of Heterogeneous programming techniques.***

**Practical:**

Implementation of parallel programs using OpenMP/MPI/CUDA.

**Assignment:** Performance evaluation of parallel algorithms (in group of 2 or 3 members)

# 1. OpenMP

## FORK – JOIN Parallelism

- OpenMP program begin as a single process: the master thread. The master thread executes sequentially until the first parallel region construct is encountered.
- When a parallel region is encountered, master thread
  - Create a group of threads by FORK.
  - Becomes the master of this group of threads and is assigned the thread id 0 within the group.
- The statement in the program that are enclosed by the parallel region construct are then executed in parallel among these threads.
- JOIN: When the threads complete executing the statement in the parallel region construct, they synchronize and terminate, leaving only the master thread.

# 2. OpenMP Programming: Task

- Tasks are available starting with OpenMP 3.0
- A task is composed of
  - Code to be executed
  - Data environment (inputs to be used and outputs to be generated)
  - A Location where the task will be executed (thread)
- The tasks were initially implicit in OpenMP
  - A parallel construct constructs implicit tasks, one per thread
  - Teams of threads are created (or declared)
  - Threads in teams are assigned to each task
  - They synchronize with the master thread using barrier once all tasks completed
- Allowing the application to explicitly create tasks provide support for different execution models
  - More flexible
  - Requires scheduling of task
  - Moves away from the original fork/join model of OpenMP construct

# 2. OpenMP Programming: Task

- Assumption here is that tasks are independent. Ex : tree traversal

```
void preorder(node *p) {
    process(p->data);
    if (p->left)
    #pragma omp task
        preorder(p->left);
    if (p->right)
    #pragma omp task
        preorder(p->right);
}
```

# 2. OpenMP programming - Tasks

## Task – Technical Notion

- When a thread encounters a task construct, it may choose to execute the task immediately or defer its execution until a later time. If deferred, the task is placed in a conceptual pool of tasks associated with the current parallel region. All team threads will take tasks out of the pool and execute them until the pool is empty. A thread that executes a task might be different from the thread that originally encountered it.
- The code associated with a task construct will be executed only once. A task is **tied** if the code is executed by the same thread from beginning to end. Otherwise, the task is **untied** (the code can be executed by more than one thread).

# 2. OpenMP programming - Tasks

## Types of Task

- **Undeferred:** the execution is not deferred with respect to its generating task region, and the generating task region is suspended until execution of the **undeferred** task is completed (such as the tasks created with the if clause)
- **Included:** execution is sequentially included in the generating task region (such as a result from a final clause)
- **Subtle difference:** for **undeferred** task, the generating task region is suspended until execution of the **undeferred** task is completed, even if the **undeferred** task is not executed immediately.
  - The **undeferred** task may be placed in the conceptual pool and executed at a later time by the encountering thread or by some other thread; in the meantime, the generating task is suspended. Once the execution of the **undeferred** task is completed, the generating task can resume.
- A merged task is a task whose data environment is the same as that of its generating task region.

# 2. OpenMP Programming: Task

```
#pragma omp task [clause,….]
    Structured-block
```

**Clauses**

depend(list)

if(expression)

final(expression)

untied

mergeable

default(shared | firstprivate | none)

private(list)

firstprivate(list)

shared(list)

priority(value)

# 2. OpenMP Programming: Task

```
#pragma omp task [clause,….]
    Structured-block
```

**Clauses**

**depend(list)**

if(expression)

final(expression)

untied

mergeable

default(shared | firstprivate | none)

private(list)

firstprivate(list)

shared(list)

priority(value)

**Depend :**

- It enforces additional constraints between tasks
- The list in depend contains storage locations (memory addresses) on which the dependency will be tracked
- The dependency is mentioned with **in, out,inout**

```
#pragma omp task shared(a,b) depend(out a,b)
task1(……)     //T1
#pragma omp shared(a) task depend(in a)
task2(……)    //T2
#pragma omp task shared(b) depend(in b)
task3(……)   //T3
```

T1 must be executed first. T2 and T3 can be executed in parallel

# 2. OpenMP Programming: Task

```
#pragma omp task [clause,….]
    Structured-block
```

**Clauses**

**depend(list)**

if(expression)

final(expression)

untied

mergeable

default(shared | firstprivate | none)

private(list)

firstprivate(list)

shared(list)

priority(value)

**Depend :**

- In: The generated task will be dependent of all previously generated sibling tasks that reference at least one of the list items in an **out** or **inout** dependence type.
- Out and **inout**: The generated task will be dependent of all previously generated sibling tasks that reference at least one of the list items in an **in,out** or **inout** dependence type list

```
#pragma omp task shared(a,b) depend(out a,b)
task1(……)     //T1
#pragma omp shared(a) task depend(in a)
task2(……)    //T2
#pragma omp task shared(b) depend(in b)
task3(……)   //T3
```

T1 must be executed first. T2 and T3 can be executed in parallel

# 2. OpenMP Programming: Task

```
#pragma omp task [clause,….]
    Structured-block
```

**Clauses**

depend(list)

**if(expression)**

final(expression)

untied

mergeable

default(shared|firstprivate|none)

private(list)

firstprivate(list)

shared(list)

priority(value)

**If (expression):**

- When the if expression argument evaluate to false, an undeferred task is generated, and the encountering thread must suspend the current task region, for which execution cannot be resumed until execution of the strucured block that is associated with the generated task is completed.
- The use of variable in an if clause expression of a task construct causes an implicit reference to the variable in all enclosing constructs

# 2. OpenMP Programming: Task

```
#pragma omp task [clause,....]
    Structured-block
```

**Clauses**

depend(list)

if(expression)

final(expression)

untied

mergeable

default(shared | firstprivate | none)

private(list)

firstprivate(list)

shared(list)

priority(value)

**If (expression):**

```
#pragma omp task if(0) // This is undeferred
{
  #pragma omp task // This is a regular task
     for( i = 0; i < 3; i++ ) {
       #pragma omp task // This is a regular task
         bar();
     }
}
```

# 2. OpenMP Programming: Task

```
#pragma omp task [clause,….]
    Structured-block
```

**Clauses**

depend(list)

if(expression)

final(expression)

untied

mergeable

default(shared|firstprivate|none)

private(list)

firstprivate(list)

shared(list)

priority(value)

**final (expression):**

- When the final clause argument is true
  - The generated task will be a final task
  - All tasks encountered during execution of a final task will generate included tasks
    - An included task is a task for which execution is sequentially included in the generating task region; that is undeferred and executed immediately by the encountering threads
- It is another user directed optimization
- Omp_in_final() returns true if the enclosing task region is final. Otherwise, it returns false.

# 2. OpenMP Programming: Task

```
#pragma omp task [clause,….]
    Structured-block
```

**Clauses**

depend(list)

if(expression)

final(expression)

untied

mergeable

default(shared | firstprivate | none)

private(list)

firstprivate(list)

shared(list)

priority(value)

**final (expression):**

```
#pragma omp task final(1) // This is a regular task
{
  #pragma omp task // This is included
    for(i=0;i<3;i++){
      #pragma omp task // This is also included
        bar();
    }
}
```

# 2. OpenMP Programming: Task

```
#include <stdio.h>>
#include <stdlib.h>
#include <omp.h>
int main(void)
{
  printf("A ");
  printf("Race ");
  printf("Car ");
    printf("\n");
return 0;
}
```

# 2. OpenMP Programming: Task

```
#include <stdio.h>>
#include <stdlib.h>
#include <omp.h>
int main(void)
{
  printf("A ");
  printf("Race ");
  printf("Car ");
    printf("\n");
return 0;
}
```

- The output of the program is

A Race Car

# 2. OpenMP Programming: Task

```
#include <stdio.h>>
#include <stdlib.h>
#include <omp.h>
int main(void)
{
#pragma omp parallel {
  printf("A ");
  printf("Race ");
  printf("Car ");
}
   printf("\n");
return 0;
}
```

- The output of the program is

A Race Car A Race Car A Race Car

(Assume three threads)

# 2. OpenMP Programming: Collapse

```
#include <stdio.h>>
#include <stdlib.h>
#include <omp.h>
int main(void)
{
#pragma omp parallel {
#pragma omp single{
  printf("A ");
  printf("Race ");
  printf("Car ");
} }
   printf("\n");
return 0;
}
```

- The output of the program is

A Race Car

(Assume three threads)

# 2. OpenMP Programming: Task

```
#include <stdio.h>>
#include <stdlib.h>
#include <omp.h>
int main(void)
{
#pragma omp parallel {
#pragma omp single{
  printf("A ");
#pragma omp task
   { printf("Race "); }
#pragma omp task
  { printf("Car "); }
} }
    printf("\n");
return 0;
}
```

- The output of the program is

A Race Car

Or

A Car Race

(Assume three threads)

# 2. OpenMP Programming: Task

```
#include <stdio.h>>
#include <stdlib.h>
#include <omp.h>
int main(void)
{
#pragma omp parallel {
#pragma omp single{
  printf("A ");
#pragma omp task
   { printf("Race "); }
#pragma omp task
  { printf("Car "); }
printf("is fun to watch");
} }
    printf("\n");
return 0;
```

- The output of the program is

**A is fun to watch Race Car**

Or

**A is fun to watch Car Race**

(Assume two threads)

# 2. OpenMP Programming: Task

```
int main(void)
{
#pragma omp parallel {
#pragma omp single{
  printf("A ");
#pragma omp task
   { printf("Race "); }
#pragma omp task
  { printf("Car "); }
#pragma omp taskwait
printf("is fun to watch");
} }
    printf("\n");
return 0;
}
```

- The output of the program is

**A  Race Car  is fun to watch**

Or

**A Car Race is fun to watch**

(Assume two threads)

# 2. OpenMP Programming: Task

```
#pragma omp task [clause,….]
    Structured-block
```

**Clauses**

depend(list)

if(expression)

final(expression)

**untied**

mergeable

default(shared|firstprivate|none)

private(list)

firstprivate(list)

shared(list)

priority(value)

**untied**

- Different parts of the task can be executed by different threads. Implies the tasks will yield, allowing the executing thread to switch context and execute another task instead.
- If the task is tied, it is guaranteed that the same thread will execute all the parts of the task, even if the task execution has been temporarily suspended
- An **untied** task generator can be moved from thread to thread allowing the tasks to be generated by different entities

# 2. OpenMP Programming: Task

```
#pragma omp task [clause,….]
    Structured-block
```

**Clauses**

depend(list)

if(expression)

final(expression)

untied

**mergeable**

default(shared | firstprivate | none)

private(list)

firstprivate(list)

shared(list)

priority(value)

**meargeable**

- A merged task is a task whose data environment is the same as that of its generating task region.
- When a mergeable clause is present on a task construct, then the implementation may choose to generate a merged task instead.
- If a merged task is generated, then the behavior is as though there was no task directive at all

# 2. OpenMP Programming: Task

```
#pragma omp task [clause,….]
    Structured-block
```

**Clauses**

depend(list)

if(expression)

final(expression)

untied

mergeable

**default(shared | firstprivate | none)**

**private(list)**

**firstprivate(list)**

shared(list)

priority(value)

**Default, private, firstprivate**

- Default defines the data-sharing attributes of variables that are referenced
- **firstprivate:** each construct has a copy of the data item, and it is initialized from the upper construct before the call
- **shared:** All references to a list item within a task refer to the storage area of the original variable
- **private:** each task receive a new item

# 2. OpenMP Programming: Task

```
#pragma omp task [clause,….]
    Structured-block
```

**Clauses**

depend(list)

if(expression)

final(expression)

untied

mergeable

default(shared | firstprivate | none)

private(list)

firstprivate(list)

shared(list)

**priority(value)**

**Priority(n)**

- Priority is a hint for the scheduler. A non-negative numerical value, that recommend a task with a high priority to be executed before a task with lower priority

# Index

- OpenMP
  - Directives : Task
- References

# Reference

**Text Books and/or Reference Books:**

1. Professional CUDA C Programming – John Cheng, Max Grossman, Ty McKercher, 2014

2. B.Wilkinson, M.Allen, ”Parallel Programming: Techniques and Applications Using Networked Workstations and Parallel Computers”, Pearson Education, 1999

3. I.Foster, ”Designing and building parallel programs”, 2003

4. Parallel Programming in C using OpenMP and MPI – Micheal J Quinn, 2004

5. Introduction to Parallel Programming – Peter S Pacheco, Morgan Kaufmann Publishers, 2011

6. Advanced Computer Architectures: A design approach, Dezso Sima, Terence Fountain, Peter Kacsuk, 2002

7. Parallel Computer Architecture : A hardware/Software Approach, David E Culler, Jaswinder Pal Singh Anoop Gupta, 2011 8. Introduction to Parallel Computing, Ananth Grama, Anshul Gupta, George Karypis, Vipin Kumar, Pearson, 2011

# Reference

Acknowledgements

1. Introduction to OpenMP https://www3.nd.edu/~zxu2/acms60212-40212/Lec-12-OpenMP.pdf

2. Introduction to parallel programming for shared memory Machines https://www.youtube.com/watch?v=LL3TAHpxOig

3. OpenMP Application Program Interface Version 2.5 May 2005

4. OpenMP Application Program Interface Version 5.0 November 2018

5 Introduction to openmp tasks

http://www.icl.utk.edu/~luszczek/teaching/courses/fall2016/cosc462/pdf/W45L1%20-%20OpenMP%20Tasks.pdf

# Thank You

Department of Information Technology
National Institute of Technology Karnataka, Surathkal

# Distributed Memory Parallelism with MPI

By,
Geetha V
Dept of IT,
NITK Surathkal

# Outline

- Distributed Memory Architecture
- Introduction to MPI
- Structure of MPI program
- Types of Message Passing
- Basic Routines in Point to Point Communication
- Example programs on Point to Point Communication
- Basic Routines in Collective Communication
- Sample Programs on Collective Communication

# Distributed Memory Architecture

- Each processor has its own memory
- They cannot access the memory of other processors.
- Any data that needs to be shared must be explicitly transmitted from one processor to another using **Message Passing.**

# DISTRIBUTED MEMORY ARCHITECTURE

- **Systems with single core communicating through distributed memory.**
- **Heterogeneous systems**

# DISTRIBUTED MEMORY ARCHITECTURE

- **Systems with multiple core communicating through shared and distributed memory**

# Hybrid Model

Reference: https://computing.llnl.gov/tutorials/parallel_comp/#ModelsMessage

# Hybrid Model

Reference: https://computing.llnl.gov/tutorials/parallel_comp/#ModelsMessage

# Parallel Computation:

**Divide large task into small tasks and allot it to multiple processes**

**For Example: N = 1,00,000 divided into P1=20,000, P2=20,000 ……**

**Computation is same. Data is different. Single Program Multiple Data**

# INTRODUCTION TO MPI

- **What is MPI?**
  - Message Passing Interface is a specification.
    - A standard for vendors to implement.
  - It is a library, i.e. a set of subroutines, functions and constants
  - Allows Message Passing between processes.
  - It is based on Single Program, Multiple Data (SPMD)
    - Every process executes the same program
    - Each process performs computations on its local variables, then communicates with other processes, in order to get the final result.

# MPI : Major Goals

- Portability :
  - An MPI library exists on ALL parallel computing platforms so it is highly portable.
- Support heterogeneity
- High performance through efficient implementations
- Encourage overlap of communication and computations.
- Reliability

# MPI is a Middleware

# MPI is a Middleware

# MPI Implementations

- OpenMPI (www.open-mpi.org)
- MPICH (www.mpich.org)
- HP MPI
- Intel MPI
- Scali MPI
- IBM MPI

# Outline

- Distributed Memory Architecture
- Introduction to MPI
- Structure of MPI program
- Types of Message Passing
- Basic Routines in Point to Point Communication
- Example programs on Point to Point Communication
- Basic Routines in Collective Communication
- Sample Programs on Collective Communication

# STRUCTURE OF MPI PROGRAM

**MPI Include File**

**Initialize MPI Environment**

**Computations and Message Passing**

**Terminate MPI Environment**

# MPI Routines

- **Start and terminate :**
  - To initialize and terminate the MPI environment
- **Communicators :**
  - To identify the communication world (cluster of processes)
- **Getting Information :**
  - To get the number of processes and process ids
- **Sending and Receiving messages :**
  - Actual computation and communication

# STRUCTURE OF MPI PROGRAM

MPI Include File

```
#include<mpi.h>
```

Initialize MPI Environment

```
MPI_Init(&argc,&argv);
```

**Computations and Message Passing**

Terminate MPI Environment

```
MPI_Finalize();
```

# MPI Start and Terminate Routines

```
#include<stdio.h>
int main(int argc,char **argv)
        {
            ------------
            ------------
         MPI_Init(&argc,&argv);
          ------------
            ------------
         MPI_Finalize();
         ------------
         return 0;
         }
```

# Communicators

- MPI defines **communication domain** – set of processes that can communicate with each other.
- MPI_comm : data type – stores information about communication domains.
- Default communicator - MPI_COMM_WORLD

Communication Domain

# Getting Information

- MPI_Comm_size
- MPI_Comm_rank

- **Syntax :**
- **int MPI_Comm_size(MPI_Comm comm, int *size)**
- **Int MPI_Comm_rank(MPI_Comm comm, int *rank)**

# General MPI Program

```
#include<mpi.h>
        int main(int argc,char **argv)
        {
         -----------
         -----------
         MPI_Init(&argc,&argv);
         -----------
         MPI_Comm_size(MPI_COMM_WORLD,&size);
         MPI_Comm_rank(MPI_COMM_WORLD,&rank);
         -----------
         MPI_Finalize();
         -----------
         return 0;
         }
```

# Example: Hello World

```
#include<mpi.h>
int main(int argc,char *argv[ ])
{
 int size,myrank;
 MPI_Init(&argc,&argv);
 MPI_Comm_size(MPI_COMM_WORLD,&size);
 MPI_Comm_rank(MPI_COMM_WORLD,&myrank);
 printf("Process %d of %d, Hello World",myrank,size);
 MPI_Finalize();
 return 0;
 }
```

# MPI Hello World :

```
tans@tans-Inspiron-3542:~/PC$ mpiexec -n 5 ./a.out
Process 0 of 5, Hello World
Process 1 of 5, Hello World
Process 4 of 5, Hello World
Process 2 of 5, Hello World
Process 3 of 5, Hello World
```

**MPI Include File**

**Initialize MPI Environment**

**Computations and Message Passing**

**Terminate MPI Environment**

# Types of Message Passing:

- **Point to Point**
  - Two processes
  - Send and Receive are the basic functions
- **Collective messages**
  - Group of processes involved in communication
  - Functions like Broadcast, Scatter, Gather, Parallel Reduction

# Point to Point Communication

- **Two processes involved in sending and receiving data.**

- **ID of sender and receiver is required.**
- **Specify what has to be sent and received.**
- **Communication needs to be synchronized.**
- **Communication makes use of buffers.**

# Point to Point Communication

- **Data transfer from Sender Process to Receiver Process.**

Path of a message buffered at the receiving process

# Send and Receive Variants

- **Blocking Send and Receive**
- **Non Blocking Send and Receive**
- **Based on modes of Communication:**
  - Standard
  - Synchronous
  - Buffered
  - Ready

# Blocking Send and Receive

- **Basic Send and Receive routine for point to point communication.**
- **MPI Routines:**
  - MPI_Send()
  - MPI_Recv()

# Outline

- Distributed Memory Architecture
- Introduction to MPI
- Structure of MPI program
- Types of Message Passing
- **Basic Routines in Point to Point Communication**
- **Example programs on Point to Point Communication**
- Basic Routines in Collective Communication
- Sample Programs on Collective Communication

# Blocking Send and Receive

- **MPI_Send()**

  **MPI_Send (void *buf, int count, MPI_Datatype type,int dest, int tag, MPI_Comm comm)**

**Parameters:**

| | |
|---|---|
| **buf :** | initial address of send buffer |
| **count :** | number of elements in send buffer (nonnegative integer) |
| **datatype :** | datatype of each send buffer element. Ex : MPI_INT, MPI_CHAR |
| **dest :** | rank of destination (integer) |
| **tag :** | message tag (integer). For tagging send and receive. |
| **comm :** | Communication domain of the communicating processes. |

# Blocking Send and Receive

- **MPI_Recv():**

**MPI_Recv(void *buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Status *status)**

**Parameters:**

| | |
|---|---|
| **buf :** | initial address of receive buffer |
| **count :** | max number of elements in receive buffer (nonnegative integer) |
| **datatype :** | datatype of each receive buffer element. Ex : MPI_INT, MPI_CHAR |
| **source :** | rank of source (integer) |
| **tag :** | message tag (integer). For tagging send and receive. |
| **comm :** | Communication domain of the communicating processes. |
| **status:** | status object (Status). It is a structure containing information about source, tag and error code. |

- **MPI DATATYPES:**

*Table 1: Basic C datatypes in MPI*

| MPI Datatype | C datatype |
|---|---|
| MPI_CHAR | signed char |
| MPI_SHORT | signed short int |
| MPI_INT | signed int |
| MPI_LONG | signed long int |
| MPI_UNSIGNED_CHAR | unsigned char |
| MPI_UNSIGNED_SHORT | unsigned short int |
| MPI_UNSIGNED | unsigned int |
| MPI_UNSIGNED_LONG | unsigned long int |
| MPI_FLOAT | float |
| MPI_DOUBLE | double |
| MPI_LONG_DOUBLE | long double |
| MPI_BYTE | |
| MPI_PACKED | |

# General MPI Program

```
#include<mpi.h>
        int main(int argc,char **argv)
        {
                ...
                MPI_Init(&argc,&argv);
                ...
                MPI_Comm_size(MPI_COMM_WORLD,&size);
                MPI_Comm_rank(MPI_COMM_WORLD,&rank);

        COMPUTATIONS AND MESSAGE PASSING

                MPI_Finalize();
                ...
                return 0;
        }
```

# MPI Example - 1

```
        for(i=0;i<50;i++) //Process 0 initializes array x
        x[i]=i+1;
if(myrank==0)
MPI_Send(x,10,MPI_INT,1,1,MPI_COMM_WORLD);
else if(myrank==1)
{
MPI_Recv(y,10,MPI_INT,0,1,MPI_COMM_WORLD,&status);
printf("Process %d Received Data from Process %d\n",
myrank,status.MPI_SOURCE);
        for(i=0;i<10;i++)
        printf("%d\t",y[i]);
        }
```

```
Process 1 Recieved data from Process 0
1       2       3       4       5       6       7       8       9       10
```

# Non Blocking Send and Receive

- **Allows overlapping of computation and communication**
- **Advantage is Performance Gain**

# Non Blocking Send and Receive

**MPI_Isend (&buf,count,datatype,dest,tag,comm,&request)**

**MPI_Irecv (&buf,count,datatype,source,tag,comm,&request)**

**Parameters:**

- **Same as Send() and Recv() except for request**
- **request : handle. This helps to get information about MPI_Isend and MPI_Irecv status.**
- **Used in routines : MPI_Wait() and MPI_Test()**

# Non Blocking Send and Receive

# MPI_Wait() and MPI_Test()

**Syntax :**

```
int MPI_Wait( MPI_Request *request, MPI_Status *status );

int MPI_Test( MPI_Request *request, int *flag, MPI_Status *status );
```

- **If request is set to MPI_REQUEST_NULL (set if operation is completed) then:**
  - **MPI_Wait returns immediately with an empty status.**
  - **MPI_Test sets flag to true and returns an empty status.**

# MPI Example - 2

```
if(myrank==0)
{
x=10;
MPI_Isend(&x,1,MPI_INT,1,20,MPI_COMM_WORLD,&request);
printf("Send returned immediately\n");
}
else if(myrank==1)
{
MPI_Irecv(&x,1,MPI_INT,0,25,MPI_COMM_WORLD,&request);
printf("Receive returned immediately\n");
printf("Process %d of %d, Value of x is %d\n",myrank,size,x);
}
```

```
tans@tans-Inspiron-3542:~/PC$ mpiexec -n 2 ./a.out
Send returned immediately
Receive returned immediately
Process 1 of 2, Value of x is 0
```

# What is the risk here?

```
if(myrank==0)
{
x=10;
MPI_Isend(&x,1,MPI_INT,1,20,MPI_COMM_WORLD,&request);
printf("Send returned immediately\n");
x=x+10;
}
```

# Make sure that x is available for reuse:

```
if(myrank==0)
{
x=10;
MPI_Isend(&x,1,MPI_INT,1,20,MPI_COMM_WORLD,&request);
printf("Send returned immediately\n");
MPI_Wait(request, status)
x=x+10;
}
```

# Communication Modes

- **Standard Mode :** Calls block until message has been either transferred or copied to an internal buffer for later delivery. Ex: MPI_Send() and MPI_Recv()
- **Buffered Mode :** Send may start and return before a matching receive. MPI_Bsend()
- **Synchronous Mode :** Call blocks until matching receive has been posted and the message reception has started. MPI_Ssend()
- **Ready Mode :** Requires that a matching receive is already posted. MPI_Rsend().

# Buffered Mode

| MPI_BUFFER_ATTACH( buffer, size) | |
|---|---|
| buffer | initial buffer address (choice) |
| size | buffer size, in bytes (integer) |

NOTE: A user may specify a buffer to be used for buffering messages sent in buffered mode.

Image Reference: https://www.codingame.com/playgrounds/47058/have-fun-with-mpi-in-c/communication-modes

# Synchronous Mode

**We see that the data is not copied to system buffer.**

Image Reference: https://www.codingame.com/playgrounds/47058/have-fun-with-mpi-in-c/communication-modes

# Ready Mode

**We make use of MPI_Barrier() to wait for the receive to be posted. This will not result in error.**

Image Reference: https://www.codingame.com/playgrounds/47058/have-fun-with-mpi-in-c/communication-modes

# MPI-Example - 3

```
if(myrank==0) {

//Blocking send will expect matching receive at the destination In Standard mode,Send will
return after copying the data to the buffer

   MPI_Send(x,10,MPI_INT,1,1,MPI_COMM_WORLD);

// This send will be initiated and matching receive is already there so the program will not
lead to deadlock

  MPI_Send(y,10,MPI_INT,1,2,MPI_COMM_WORLD);
}
```

```
else if(myrank==1)
{
//P1 will block as it has not received a matching send with tag 2

  MPI_Recv(x,10,MPI_INT,0,2,MPI_COMM_WORLD,&status);
  MPI_Recv(y,10,MPI_INT,0,1,MPI_COMM_WORLD,MPI_STATUS_IGNORE);

}
```

# MPI Example 3

**PROCESS 1**

*MPI_Send(x,10,..1,1,..);*

*MPI_Send(y,10,..,1,2,..);*

**PROCESS 2**

*MPI_Recv(x,10,..,0,2,..,..);*

**BLOCK**

*MPI_Recv(y,10,..,0,1,..,..);*

# MPI Example - 4

```
if(myrank==0) {

   MPI_Ssend(x,10,MPI_INT,1,1,MPI_COMM_WORLD);

   MPI_Send(y,10,MPI_INT,1,2,MPI_COMM_WORLD);
}
```

```
else if(myrank==1)
{
   MPI_Recv(x,10,MPI_INT,0,2,MPI_COMM_WORLD,&status);

MPI_Recv(y,10,MPI_INT,0,1,MPI_COMM_WORLD,MPI_STATUS_IGNORE);

}
```

# MPI Example - 4

```
if(myrank==0) {

   MPI_Ssend(x,10,MPI_INT,1,1,MPI_COMM_WORLD);

// Synchronous Blocking send will expect matching receive at the destination.
This results in deadlock.

   MPI_Send(y,10,MPI_INT,1,2,MPI_COMM_WORLD); //This call will not be
executed
}
```

```
else if(myrank==1)
{
   MPI_Recv(x,10,MPI_INT,0,2,MPI_COMM_WORLD,&status);//P1 will block
as it has not received a matching send with tag 2


MPI_Recv(y,10,MPI_INT,0,1,MPI_COMM_WORLD,MPI_STATUS_IGNORE);

}
```

# Outline

- Distributed Memory Architecture
- Introduction to MPI
- Structure of MPI program
- Types of Message Passing
- **Basic Routines in Point to Point Communication**
- **Example programs on Point to Point Communication**
- Basic Routines in Collective Communication
- Sample Programs on Collective Communication

# Communication Modes

- **Standard Mode :** Calls block until message has been either transferred or copied to an internal buffer for later delivery. Ex: MPI_Send() and MPI_Recv()
- **Buffered Mode :** Send may start and return before a matching receive. MPI_Bsend()
- **Synchronous Mode :** Call blocks until matching receive has been posted and the message reception has started. MPI_Ssend()
- **Ready Mode :** Requires that a matching receive is already posted. MPI_Rsend().

# Buffered Mode

| MPI_BUFFER_ATTACH( buffer, size) | |
|---|---|
| buffer | initial buffer address (choice) |
| size | buffer size, in bytes (integer) |

NOTE: A user may specify a buffer to be used for buffering messages sent in buffered mode.

Image Reference: https://www.codingame.com/playgrounds/47058/have-fun-with-mpi-in-c/communication-modes

# Synchronous Mode

**We see that the data is not copied to system buffer.**

Image Reference: https://www.codingame.com/playgrounds/47058/have-fun-with-mpi-in-c/communication-modes

# Ready Mode

**We make use of MPI_Barrier() to wait for the receive to be posted. This will not result in error.**

Image Reference: https://www.codingame.com/playgrounds/47058/have-fun-with-mpi-in-c/communication-modes

# Outline

- Distributed Memory Architecture
- Introduction to MPI
- Structure of MPI program
- Types of Message Passing
- Basic Routines in Point to Point Communication
- Example programs on Point to Point Communication
- **Basic Routines in Collective Communication**
- **Sample Programs on Collective Communication**

# Collective Communication

- **Multiple processes in same communicator involve in collective communication.**
- **They are blocking calls.**
- **No tags required.**

# Collective Communication

- Barrier
- Broadcast
- Scatter
- Gather
- Reduce
- Scatterv
- Gatherv

# Collective communication: MPI_Barrier

- **Mainly used for synchronization**
- **The call returns only after all the processes have called Barrier function.**
- **Uses:**
  - Access to files
  - Achieve consistency

**Syntax: MPI_Barrier(MPI_COMM_WORLD)**

# Collective Communication: Broadcast

- **MPI_Bcast(buf, count, datatype, source, comm)**
  - buf : send buffer of sender and receive buffer of receiver
  - source : process which sends data to others

broadcast

# MPI Example - 5

```
if(myrank==0)
{
scanf("%d",&x);
}
MPI_Bcast(&x,1,MPI_INT,0,MPI_COMM_WORLD);
printf("Value of x in process %d : %d\n",myrank,x);
MPI_Finalize();
return 0;
}
```

# Bcast():

Process 0
x=10
Bcast(x)
Process 1
x=10
Process 2
x=10
Process 3
x=10

# Broadcast Output:

```
tans@tans-Inspiron-3542:~/PC$ mpiexec -n 4 ./a.out
3
Value of x in process 0 : 3
Value of x in process 1 : 3
Value of x in process 2 : 3
Value of x in process 3 : 3
```

# Collective Communication: Scatter

# Collective Communication: Scatter

**MPI_Scatter(sendbuf, sendcount, datatype, recvbuf, recvcount, datatype, root, comm)**

**Parameters:**

- **sendbuf : sender buffer**
- **sendcount : specify the number of elements to be sent. recvcount should be same as sendcount**
- **recvbuf : recv buffer**
- **root : Sender**

# MPI_Scatter

**Example:**

# MPI Example - 6

```
if(myrank==0)
{
printf("Enter values into array x:\n");
for(i=0;i<8;i++)
scanf("%d",&x[i]);
}
MPI_Scatter(x,2,MPI_INT,y,2,MPI_INT,0,MPI_COMM_WORLD);
for(i=0;i<2;i++)
printf("\nValue of y in process %d : %d\n",myrank,y[i]);
```

# Output

```
tans@tans-Inspiron-3542:~/PC$ mpiexec -n 4 ./a.out
Enter values into array x:
1 2 3 4 5 6 7 8

Value of y in process 0 : 1

Value of y in process 0 : 2

Value of y in process 1 : 3

Value of y in process 1 : 4

Value of y in process 2 : 5

Value of y in process 2 : 6

Value of y in process 3 : 7

Value of y in process 3 : 8
```

# Collective Communication: Gather

# Collective Communication: Gather

**MPI_Gather(sendbuf, sendcount, datatype, recvbuf, recvcount, datatype, root, comm)**

**Parameters:**

- **sendbuf: buffer of sending processes**
- **sendcount and recvcount value is same**
- **recvbuf: root process's buffer**
- **root : process where the data is gathered**

# MPI_Gather

# MPI-Example 7

```
x=10, y[50]

MPI_Gather(&x,1,MPI_INT,y,1,MPI_INT,0,MPI_COMM_WORLD);
// Value of x at each process is copied to array y in Process 0
if(myrank==0)
{
for(i=0;i<size;i++)
printf("\nValue of y[%d] in process %d : %d\n",i,myrank,y[i]);
}
```

# Output

```
tans@tans-Inspiron-3542:~/PC$ mpiexec -n 4 ./a.out

Value of y[0] in process 0 : 10

Value of y[1] in process 0 : 10

Value of y[2] in process 0 : 10

Value of y[3] in process 0 : 10
tans@tans-Inspiron-3542:~/PC$ mpiexec -n 6 ./a.out

Value of y[0] in process 0 : 10

Value of y[1] in process 0 : 10

Value of y[2] in process 0 : 10

Value of y[3] in process 0 : 10

Value of y[4] in process 0 : 10

Value of y[5] in process 0 : 10
```

# Collective Communication: Reduce

- Allows to perform computations on data present at multiple processes.
- Computations like : Sum, Product, Maximum, Minimum
- Stores the result in one process.

# Collective Communication: Reduce

**MPI_Reduce(sendbuf, recvbuf, count, datatype, operation, dest, comm)**

**Parameters:**

**count: size of receive buffer**

**operation:**

| MPI name | Operation |
|---|---|
| MPI_MAX | Maximum |
| MPI_MIN | Minimum |
| MPI_SUM | Summation |
| MPI_PROD | Product |
| MPI_LAND | Logical AND |
| MPI_LOR | Logical OR |
| MPI_LXOR | Logical XOR |

# MPI Example - 8

```
x=myrank;
MPI_Reduce(&x,&y,1,MPI_INT,MPI_SUM,0,MPI_COMM_WORLD)
;
if(myrank==0)
{
printf("Value of y after reduce : %d\n",y);
}
```

# Output

```
tans@tans-Inspiron-3542:~/PC$ mpiexec -n 3 ./a.out
Value of y after reduce : 3
tans@tans-Inspiron-3542:~/PC$ mpiexec -n 4 ./a.out
Value of y after reduce : 6
```

# Outline

- Distributed Memory Architecture
- Introduction to MPI
- Structure of MPI program
- Types of Message Passing
- Basic Routines in Point to Point Communication
- Example programs on Point to Point Communication
- **Basic Routines in Collective Communication**
- **Sample Programs on Collective Communication**

# MORE Collective Communication Routines:

- MPI_Gatherv()
- MPI_Scatterv()
- MPI_Allgather
- MPI_AllReduce()
- MPI_Scan()
- MPI_Comm_Split()

# MPI_Allgather

# MPI_Scatterv

# MPI_Scatterv():

**MPI_Scatterv(sendbuf, sendcounts, displacement, datatype, recvbuf, recvcount, datatype, root, comm)**

**Parameters:**

**sendcounts :** array with number of elements to be sent to each process. ex: sendcount[0]=10 means send 10 elements to Process zero. sendcount[1]=20 means send 20 elements to Process one.

**displacement:** array which holds the index from where the data is to be sent to each process. Ex: disp[0]=0 means Process zero gets elements starting with index zero. disp[1]=10 means Process 1 will get elements starting from index 10.

# MPI_Scatterv

```
if(myrank==0)
        {
        printf("Enter the number of elements:\n");
        scanf("%d",&m);
        printf("Initializing array x:\n");
        for(i=0;i<m;i++)
        x[i]=i+1;
        disp[0]=0;
        for(i=0;i<size;i++)
        {
        z[i]=i+2; // Process 0 gets 2 elements, Process 1 gets 3, Process 2 gets 4 and so on
        disp[i+1]=disp[i]+z[i]; // disp[1]=2, disp[2]=5,disp[3]=9 and so on
        }
}

MPI_Scatterv(x,z,disp,MPI_INT,y,myrank+2,MPI_INT,0,MPI_COMM_WORLD);
```

# OUTPUT:

```
Enter the number of elements:
20

Value of y in process 1 : 3

Value of y in process 1 : 4

Value of y in process 1 : 5

Value of y in process 2 : 6

Value of y in process 2 : 7

Value of y in process 2 : 8

Value of y in process 2 : 9

Value of y in process 3 : 10

Value of y in process 3 : 11

Value of y in process 3 : 12

Value of y in process 3 : 13

Value of y in process 3 : 14

Value of y in process 4 : 15

Value of y in process 4 : 16

Value of y in process 4 : 17

Value of y in process 4 : 18

Value of y in process 4 : 19

Value of y in process 4 : 20
Initializing array x:

Value of y in process 0 : 1

Value of y in process 0 : 2
```

# MPI_Gatherv

# MPI_Gatherv():

**MPI_Gatherv(sendbuf, sendcount, datatype, recvbuf, recvcounts, displacements, datatype, root, comm)**

**Parameters:**

**recvcounts : array with number of elements to be received from each process.**

**displacement: array which holds the beginning index where the data is to be received from each process.**

# MPI_Scan

**int MPI_Scan(sendbuf, recvbuf, int count, datatype, MPI_Op, comm)**

# MPI_Comm_Split : Split the communication Domain

**MPI_Comm_Split(MPI_Comm comm, int color, int key, MPI_Comm *newcomm);**

**color: controls subset assignment**

**key: controls rank assignment of processes in different group**

Ex: MPI_Comm_split(MPI_COMM_WORLD,0,0,&comm1);

MPI_Comm_split(MPI_COMM_WORLD,1,0,&comm2);

MPI_Comm_split(MPI_COMM_WORLD,2,0,&comm3);

# MPI Collective Routine

Reference: Introduction to MPI and OpenMP (with Labs) Brandon Barker Computational Scientist Cornell University Center for Advanced Computing (CAC) . https://www.cac.cornell.edu/

# Summary

- MPI provides a simplified way for sending and receiving messages
- MPI rich set of collective functions
- MPI helps for developing Scalable and Portable Parallel Programs
- MPI is the defacto standard for Distributed Memory Parallelism

# Thank You

Department of Information Technology
National Institute of Technology Karnataka, Surathkal

# IT301: INTRODUCTION TO CUDA

(Compute Unified Device Architecture)

**Dr Geetha V**
**Asst Professor Grade I**
**Department of IT,**
**NITK, Surathkal**

# OUTLINE

1. Introduction to GPU
2. Introduction to CUDA
3. Microarchitecture of GPU
4. CUDA Execution and Memory Model
5. Hello World Program
6. CUDA Device Variables

# 1. Introduction to GPU

# Let's learn about GPU..

- A little history:
  - The first GPUs were designed as graphics accelerators
    - supported only specific fixed-function pipelines.
  - In the late 1990s, the hardware became increasingly programmable
    - Culminating in NVIDIA's first GPU in 1999.

The improvement in GUI of computer over the years.....

# CPU versus GPU

# Graphics Processing Unit

- Has thousands of cores and ALUs.
- They can handle billions of repetitive low level tasks.
- GPU is specialized for compute-intensive, highly parallel computation.
- They are devoted to data processing rather than data caching and flow control.
- Follows SPMD (Single Program Multiple Data) processing model.

# More closer look at GPU:

# Latency Hiding in GPU

# General Purpose GPU : GPGPU

# CPU vs GPU

- Need to understand how CPUs and GPUs differ
  - Simpler calculation versus complex calculation
  - Basic graphics versus 3D rendering, animations.
  - Few higher capacity cores versus many low capacity cores
  - Latency Intolerance versus Latency Tolerance
  - Task Parallelism versus Data Parallelism
  - 10s of Threads versus 10,000s of Threads

# The dawn of GPGPU

General Purpose Computing on GPU was far from easy back then

- Even for those who knew graphics programming languages such as OpenGL!
- Developers had to map scientific calculations onto problems that could be represented by triangles and polygons.

# Applications

# Applications

- Machine Learning – self driving cars, Watson AI Supercomputer.
- Scientific Applications such as Genome sequencing, molecular simulations.
- Medical Image processing.
- Image tagging in Facebook.
- Numeric weather predictions.
- Oil exploration.
- Movie making.
- Atmospheric simulation.
- Sequencing the novel coronavirus and the genomes of people afflicted with COVID-19.

# 2. Introduction CUDA

# CUDA – Compute Unified Device Architecture

- In 2003, a team of researchers led by Ian Buck unveiled Brook, the first widely adopted programming model to extend C with data-parallel constructs.
- Exposed the GPU as a general - purpose processor in a high- level language
  - Most importantly, Brook programs were
    - Easier to write than hand-tuned GPU code
    - Seven times faster than similar existing code

# CUDA – Compute Unified Device Architecture

- NVIDIA invited Ian Buck to join the company.
  - Started evolving a solution to seamlessly run C on the GPU.
  - Putting the software and hardware together, NVIDIA unveiled CUDA in 2006
  - CUDA was launched in 2007.
  - The world's first solution for general-computing on GPUs
  - CUDA:
    - is a parallel computing architecture and programming model.
    - Includes C/C++ compiler and also support for OpenCL, DirectCompute.

# General Structure of the GPU Program in CUDA

- **Host Program – Executed by the CPU.**
  - This is a serial code.
  - Sets up the parameters for GPU (kernel) execution.
- **Kernel Program – Executed in Parallel by the SIMD cores** (Streaming Processors) in the GPU.

# Compiling CUDA Program:

# CUDA Execution Model

- Threads :
  - perform computations. They run on Scalar Processor (Streaming Processors) in GPU.
  - Thousands are needed to get full efficiency.
- Blocks :
  - Group of Threads. Max. Number of Threads vary from 1 to 1024.
  - They are alloted to Streaming Multiprocessors (SMs) in GPU.
  - Multiple blocks can reside in one SM.
- Grid :
  - Group of Blocks.
  - Holds the complete computation task. They represent the Kernel.

# THANK YOU

Department of Information Technology
National Institute of Technology Karnataka, Surathkal

# IT301: INTRODUCTION TO CUDA

(Compute Unified Device Architecture)

**Dr Geetha V**
**Asst Professor Grade I**
**Department of IT,**
**NITK, Surathkal**

# OUTLINE

1. Introduction to GPU
2. Introduction to CUDA
3. Microarchitecture of GPU
4. CUDA Execution and Memory Model
5. Hello World Program
6. CUDA Device Variables

# 3. Microarchitecture of GPU

# GPU Terms:

- **Stream processing** -- Term used to denote processing of a stream of instructions operating in a data parallel fashion.
- **Stream Processors (SPs)** – the execution cores that will execute the stream. Each stream processor has compute resources such as register file, instruction scheduler
- **Streaming multiprocessors (SMs)** -- groups of streaming processors that shares control logic and cache.

# GPU Microarchitectures

Fermi Architecture (2010)

Fermi's 16 SM are positioned around a common L2 cache. Each SM is a vertical rectangular strip that contain an orange portion (scheduler and dispatch), a green portion (execution units), and light blue portions (register file and L1 cache).

https://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf

# GPU Microarchitectures

Fermi Architecture (2010)

https://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf

# What each SMs have..

- Memory operations are handled by a set of 16 load-store units ir each SM.
- A set of four Special Function Units (SFUs) is also available to handle transcendental and other special operations such as sin, cos, exp, and rcp (reciprocal)
- Along with the group of 16 load-store units and the four SFUs, there are four execution blocks per SM.
    - 16 + 16 Cores (2 blocks)
    - LD/ST
    - SFU

# A total of 32 instructions from one or two warps can be dispatched in each cycle to any two of the four execution blocks within a Fermi SM

# Kepler (2012)

PCI Express 3.0 Host Interface
GigaThread Engine
Memory Controller
Memory Controller
Memory Controller
Memory Controller
Memory Controller
Memory Controller
SMX
L2 Cache

# Fermi Versus Kepler

| Feature | Fermi (GTX 580) | Kepler (GTX 680) |
|---|---|---|
| Fabrication | 40 nm | 28 nm |
| Power Consumption | 244 Watts | 195 Watts |
| CUDA Cores | 512 | 1536 |
| Compute Capability | 2.1 | 2.1 |
| Streaming Multiprocessors (SM) | 16 | 8 |
| Cores per SM | 32 | 192 |
| Texture Units | 64 | 128 |
| Warp Schedulers per SM | 2 | 4 |
| L1 Cache per SM | 64 KB | 64 KB |
| L2 Cache | 768 KB | 512 KB |
| Memory Interface | 384 bits | 256 bits |
| Bus Type | PCIe 2.0 x16 | PCIe 3.0 x16 |
| Memory Type | GDDR5 | GDDR5 |
| Memory Bandiwdth | 192.2 GB/s | 192.2 GB/s |
| Core Speed | 772 MHz | 1002 MHz |

# Maxwell (2014)

PCI Express 3.0 Host Interface
GigaThread Engine
GPC
Raster Engine
SMM
L2 Cache
Memory Controller

# NVIDIA Kepler SM

# NVIDIA Maxwell SM

# MAXWELL "GM204"

## Top Level

- 5.2 Billion Transistors
- 2x performance vs GK104
- 16 SMM
- 2048 CUDA Cores
- 16 Geometry Units
- 128 Texture Units
- 64 ROP Units
- 256-bit GDDR5

# Pascal (2016)

PCI Express 3.0 Host Interface
GigaThread Engine
GPC
TPC
SM
Instruction Cache
Memory Controller
High Bandwidth Memory 2
L2 Cache
High-Speed Hub
NVLink

PCI Express 3.0 Host Interface
GigaThread Engine
GPC
Raster Engine
TPC
PolyMorph Engine
SM
L2 Cache
Memory Controller

# Pascal

## Unified Memory

Dramatically Lower Developer Effort

## Stacked Memory

4x Higher Bandwidth (~1 TB/s)
3x Larger Capacity
4x More Energy Efficient per bit

## NVLink

GPU high speed interconnect
80-200 GB/s
Supported in POWER & ARM64 CPUs

NVIDIA Confidential - NDA Required

**Fermi** : GTX480
GTX580

**Kepler** : GTX 680
GTX 780

GTX 780 Ti

**Maxwell**: GTX 980
GTX 980 Ti

**Pascal** : GTX 1080

| GeForce | GTX 480 | GTX 580 | GTX 680 | GTX 780 | GTX 780 Ti | GTX 980 | GTX 980 Ti | GTX 1080 |
|---|---|---|---|---|---|---|---|---|
| Codename | GF100 | GF100 | GK104 | GK110 | GK110 | GM204 | GM200 | GP104 |
| Fab (nm) | 40 | 40 | 28 | 28 | 28 | 28 | 28 | 16 |
| Transistors (Billion) | 3 | 3 | 3.54 | 7.08 | 7.08 | 5.2 | 8.0 | 7.2 |
| Die size (mm2) | 529 | 520 | 294 | 561 | 561 | 398 | 601 | 314 |
| CUDA Cores | 480 | 512 | 1536 | 2304 | 2880 | 2048 | 2816 | 2560 |
| TAU | 60 | 64 | 128 | 192 | 240 | 128 | 176 | 160 |
| ROP | 48 | 48 | 32 | 48 | 48 | 64 | 96 | 64 |
| Memory (MB) | 1536 | 1536 | 2048 | 3072 | 3072 | 4096 | 6144 | 8192 |
| Bus width (bit) | 384 | 384 | 256 | 384 | 384 | 256 | 384 | 256 |
| Bandwidth (GB/s) | 177.4 | 192.3 | 192.2 | 288.4 | 336.4 | 224 | 336 | 320 |
| Release date | Mar-10 | Nov-10 | Mar-12 | May-13 | Nov-13 | Oct-14 | Jun-15 | Jun-16 |
| Price at release | $500 | $500 | $500 | $650 | $700 | $550 | $650 | $600 |

Source : https://www.techspot.com/article/1191-nvidia-geforce-six-generations-tested/

# Volta micro architecture (2017)

SM
L1 Instruction Cache
L0 Instruction Cache
Warp Scheduler (32 thread/clk)
Dispatch Unit (32 thread/clk)
Register File (16,384 x 32-bit)
FP64
INT
FP32
TENSOR CORE
LD/ ST
SFU
128KB L1 Data Cache / Shared Memory

# NVIDIA Tesla GPUs

| Tesla Product | Tesla K40 | Tesla M40 | Tesla P100 | Tesla V100 |
|---|---|---|---|---|
| GPU | GK180 (Kepler) | GM200 (Maxwell) | GP100 (Pascal) | GV100 (Volta) |
| SMs | 15 | 24 | 56 | 80 |
| TPCs | 15 | 24 | 28 | 40 |
| FP32 Cores / SM | 192 | 128 | 64 | 64 |
| FP32 Cores / GPU | 2880 | 3072 | 3584 | 5120 |
| FP64 Cores / SM | 64 | 4 | 32 | 32 |
| FP64 Cores / GPU | 960 | 96 | 1792 | 2560 |
| Tensor Cores / SM | NA | NA | NA | 8 |
| Tensor Cores / GPU | NA | NA | NA | 640 |
| GPU Boost Clock | 810/875 MHz | 1114 MHz | 1480 MHz | 1530 MHz |
| Peak FP32 TFLOPS[1] | 5 | 6.8 | 10.6 | 15.7 |
| Peak FP64 TFLOPS[1] | 1.7 | .21 | 5.3 | 7.8 |
| Peak Tensor TFLOPS[1] | NA | NA | NA | 125 |
| Texture Units | 240 | 192 | 224 | 320 |
| Memory Interface | 384-bit GDDR5 | 384-bit GDDR5 | 4096-bit HBM2 | 4096-bit HBM2 |
| Memory Size | Up to 12 GB | Up to 24 GB | 16 GB | 16 GB |
| L2 Cache Size | 1536 KB | 3072 KB | 4096 KB | 6144 KB |
| Shared Memory Size / SM | 16 KB/32 KB/48 KB | 96 KB | 64 KB | Configurable up to 96 KB |
| Register File Size / SM | 256 KB | 256 KB | 256 KB | 256KB |
| Register File Size / GPU | 3840 KB | 6144 KB | 14336 KB | 20480 KB |
| TDP | 235 Watts | 250 Watts | 300 Watts | 300 Watts |
| Transistors | 7.1 billion | 8 billion | 15.3 billion | 21.1 billion |
| GPU Die Size | 551 $mm^2$ | 601 $mm^2$ | 610 $mm^2$ | 815 $mm^2$ |
| Manufacturing Process | 28 nm | 28 nm | 16 nm FinFET+ | 12 nm FFN |

[1] Peak TFLOPS rates are based on GPU Boost Clock

# Accelerated Computing
# 5x Higher Energy Efficiency

**IBM POWER CPU**
Most Powerful Serial Processor

**NVIDIA NVLink**
Fastest CPU-GPU Interconnect

**NVIDIA Volta GPU**
Most Powerful Parallel Processor

# Rich Design Options for Next-Gen Servers

# THANK YOU

Department of Information Technology
National Institute of Technology Karnataka, Surathkal

# IT301: INTRODUCTION TO CUDA

(Compute Unified Device Architecture)

**Dr Geetha V**
**Asst Professor Grade I**
**Department of IT,**
**NITK, Surathkal**

# OUTLINE

1. Introduction to GPU
2. Introduction to CUDA
3. Microarchitecture of GPU
4. CUDA Execution and Memory Model
5. Hello World Program
6. CUDA Device Variables

# 4. CUDA Execution and Memory Model

# CUDA Memory Model

- **Local Memory** – Accessible only by a single thread in a thread block.
- **Shared Memory** – All threads in a single thread block have access to the shared memory.
- **Global Memory** – All thread blocks and Grids (of different Kernels) have access to global memory. Can be read or write.

# CUDA Memory Model

- **Local Memory – Accessible only by a single thread in a thread block.**
- **Shared Memory – All threads in a single thread block have access to the shared memory.**
- **Global Memory – All thread blocks and Grids (of different Kernels) have access to global memory. Can be read or write.**
- **Constant Memory** – All thread blocks have access to Constant memory. It is **read only memory**.
- **Texture Memory** – It is a type of Global memory which is **read only cache memory**. Accessible by all the thread blocks. High speed memory.

# Execution Flow of CUDA Program

## Step 1: Copy input data from CPU memory to GPU memory

## Step 2: Launch the kernel on GPU to process the data

## Step 3: Copy results back to CPU memory from GPU memory

# Executing CUDA programs in colabs

**Follow the instructions as given in following link**

https://www.geeksforgeeks.org/how-to-run-cuda-c-c-on-jupyter-notebook-in-google-colaboratory/

**The notebook will be ready for executing cuda programs.**

# Status information of device

```
%%cu
#include<stdio.h>
int main()
{
int devcount;
cudaGetDeviceCount(&devcount);
printf("Device count:%d\n",devcount);
for (int i = 0; i < devcount; ++i)
{
// Get device properties
printf("\nCUDA Device #%d\n", i);
cudaDeviceProp devProp;
cudaGetDeviceProperties(&devProp, i);
printf("Name:%s\n", devProp.name);
printf("Compute capability: %d.%d\n",devProp.major ,devProp.minor);
printf("Warp Size %d\n",devProp.warpSize);
printf("Total global memory:%u bytes\n",devProp.totalGlobalMem);
printf("Total shared memory per block: %u bytes\n",  devProp.sharedMemPerBlock);
printf("Total registers per block : %d\n",devProp.regsPerBlock);
printf("Clock rate: %d khz\n",devProp.clockRate);
printf("Maximum threads per block:%d\n", devProp.maxThreadsPerBlock);
for (int i = 0; i < 3; ++i)
printf("Maximum dimension %d of block: %d\n", i, devProp.maxThreadsDim[i]);
for (int i = 0; i <= 2; ++i)
printf("Maximum dimension %d of grid: %d\n", i, devProp.maxGridSize[i]);
printf("Number of multiprocessors:%d\n", devProp.multiProcessorCount);

}
return 0;
}
```

## Result of the program : ex1

```
Device count:1

CUDA Device #0
Name:Tesla K80
Compute capability: 3.7
Warp Size 32
Total global memory:3407020032 bytes
Total shared memory per block: 49152 bytes
Total registers per block : 65536
Clock rate: 823500 khz
Maximum threads per block:1024
Maximum dimension 0 of block: 1024
Maximum dimension 1 of block: 1024
Maximum dimension 2 of block: 64
Maximum dimension 0 of grid: 2147483647
Maximum dimension 1 of grid: 65535
Maximum dimension 2 of grid: 65535
Number of multiprocessors:13
```

# Status information of device

```
%%cu
#include<stdio.h>
int main()
{
int devcount;
cudaGetDeviceCount(&devcount);
printf("Device count:%d\n",devcount);
for (int i = 0; i < devcount; ++i)
{
// Get device properties
printf("\nCUDA Device #%d\n", i);
cudaDeviceProp devProp;
cudaGetDeviceProperties(&devProp, i);
printf("Name:%s\n", devProp.name);
printf("Compute capability: %d.%d\n",devProp.major ,devProp.minor);
printf("Warp Size %d\n",devProp.warpSize);
printf("Total global memory:%u bytes\n",devProp.totalGlobalMem);
printf("Total shared memory per block: %u bytes\n",  devProp.sharedMemPerBlock);
printf("Total registers per block : %d\n",devProp.regsPerBlock);
printf("Clock rate: %d khz\n",devProp.clockRate);
printf("Maximum threads per block:%d\n", devProp.maxThreadsPerBlock);
for (int i = 0; i < 3; ++i)
printf("Maximum dimension %d of block: %d\n", i, devProp.maxThreadsDim[i]);
for (int i = 0; i <= 2; ++i)
printf("Maximum dimension %d of grid: %d\n", i, devProp.maxGridSize[i]);
printf("Number of multiprocessors:%d\n", devProp.multiProcessorCount);

}
return 0;
}
```

## Result of the program : ex1

```
Device count:1

CUDA Device #0
Name:Tesla K80
Compute capability: 3.7
Warp Size 32
Total global memory:3407020032 bytes
Total shared memory per block: 49152 bytes
Total registers per block : 65536
Clock rate: 823500 khz
Maximum threads per block:1024
Maximum dimension 0 of block: 1024
Maximum dimension 1 of block: 1024
Maximum dimension 2 of block: 64
Maximum dimension 0 of grid: 2147483647
Maximum dimension 1 of grid: 65535
Maximum dimension 2 of grid: 65535
Number of multiprocessors:13
```

# Every programming introduction starts with Hello World!

```
int main(void)
{
printf("Hello World!\n");
return 0;
}
```

# Hello World in CUDA:

```
__global__ void helloworldKernel(void)
{

}

int main(void)
{
helloworldKernel<<<1,1>>>();
printf("Hello World!\n");
return 0;
}
```

- Keyword __global__ indicates a function that runs on device.It is called from host.
- helloworldKernel() is a device function processed by NVIDIA compiler.
- Host functions are processed by host compiler.
- helloworldKernel()<<<1,1>>>()

<<<num of blocks, num threads/block>>> launches the kernel. Two parameters indicate the number of blocks and number of threads per block.

- What is the output of this code?

# Hello World in CUDA:

```
__global__ void helloworldKernel(void)
{

}

int main(void)
{
helloworldKernel<<<1,1>>>();
printf("Hello World!\n");
return 0;
}
```

- Keyword __global__ indicates a function that runs on device.It is called from host.
- helloworldKernel() is a device function processed by NVIDIA compiler.
- Host functions are processed by host compiler.
- helloworldKernel()<<<1,1>>>()

<<<num of blocks, num threads/block>>> launches the kernel. Two parameters indicate the number of blocks and number of threads per block.

- What is the output of this code?

# Hello World by Parallel Threads

```
__global__ void helloworldKernel(void)
{
printf("Hello World\n");
}

int main(void)
{
helloworldKernel<<<1,1>>>();
printf("Hello World!\n");
return 0;
}
```

```
Hello World
```

- What do we expect from this code execution?
- How many times the Hello world will be printed?
- <<<1,1>>> number of threads

# Hello World by Parallel Threads

```
__global__ void helloworldKernel(void)
{
printf("Hello World\n");
}

int main(void)
{
helloworldKernel<<<1,10>>>();
printf("Hello World!\n");
return 0;
}
```

What is expected output?

- What do we expect from this code execution?
- How many times the Hello world will be printed?
- <<<1,10>>> number of threads

# Hello World by Parallel Threads

```
__global__ void helloworldKernel(void)
{
printf("Hello World\n");
}

int main(void)
{
helloworldKernel<<<1,10>>>();
printf("Hello World!\n");
return 0;
}
```

- What do we expect from this code execution?
- How many times the Hello world will be printed?
- <<<1,10>>> number of threads

What is expected output?
Printing hello world 10 +1 =11 times
But Actual output is `Hello World`

# Execution Flow of CUDA Program

## Step 1: Copy input data from CPU memory to GPU memory

## Step 2: Launch the kernel on GPU to process the data

## Step 3: Copy results back to CPU memory from GPU memory (I/O operation by CPU)

# Program for performing c[i]=a[i]+b[i]

Read the array content

Perform addition

Display result

# Execution Flow of CUDA Program

**Step 1: Copy input data from CPU memory to GPU memory**

- **Allocate Memory in GPU**

**malloc() and cudaMalloc()**

- **Copy data to GPU memory**

**cudaMemcpy()**

```
// Size of vectors

int n = 1000;

// Host input and output vectors (host => CPU)

float *h_a, *h_b, *h_c;

// Device input vectors (device => GPU)

float *d_a, *d_b, *d_c;
```

```
// Size, in bytes, of each vector
size_t bytes = n*sizeof(float);

// Allocate memory for each vector on host
h_a = (float*)malloc(bytes);
h_b = (float*)malloc(bytes);
h_c = (float*)malloc(bytes);

// Allocate memory for each vector on GPU
cudaMalloc(&d_a, bytes);
cudaMalloc(&d_b, bytes);
cudaMalloc(&d_c, bytes);
```

# Execution Flow of CUDA Program

**Step 1: Copy input data from CPU memory to GPU memory**

- **Allocate Memory in GPU**

**malloc() and cudaMalloc()**

- **Copy data to GPU memory**

**cudaMemcpy()**

```
int i;
   // Initialize vectors on host
   for( i = 0; i < n; i++ ) {
      h_a[i] = i+1;
      h_b[i] = i+1;
   }

   // Copy host vectors to device
   cudaMemcpy( d_a, h_a, bytes, cudaMemcpyHostToDevice);
   cudaMemcpy( d_b, h_b, bytes, cudaMemcpyHostToDevice);
```

# Execution Flow of CUDA Program

**Step 2: Launch the kernel on GPU to process the data**

- **Launch the kernel or task**

**kernel<<<blocks,threads>>>**

```
// Execute the kernel. n=1000.
vecAdd<<<1,n>>>(d_a, d_b, d_c, n);
 ..........

-----------------------------------------------

//Kernel program
__global__ void vecAdd(float *a, float *b,float *c, int n)
{
    int id = threadIdx.x;
   // Make sure we do not go out of bounds
   if (id < n)
      c[id] = a[id] + b[id];
}
```

# Execution Flow of CUDA Program

**Step 3: Copy results back to CPU memory from GPU memory**

**Copy output data to CPU**
**cudaMemcpy()**

```
// Copy array back to host
   cudaMemcpy( h_c, d_c, bytes, cudaMemcpyDeviceToHost );
    float  sum = 0;
   for(i=0; i<n; i++)
      sum += h_c[i];
   printf("final result: %f\n", sum);

// Release device memory
   cudaFree(d_a);
   cudaFree(d_b);
   cudaFree(d_c);
   // Release host memory
   free(h_a);
   free(h_b);
   free(h_c);
   return 0;
}
```

# Program for performing c[i]=a[i]+b[i]

```
%%cu
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
 __global__ void vecAdd(double *a, double *b, double *c, int n)
{
    // Get global thread ID
    int id = threadIdx.x;
     // Do not go out of bounds
    if (id < n)
        c[id] = a[id] + b[id];
}
 int main( int argc, char* argv[] )
{
    // Size of vectors
    int n = 100;
     // Host input vectors
    double *h_a, *h_b;
    //Host output vector
    double *h_c;
     // Device input vectors
    double *d_a, *d_b;
    //Device output vector
    double *d_c;
     // Size, in bytes, of each vector
    size_t bytes = n*sizeof(double);

    // Allocate memory for each vector on host
    h_a = (double*)malloc(bytes);
    h_b = (double*)malloc(bytes);
    h_c = (double*)malloc(bytes);
```

```
// Allocate memory for each vector on GPU
cudaMalloc(&d_a, bytes);
cudaMalloc(&d_b, bytes);
cudaMalloc(&d_c, bytes);
 int i;
// Initialize vectors on host
for( i = 0; i < n; i++ ) {
    h_a[i] = i+1;
    h_b[i] = i+1;
}

 // Copy host vectors to device
cudaMemcpy( d_a, h_a, bytes, cudaMemcpyHostToDevice);
cudaMemcpy( d_b, h_b, bytes, cudaMemcpyHostToDevice);


 // Execute the kernel
vecAdd<<<1,n>>>(d_a, d_b, d_c, n);
cudaDeviceSynchronize();
 // Copy array back to host
cudaMemcpy( h_c, d_c, bytes, cudaMemcpyDeviceToHost );


for(i=0; i<n; i=i+10)
   printf("c[%d]=%f\n",i,h_c[i]);
 // Release device memory
cudaFree(d_a);
cudaFree(d_b);
cudaFree(d_c);
 // Release host memory
free(h_a);
free(h_b);
```

# Program for performing c[i]=a[i]+b[i]

Results for

vecAdd<<<1,n>>>(d_a, d_b, d_c, n);

Only few elements are printed…

```
c[0]=2.000000
c[10]=22.000000
c[20]=42.000000
c[30]=62.000000
c[40]=82.000000
c[50]=102.000000
c[60]=122.000000
c[70]=142.000000
c[80]=162.000000
c[90]=182.000000
```

## Result of the program :

```
 // Execute the kernel
    vecAdd<<<2,50>>>(d_a, d_b, d_c, n);
 __global__ void vecAdd(double *a, double *b, doub
le *c, int n)
{
    // Get global thread ID
    int id = threadIdx.x;
     // Do not go out of bounds
    if (id < n)
        c[id] = a[id] + b[id];
}
```

```
c[0]=2.000000
c[10]=22.000000
c[20]=42.000000
c[30]=62.000000
c[40]=82.000000
c[50]=0.000000
c[60]=0.000000
c[70]=0.000000
c[80]=0.000000
c[90]=0.000000
```

# Program for performing c[i]=a[i]+b[i]

Results for

vecAdd<<<2,50>>>(d_a, d_b, d_c, n);

```
 __global__ void vecAdd(double *a, double *b, double *c, int n)
{
     // Get global thread ID
    int id = blockIdx.x*blockDim.x+threadIdx.x;

     // Do not go out of bounds
    if (id < n)
        c[id] = a[id] + b[id];
}
```

```
c[0]=2.000000
c[10]=22.000000
c[20]=42.000000
c[30]=62.000000
c[40]=82.000000
c[50]=102.000000
c[60]=122.000000
c[70]=142.000000
c[80]=162.000000
c[90]=182.000000
```

# Program for performing c[i]=a[i]+b[i]

Results for

**vecAdd<<<100,1>>>(d_a, d_b, d_c, n);**

```
 __global__ void vecAdd(double *a, double *b, double *c, int n)
{
     // Get global thread ID
    int id = blockIdx.x*blockDim.x+threadIdx.x;

     // Do not go out of bounds
    if (id < n)
        c[id] = a[id] + b[id];
}
```

```
c[0]=2.000000
c[10]=22.000000
c[20]=42.000000
c[30]=62.000000
c[40]=82.000000
c[50]=102.000000
c[60]=122.000000
c[70]=142.000000
c[80]=162.000000
c[90]=182.000000
```

# 6. CUDA Device Variables

# Blocks in SMs

8 blocks of work can be assigned to GPUs with 2 SMs and GPU with 4 SMs as shown in the figure

More on Device Variables in next session……

# THANK YOU

# CUDA Device Variables

# Device Variables ...(1D data : 1Block)

```
%%cu
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
 __global__ void SingleLoop()
{
     //int id = blockIdx.x+blockIdx.x*blockDim.x;
     int idx = blockIdx.x*blockDim.x+threadIdx.x;
     int idy = blockIdx.y*blockDim.y+threadIdx.y;
     int id = idx + idy *blockDim.x;
     printf("GPU-i=%d Tx=%d Ty=%d Tz=%d Bx=%d By=%d Bz=%d\n",
    id,threadIdx.x,threadIdx.y, threadIdx.z, blockIdx.x,blockIdx.y, blockIdx.z);
   }
 int main(int argc, char **argv)
 {
     for(int i=0;i<16;i++){
          printf("CPU-i=%d\n",i);
     }
  dim3 grid(1,1,1);
  dim3 block(16,1,1);
   printf("..................\n");
  SingleLoop <<<grid, block>>>();
  cudaDeviceSynchronize();
  return 0;

 }
```

- 16 single dimention

```
CPU-i=0
CPU-i=1
CPU-i=2
CPU-i=3
CPU-i=4
CPU-i=5
CPU-i=6
CPU-i=7
CPU-i=8
CPU-i=9
CPU-i=10
CPU-i=11
CPU-i=12
CPU-i=13
CPU-i=14
CPU-i=15
..................
GPU-i=0 Tx=0 Ty=0 Tz=0 Bx=0 By=0 Bz=0
GPU-i=1 Tx=1 Ty=0 Tz=0 Bx=0 By=0 Bz=0
GPU-i=2 Tx=2 Ty=0 Tz=0 Bx=0 By=0 Bz=0
GPU-i=3 Tx=3 Ty=0 Tz=0 Bx=0 By=0 Bz=0
GPU-i=4 Tx=4 Ty=0 Tz=0 Bx=0 By=0 Bz=0
GPU-i=5 Tx=5 Ty=0 Tz=0 Bx=0 By=0 Bz=0
GPU-i=6 Tx=6 Ty=0 Tz=0 Bx=0 By=0 Bz=0
GPU-i=7 Tx=7 Ty=0 Tz=0 Bx=0 By=0 Bz=0
GPU-i=8 Tx=8 Ty=0 Tz=0 Bx=0 By=0 Bz=0
GPU-i=9 Tx=9 Ty=0 Tz=0 Bx=0 By=0 Bz=0
GPU-i=10 Tx=10 Ty=0 Tz=0 Bx=0 By=0 Bz=0
GPU-i=11 Tx=11 Ty=0 Tz=0 Bx=0 By=0 Bz=0
GPU-i=12 Tx=12 Ty=0 Tz=0 Bx=0 By=0 Bz=0
GPU-i=13 Tx=13 Ty=0 Tz=0 Bx=0 By=0 Bz=0
GPU-i=14 Tx=14 Ty=0 Tz=0 Bx=0 By=0 Bz=0
GPU-i=15 Tx=15 Ty=0 Tz=0 Bx=0 By=0 Bz=0
```

# Device Variables ...(1D data : 1Block)

16 data element : single dimention array

```
dim3 grid(1,1,1);
dim3 block(16,1,1);

//__global__

int idx = blockIdx.x*blockDim.x+threadIdx.x;
int idy = blockIdx.y*blockDim.y+threadIdx.y;
int id = idx + idy *blockDim.x;
```

**Block representation:**

Data Array

| | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| T0 | T1 | T2 | T3 | T4 | T5 | T6 | T7 | T8 | T9 | T10 | T11 | T12 | T13 | T14 | T15 |

| | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| T0 | T1 | T2 | T3 | T4 | T5 | T6 | T7 | T8 | T9 | T10 | T11 | T12 | T13 | T14 | T15 |

Block(0,0,0)

- 16 single dimention array

```
GPU-i=0 Tx=0 Ty=0 Tz=0 Bx=0 By=0 Bz=0
GPU-i=1 Tx=1 Ty=0 Tz=0 Bx=0 By=0 Bz=0
GPU-i=2 Tx=2 Ty=0 Tz=0 Bx=0 By=0 Bz=0
GPU-i=3 Tx=3 Ty=0 Tz=0 Bx=0 By=0 Bz=0
GPU-i=4 Tx=4 Ty=0 Tz=0 Bx=0 By=0 Bz=0
GPU-i=5 Tx=5 Ty=0 Tz=0 Bx=0 By=0 Bz=0
GPU-i=6 Tx=6 Ty=0 Tz=0 Bx=0 By=0 Bz=0
GPU-i=7 Tx=7 Ty=0 Tz=0 Bx=0 By=0 Bz=0
GPU-i=8 Tx=8 Ty=0 Tz=0 Bx=0 By=0 Bz=0
GPU-i=9 Tx=9 Ty=0 Tz=0 Bx=0 By=0 Bz=0
GPU-i=10 Tx=10 Ty=0 Tz=0 Bx=0 By=0 Bz=0
GPU-i=11 Tx=11 Ty=0 Tz=0 Bx=0 By=0 Bz=0
GPU-i=12 Tx=12 Ty=0 Tz=0 Bx=0 By=0 Bz=0
GPU-i=13 Tx=13 Ty=0 Tz=0 Bx=0 By=0 Bz=0
GPU-i=14 Tx=14 Ty=0 Tz=0 Bx=0 By=0 Bz=0
GPU-i=15 Tx=15 Ty=0 Tz=0 Bx=0 By=0 Bz=0
```

# Device Variables ...(1D data : 1Block)

16 data element : single dimention array

```
dim3 grid(1,1,1);
dim3 block(4,4,1);

//__global__

int idx = blockIdx.x*blockDim.x+threadIdx.x;
int idy = blockIdx.y*blockDim.y+threadIdx.y;
int id = idx + idy *blockDim.x;
```

Data Array

| | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| T0 | T1 | T2 | T3 | T4 | T5 | T6 | T7 | T8 | T9 | T10 | T11 | T12 | T13 | T14 | T15 |

| | | | |
|---|---|---|---|
| T0 (0,0,0) | T1 (1,0,0) | T2 (2,0,0) | T3 (3,0,0) |
| T4 (0,1,0) | T5 (1,1,0) | T6 (2,1,0) | T7 (3,1,0) |
| T8 (0,2,0) | T9 (1,2,0) | T10 (2,2,0) | T11 (3,2,0) |
| T12 (0,3,0) | T13 (1,3,0) | T14 (2,3,0) | T15 (3,3,0) |

Block(0,0,0)

- 16 single dimention array

```
GPU-i=0 Tx=0 Ty=0 Tz=0 Bx=0 By=0 Bz=0
GPU-i=1 Tx=1 Ty=0 Tz=0 Bx=0 By=0 Bz=0
GPU-i=2 Tx=2 Ty=0 Tz=0 Bx=0 By=0 Bz=0
GPU-i=3 Tx=3 Ty=0 Tz=0 Bx=0 By=0 Bz=0
GPU-i=4 Tx=0 Ty=1 Tz=0 Bx=0 By=0 Bz=0
GPU-i=5 Tx=1 Ty=1 Tz=0 Bx=0 By=0 Bz=0
GPU-i=6 Tx=2 Ty=1 Tz=0 Bx=0 By=0 Bz=0
GPU-i=7 Tx=3 Ty=1 Tz=0 Bx=0 By=0 Bz=0
GPU-i=8 Tx=0 Ty=2 Tz=0 Bx=0 By=0 Bz=0
GPU-i=9 Tx=1 Ty=2 Tz=0 Bx=0 By=0 Bz=0
GPU-i=10 Tx=2 Ty=2 Tz=0 Bx=0 By=0 Bz=0
GPU-i=11 Tx=3 Ty=2 Tz=0 Bx=0 By=0 Bz=0
GPU-i=12 Tx=0 Ty=3 Tz=0 Bx=0 By=0 Bz=0
GPU-i=13 Tx=1 Ty=3 Tz=0 Bx=0 By=0 Bz=0
GPU-i=14 Tx=2 Ty=3 Tz=0 Bx=0 By=0 Bz=0
GPU-i=15 Tx=3 Ty=3 Tz=0 Bx=0 By=0 Bz=0
```

# Device Variables ...(1D data : 1Block)

16 data element : single dimention array

```
dim3 grid(1,1,1);
dim3 block(8,2,1);

//__global__

int idx = blockIdx.x*blockDim.x+threadIdx.x;
int idy = blockIdx.y*blockDim.y+threadIdx.y;
int id = idx + idy *blockDim.x;
```

**Block representation:**

Data Array

| | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| T0 | T1 | T2 | T3 | T4 | T5 | T6 | T7 | T8 | T9 | T10 | T11 | T12 | T13 | T14 | T15 |

GPU Threads organization

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| T0 (0,0,0) | T1 (1,0,0) | T2 (2,0,0) | T3 (3,0,0) | T4 (4,0,0) | T5 (5,0,0) | T6 (6,0,0) | T7 (7,0,0) |
| T8 (0,1,0) | T9 (1,1,0) | T10 (2,1,0) | T11 (3,1,0) | T12 (4,1,0) | T13 (5,1,0) | T14 (6,1,0) | T15 (7,1,0) |

Block(0,0,0)

- 16 single dimention array

```
GPU-i=0 Tx=0 Ty=0 Tz=0 Bx=0 By=0 Bz=0
GPU-i=1 Tx=1 Ty=0 Tz=0 Bx=0 By=0 Bz=0
GPU-i=2 Tx=2 Ty=0 Tz=0 Bx=0 By=0 Bz=0
GPU-i=3 Tx=3 Ty=0 Tz=0 Bx=0 By=0 Bz=0
GPU-i=4 Tx=4 Ty=0 Tz=0 Bx=0 By=0 Bz=0
GPU-i=5 Tx=5 Ty=0 Tz=0 Bx=0 By=0 Bz=0
GPU-i=6 Tx=6 Ty=0 Tz=0 Bx=0 By=0 Bz=0
GPU-i=7 Tx=7 Ty=0 Tz=0 Bx=0 By=0 Bz=0
GPU-i=8 Tx=0 Ty=1 Tz=0 Bx=0 By=0 Bz=0
GPU-i=9 Tx=1 Ty=1 Tz=0 Bx=0 By=0 Bz=0
GPU-i=10 Tx=2 Ty=1 Tz=0 Bx=0 By=0 Bz=0
GPU-i=11 Tx=3 Ty=1 Tz=0 Bx=0 By=0 Bz=0
GPU-i=12 Tx=4 Ty=1 Tz=0 Bx=0 By=0 Bz=0
GPU-i=13 Tx=5 Ty=1 Tz=0 Bx=0 By=0 Bz=0
GPU-i=14 Tx=6 Ty=1 Tz=0 Bx=0 By=0 Bz=0
GPU-i=15 Tx=7 Ty=1 Tz=0 Bx=0 By=0 Bz=0
```

# Device Variables ...(1D data : 1Block)

16 data element : single dimention array

```
dim3 grid(1,1,1);
dim3 block(4,2,2);

//__global__

  int idx = blockIdx.x*blockDim.x+threadIdx.x;
  int idy = blockIdx.y*blockDim.y+threadIdx.y;
  int idz = blockIdx.z*blockDim.z+threadIdx.z;
    int id = idx + idy *blockDim.x+idz*blockDim.x*blockDim.y;
```

**Block representation:**

Data Array

| | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| T0 | T1 | T2 | T3 | T4 | T5 | T6 | T7 | T8 | T9 | T10 | T11 | T12 | T13 | T14 | T15 |

GPU Threads organization

| | | | |
|---|---|---|---|
| T0 (0,0,0) | T1 (1,0,0) | T2 (2,0,0) | T3 (3,0,0) |
| T4 (0,1,0) | T5 (1,1,0) | T6 (2,1,0) | T7 (3,1,0) |

threadIdx.z=0

| | | | |
|---|---|---|---|
| T8 (0,0,0) | T9 (1,0,0) | T10 (2,0,0) | T11 (3,0,0) |
| T12 (0,1,0) | T13 (1,1,0) | T14 (2,1,0) | T15 (3,1,0) |

ThreadIdx.z=1

Block(0,0,0)

- 16 single dimention array

```
GPU-i=0 Tx=0 Ty=0 Tz=0 Bx=0 By=0 Bz=0
GPU-i=1 Tx=1 Ty=0 Tz=0 Bx=0 By=0 Bz=0
GPU-i=2 Tx=2 Ty=0 Tz=0 Bx=0 By=0 Bz=0
GPU-i=3 Tx=3 Ty=0 Tz=0 Bx=0 By=0 Bz=0
GPU-i=4 Tx=0 Ty=1 Tz=0 Bx=0 By=0 Bz=0
GPU-i=5 Tx=1 Ty=1 Tz=0 Bx=0 By=0 Bz=0
GPU-i=6 Tx=2 Ty=1 Tz=0 Bx=0 By=0 Bz=0
GPU-i=7 Tx=3 Ty=1 Tz=0 Bx=0 By=0 Bz=0
GPU-i=8 Tx=0 Ty=0 Tz=1 Bx=0 By=0 Bz=0
GPU-i=9 Tx=1 Ty=0 Tz=1 Bx=0 By=0 Bz=0
GPU-i=10 Tx=2 Ty=0 Tz=1 Bx=0 By=0 Bz=0
GPU-i=11 Tx=3 Ty=0 Tz=1 Bx=0 By=0 Bz=0
GPU-i=12 Tx=0 Ty=1 Tz=1 Bx=0 By=0 Bz=0
GPU-i=13 Tx=1 Ty=1 Tz=1 Bx=0 By=0 Bz=0
GPU-i=14 Tx=2 Ty=1 Tz=1 Bx=0 By=0 Bz=0
GPU-i=15 Tx=3 Ty=1 Tz=1 Bx=0 By=0 Bz=0
```

# Device Variables ...(1D data : 1Block)

## 16 single dimension array

```
%%cu
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
 __global__ void TwoNestedLoops()
{
     int i = blockIdx.x*blockDim.x+threadIdx.x;
     int j = blockIdx.y*blockDim.y+threadIdx.y;
    printf("GPU-(%d,%d),Tx=%d Ty=%d Tz=%d Bx=%d By=%d Bz=%d\n",
           i,j,threadIdx.x,threadIdx.y, threadIdx.z, blockIdx.x,blockIdx.y, blockIdx.z);
   }
 int main(int argc, char **argv)
 {
     for(int i=0;i<4;i++){
         for(int j=0;j<4;j++){
          printf("CPU-(%d,%d)\n",i,j);
     }
     }
  dim3 grid(1,1,1);
  dim3 block(4,4,1);
   printf("..................\n");
  TwoNestedLoops <<<grid, block>>>();
  cudaDeviceSynchronize();
  return 0;

 }
```

```
CPU-(0,0)
CPU-(0,1)
CPU-(0,2)
CPU-(0,3)
CPU-(1,0)
CPU-(1,1)
CPU-(1,2)
CPU-(1,3)
CPU-(2,0)
CPU-(2,1)
CPU-(2,2)
CPU-(2,3)
CPU-(3,0)
CPU-(3,1)
CPU-(3,2)
CPU-(3,3)
..................
GPU-(0,0),Tx=0 Ty=0 Tz=0 Bx=0 By=0 Bz=0
GPU-(1,0),Tx=1 Ty=0 Tz=0 Bx=0 By=0 Bz=0
GPU-(2,0),Tx=2 Ty=0 Tz=0 Bx=0 By=0 Bz=0
GPU-(3,0),Tx=3 Ty=0 Tz=0 Bx=0 By=0 Bz=0
GPU-(0,1),Tx=0 Ty=1 Tz=0 Bx=0 By=0 Bz=0
GPU-(1,1),Tx=1 Ty=1 Tz=0 Bx=0 By=0 Bz=0
GPU-(2,1),Tx=2 Ty=1 Tz=0 Bx=0 By=0 Bz=0
GPU-(3,1),Tx=3 Ty=1 Tz=0 Bx=0 By=0 Bz=0
GPU-(0,2),Tx=0 Ty=2 Tz=0 Bx=0 By=0 Bz=0
GPU-(1,2),Tx=1 Ty=2 Tz=0 Bx=0 By=0 Bz=0
GPU-(2,2),Tx=2 Ty=2 Tz=0 Bx=0 By=0 Bz=0
GPU-(3,2),Tx=3 Ty=2 Tz=0 Bx=0 By=0 Bz=0
GPU-(0,3),Tx=0 Ty=3 Tz=0 Bx=0 By=0 Bz=0
GPU-(1,3),Tx=1 Ty=3 Tz=0 Bx=0 By=0 Bz=0
GPU-(2,3),Tx=2 Ty=3 Tz=0 Bx=0 By=0 Bz=0
GPU-(3,3),Tx=3 Ty=3 Tz=0 Bx=0 By=0 Bz=0
```