VISVESVARAYA TECHNOLOGICAL UNIVERSITY BELGAUM

PARALLEL COMPUTING

(Subject Code: BCS702)

LECTURE NOTES

VII-SEMESTER

Dr. Lokesh M R
Professor, Dept of ISE

A J INSTITUTE OF ENGINEERING & TECHNOLOGY

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

(A unit of Laxmi Memorial Education Trust. (R))

NH - 66, KottaraChowki, Kodical Cross - 575 006

VISION AND MISSION STATEMENT OF AJIET

Vision of the Institute

To produce top-quality engineers who are groomed for attaining excellence in their profession and competitive enough to help in the growth of nation and global society.

Mission of the Institute

- To offer affordable high-quality graduate program in engineering with value education and make the students socially responsible.
- To support and enhance the institutional environment to attain research excellence in both faculty and students and to inspire them to push the boundaries of knowledge base.
- To identify the common areas of interest amongst the individuals for the effective industry-institute partnership in a sustainable way by systematically working together.
- To promote the entrepreneurial attitude and inculcate innovative ideas among the engineering professionals.

Department of Computer Science and Engineering

Vision of the Department

To adapt the evolutionary changes in computer science and expose the students to the cutting-edge technologies to produce globally competent professionals.”.

Mission of the Department

M 1. To ensure holistic professional education in the field of computer science & engineering and produce efficient industry ready IT graduates capable of solving societal problems to be a part of nation building.

M 2. To provide an inspirational atmosphere in the department to nurture research and innovation capabilities among students and faculties making them good innovators and visionaries.

M 3. To establish a vibrant Industry-Academic relationships and collaborations among the individuals having the opportunities to work on the latesttechnologies and professional challenges with integrity.

M 4. To cultivate students with professional skills, foster innovative research endeavors, and cultivate entrepreneurial capabilities.

PROGRAM EDUCATIONAL OBJECTIVES (PEOs)	
PEO1	To develop in students, the ability to solve real life problems by applying fundamental science and elementary strengths of computer science courses
PEO2	To mould students, to have a successful career in the IT industry where graduates will be able to design and implement the needs of society and nation.
PEO3	To evolve the students as a potent team player and to emerge as a leader and hence to enhance their contribution to the technical undertaken work in a significant manner.
PEO4	To transform students, to excel in a competitive world through higher education and indulge in research through continuous learning process.

PROGRAM OUTCOMES (POs)	
PO1	**Engineering knowledge**: Apply the knowledge of mathematics, science, engineering fundamentals, and specialization in Mechanical Engineering for the solution of complex engineering problems.
PO2	**Problem analysis**: Identify, formulate, research literature, and analyse complex engineering problems reaching substantiated conclusions using first principles of mathematics, natural sciences, and engineering sciences.
PO3	**Design/development of solutions:** Design solutions for complex engineering problems and design system components or processes that meet the specified needs with appropriate consideration for public health and safety, and cultural, societal, and environmental considerations.
PO4	**Conduct investigations of complex problems:** Use research-based knowledge and research methods including design of experiments, analysis and interpretation of data, and synthesis of the information to provide valid conclusions.
PO5	**Modern tool usage:** Create, select, and apply appropriate techniques, resources, and modern engineering and IT tools, including prediction and modelling to complex engineering activities, with an understanding of the limitations.
PO6	**The engineer and society:** Apply reasoning informed by the contextual knowledge to assess societal, health, safety, legal, and cultural issues and the consequent responsibilities relevant to the professional engineering practice
PO7	**Environment and sustainability**: Understand the impact of the professional engineering solutions in societal and environmental contexts, and demonstrate the knowledge of, and need for sustainable development.
PO8	**Ethics:** Apply ethical principles and commit to professional ethics and responsibilities and norms of the engineering practice.
PO9	**Individual and team work:** Function effectively as an individual, and as a member or leader in diverse teams, and in multidisciplinary settings.
PO10	**Communication**: Communicate effectively on complex engineering activities with the engineering community and with the society at large, such as, being able

	to comprehend and write effective reports and design documentation, make effective presentations, and give and receive clear instructions.
PO11	**Project Management and Finance:** Demonstrate knowledge and understanding of the engineering and management principles and apply these to one's own work, as a member and leader in a team, to manage projects and in multidisciplinary environment.
PO12	**Life-long learning:** Recognize the need for, and have the preparation and ability to engage in independent and life-long learning in the broadest context of technological change.

PROGRAM SPECIFIC OUTCOMES (PSOs)	
PSO1	Apply engineering principles, professional ethics and fundamental science in designing systems and communication models (protocols).
PSO2	Design and develop smart and intelligent based applications in computational environment.

COURSE SYLLABUS

Course outcomes

At the end of the course the student will be able to:

CO1	Explore the need for parallel programming
CO2	Explain how to parallelize on MIMD systems
CO3	To demonstrate how to apply MPI library and parallelize the suitable programs
CO4	To demonstrate how to apply OpenMP pragma and directives to parallelize the suitable programs
CO5	To demonstrate how to design CUDA program

Suggested Learning Resources:

1. **Text Books**

1. . Peter S Pacheco, Matthew Malensek – An Introduction to Parallel Programming, second edition, Morgan Kauffman.
2. Michael J Quinn – Parallel Programming in C with MPI and OpenMp, McGrawHill

2. Reference Books

1. Calvin Lin, Lawrence Snyder – Principles of Parallel Programming, Pearson
2. Barbara Chapman – Using OpenMP: Portable Shared Memory Parallel Programming, Scientific and Engineering Computation
3. William Gropp, Ewing Lusk – Using MPI:Portable Parallel Programing, Third edition, Scientific and Engineering Computation

3. **Web links and Video Lectures (e-Resources):**

1. Introduction to parallel programming: https://nptel.ac.in/courses/106102163

1.8 VTU Syllabus

PARALLEL COMPUTING **(Effective from the academic year 2024 -2025)** **SEMESTER– VII**			
Course Code	BCS702	**CIE Marks**	50
Teaching Hours/Week (L:T:P: S)	3:0:2:0	**SEE Marks**	50
Total Hours of Pedagogy	40 hours Theory + 8-10 Lab slots	Total Marks	100
Credits	04	**Exam Hours**	03
Examination nature (SEE)	Theory/Practical		
Course Learning Objectives: This course (BCS702) will enable students to:			
• Explore the need for parallel programming • Explain how to parallelize on MIMD systems • To demonstrate how to apply MPI library and parallelize the suitable programs • To demonstrate how to apply OpenMP pragma and directives to parallelize the suitable programs To demonstrate how to design CUDA program			
Teaching-Learning Process (General Instructions) These are sample Strategies that teachers can use to accelerate the attainment of the various course outcomes. 1. Lecturer method (L) need not to be only traditional lecture methods, but alternative effective teaching methods could be adopted to attain the outcomes. 2. Use of Video/Animation to explain functioning of various concepts. 3. Encourage collaborative (Group Learning) Learning in the class. 4. Ask at least three HOT (Higher order Thinking) questions in the class, which promotes critical thinking. 5. Adopt Programming assignment, which fosters student's Analytical skills, develop design thinking skills such as the ability to design, evaluate, generalize, and analyze information rather than simply recall it.			

Module 1	**Contact Hours**
Introduction to parallel programming, Parallel hardware and parallel software – Classifications of parallel computers, SIMD systems, MIMD systems, Interconnection networks, Cache coherence, Shared-memory vs. distributed-memory, Coordinating the processes/threads, Shared-memory, Distributed-memory.	8
Module 2	
GPU programming, Programming hybrid systems, MIMD systems, GPUs, Performance – Speedup and efficiency in MIMD systems, Amdahl's law, Scalability in MIMD systems, Taking timings of MIMD programs, GPU performance.	8
Module 3	
Distributed memory programming with MPI – MPI functions, The trapezoidal rule in MPI, Dealing with I/O, Collective communication, MPI-derived datatypes, Performance evaluation of MPI programs, A parallel sorting algorithm.	8
Module 4	
Shared-memory programming with OpenMP – openmp pragmas and directives, The trapezoidal rule, Scope of variables, The reduction clause, loop carried dependency, scheduling, producers and consumers, Caches, cache coherence and false sharing in openmp, tasking, tasking, thread safety.	8

Module 5	
GPU programming with CUDA - GPUs and GPGPU, GPU architectures, Heterogeneous computing, Threads, blocks, and grids Nvidia compute capabilities and device architectures, Vector addition, Returning results from CUDA kernels, CUDA trapezoidal rule I, CUDA trapezoidal rule II: improving performance, CUDA trapezoidal rule III: blocks with more than one warp.	8

Course Outcomes(Course Skill Set)**:** At the end of the course, the student will be able to:

- Explain the need for parallel programming
- Demonstrate parallelism in MIMD system.
- Apply MPI library to parallelize the code to solve the given problem.
- Apply OpenMP pragma and directives to parallelize the code to solve the given problem
- Design a CUDA program for the given problem.

Assessment Details (both CIE and SEE)

The weightage of Continuous Internal Evaluation (CIE) is 50% and for Semester End Exam (SEE) is 50%. The minimum passing mark for the CIE is 40% of the maximum marks (20 marks out of 50) and for the SEE minimum passing mark is 35% of the maximum marks (18 out of 50 marks). A student shall be deemed to have satisfied the academic requirements and earned the credits allotted to each subject/ course if the student secures a minimum of 40% (40 marks out of 100) in the sum total of the CIE (Continuous Internal Evaluation) and SEE (Semester End Examination) taken together.

CIE for the theory component of the IPCC (maximum marks 50)

- IPCC means practical portion integrated with the theory of the course.
- CIE marks for the theory component are 25 marks and that for the practical component is 25 marks.
- 25 marks for the theory component are split into 15 marks for two Internal Assessment Tests (Two Tests, each of 15 Marks with 01-hour duration, are to be conducted) and 10 marks for other assessment methods mentioned in 22OB4.2. The first test at the end of 40-50% coverage of the syllabus and the second test after covering 85-90% of the syllabus.
- Scaled-down marks of the sum of two tests and other assessment methods will be CIE marks for the theory component of IPCC (that is for 25 marks).
- The student has to secure 40% of 25 marks to qualify in the CIE of the theory component of IPCC.

CIE for the practical component of the IPCC

- 15 marks for the conduction of the experiment and preparation of laboratory record, and 10 marks for the test to be conducted after the completion of all the laboratory sessions.
- On completion of every experiment/program in the laboratory, the students shall be evaluated including viva-voce and marks shall be awarded on the same day.
- The CIE marks awarded in the case of the Practical component shall be based on the continuous evaluation of the laboratory report. Each experiment report can be evaluated for 10 marks. Marks of all experiments' write-ups are added and scaled down to 15 marks.
- The laboratory test (duration 02/03 hours) after completion of all the experiments shall be conducted for 50 marks and scaled down to 10 marks.
- Scaled-down marks of write-up evaluations and tests added will be CIE marks for the laboratory component of IPCC for 25 marks.
- The student has to secure 40% of 25 marks to qualify in the CIE of the practical component of the IPCC.

The theory portion of the IPCC shall be for both CIE and SEE, whereas the practical portion will have a CIE component only. Questions mentioned in the SEE paper may include questions from the practical component.
SEE for IPCC :Question Paper Pattern:
Theory SEE will be conducted by University as per the scheduled timetable, with common question papers for the course (duration 03 hours) 1. The question paper will have ten questions. Each question is set for 20 marks. 2. There will be 2 questions from each module. Each of the two questions under a module (with a maximum of 3 sub-questions), should have a mix of topics under that module. 3. The students have to answer 5 full questions, selecting one full question from each module. 4. Marks scored by the student shall be proportionally scaled down to 50 Marks
Textbooks: Suggested Learning Resources:
1. Textbook: 1. Peter S Pacheco, Matthew Malensek – An Introduction to Parallel Programming, second edition, Morgan Kauffman. 2. Michael J Quinn – Parallel Programming in C with MPI and OpenMp, McGrawHill.
Reference Books:
1. Calvin Lin, Lawrence Snyder – Principles of Parallel Programming, Pearson 2. Barbara Chapman – Using OpenMP: Portable Shared Memory Parallel Programming, Scientific and Engineering Computation 3. William Gropp, Ewing Lusk – Using MPI:Portable Parallel Programing, Third edition, Scientific and Engineering Computation
Web links and Video Lectures (e-Resources):
1. Introduction to parallel programming: https://nptel.ac.in/courses/106102163
Activity Based Learning (Suggested Activities in Class)/ Practical Based learning
▪ Programming Assignment at higher bloom level (10 Marks)

PARALLEL COMPUTINGLABORATORY **(Effective from the academic year 2024 -2025)** **SEMESTER– VII**			
Course Code	BCS702	**CIE Marks**	50
Teaching Hours/Week (L:T:P: S)	3:0:2:0	**SEE Marks**	50
Total Hours of Pedagogy	40 hours Theory + 8-10 Lab slots	Total Marks	100
PRACTICAL COMPONENT OF IPCC			
Course Learning Objectives: This course (18CSL38) will enable students to:			
This laboratory course enable students to get practical experience in design, develop, implement, analyze and evaluation/testing of • Explore the need for parallel programming • Explain how to parallelize on MIMD systems • To demonstrate how to apply MPI library and parallelize the suitable programs • To demonstrate how to apply OpenMP pragma and directives to parallelize the suitable programs ▯ To demonstrate how to design CUDA program			
Descriptions (if any):			
▯ Implement all the programs in „C / C++" Programming Language and Linux / Windows as OS.			
Programs List:			

1.	Write a OpenMP program to sort an array on n elements using both sequential and parallel mergesort(using Section). Record the difference in execution time.
2.	Write an OpenMP program that divides the Iterations into chunks containing 2 iterations, respectively (OMP_SCHEDULE=static,2). Its input should be the number of iterations, and its output should be which iterations of a parallelized for loop are executed by which thread. For example, if there are two threads and four iterations, the output might be the following: a. Thread 0 : Iterations 0 −− 1 b. Thread 1 : Iterations 2 −− 3
3.	Write a OpenMP program to calculate n Fibonacci numbers using tasks.
4.	Write a OpenMP program to find the prime numbers from 1 to n employing parallel for directive. Record both serial and parallel execution times.
5.	Write a MPI Program to demonstration of MPI_Send and MPI_Recv.
6	Write a MPI program to demonstration of deadlock using point to point communication and avoidance of deadlock by altering the call sequence
7	Write a MPI Program to demonstration of Broadcast operation.
8	Write a MPI Program demonstration of MPI_Scatter and MPI_Gather
9	Write a MPI Program to demonstration of MPI_Reduce and MPI_Allreduce (MPI_MAX, MPI_MIN, MPI_SUM, MPI_PROD)

Introduction to parallel programming, Parallel hardware and parallel software – Classifications of parallel computers, SIMD systems, MIMD systems, Interconnection networks, Cache coherence, Sharedmemory vs. distributed-memory, Coordinating the processes/threads, Shared-memory, Distributedmemory.

1 Module

Module -1 Lecturer Notes:

Introduction to parallel programming, Parallel hardware and parallel software

This Lecture Notes provides a comprehensive overview of Introduction to parallel programming, Parallel hardware and parallel software, based on Chapter 1 of "Peter S Pacheco, Matthew Malensek – An Introduction to Parallel Programming, second edition, Morgan Kauffman , VTU syllabus based BCS702 Parallel Computing" It is designed to serve as a detailed lecture presentation for a classroom setting, covering Introduction to parallel programming, Parallel hardware and parallel software, and practical examples, while ensuring clarity for students new to parallel computing.

MODULE-1: Introduction to parallel programming, Parallel hardware and parallel software –

Classifications of Parallel Computers,

1. MIMD systems,
2. Interconnection networks,
3. Cache coherence,
4. Shared-memory vs. distributed-memory,
5. Coordinating the processes/threads,
6. Shared-memory,
7. Distributed-memory

1 Classifications of Parallel Computers,

Parallel computers can be classified based on their architecture and processing style. Key classifications:

1. Flynn's Taxonomy:
   1. SISD (Single Instruction, Single Data): Sequential processing.
   2. SIMD (Single Instruction, Multiple Data): Same instruction on multiple data.
   3. MIMD (Multiple Instruction, Multiple Data): Multiple instructions on multiple data.
   4. MISD (Multiple Instruction, Single Data): Rarely used.
2. Shared Memory vs. Distributed Memory:
   1. Shared Memory: Multiple processors share common memory.
   2. Distributed Memory: Each processor has its own memory.
3. Architecture:
   1. Multicore: Multiple cores on a single chip.
   2. Multiprocessor: Multiple processors on multiple chips.
   3. Cluster: Multiple computers connected together.
4. Other classifications:
   1. GPU-accelerated: Using GPUs for parallel processing.
   2. FPGA-based: Using Field-Programmable Gate Arrays.

Understanding these classifications helps in designing and optimizing parallel systems for specific applications.

Q1: Explain Flynn's taxonomy for classifying parallel computers, including the types of systems it defines and their characteristics.

A1: Flynn's taxonomy classifies parallel computers based on the number of instruction streams and data streams they can handle simultaneously. It defines four categories, but the document focuses on three relevant to parallel systems:

1. **SISD (Single Instruction, Single Data):** This represents a classical von Neumann system, where a single instruction is executed on a single data item at a time. It is not inherently parallel and serves as the baseline for comparison.
2. **SIMD (Single Instruction, Multiple Data):** In SIMD systems, a single instruction is broadcast to multiple datapaths, each applying the instruction to different data items simultaneously. For example, in vector addition (x[i]+=y[i]), each datapath adds corresponding elements of arrays x and y. SIMD systems are synchronous, with all datapaths executing the same instruction or idling, making them ideal for data-parallel tasks like loop operations on large arrays. However, conditional operations can degrade performance if some datapaths must idle.
3. **MIMD (Multiple Instruction, Multiple Data):** MIMD systems support multiple independent instruction streams, each operating on different data streams. They consist of autonomous processors with their own control units and datapaths, operating asynchronously without a global clock. MIMD systems are versatile, supporting a wide range of parallel tasks, and are divided into shared memory and distributed memory systems based on memory access.

   Flynn's taxonomy highlights the degree of parallelism in instruction and data processing, guiding the design and programming of parallel systems.

Q2: Compare and contrast shared memory and distributed memory systems in terms of their architecture, communication methods, and suitability for different types of problems.

A2: Shared memory and distributed memory systems are two primary types of MIMD systems, distinguished by how cores access memory and coordinate tasks.

- **Architecture:**
  - **Shared Memory Systems:** All cores access a common memory space via an interconnection network (e.g., bus or crossbar). Multicore processors are common examples, with private L1 caches and possibly shared lower-level caches. They can be Uniform Memory Access (UMA), where all memory locations have equal access times, or Non-Uniform Memory Access (NUMA), where access times vary depending on memory location proximity.
  - **Distributed Memory Systems:** Each core has its own private memory, and processormemory pairs communicate over a network (e.g., toroidal mesh or hypercube).

Clusters, composed of commodity PCs connected via Ethernet, are typical examples, often called hybrid systems if nodes are shared-memory multicore systems.

- **Communication Methods:**
    - **Shared Memory Systems:** Cores communicate implicitly by reading and writing to shared memory locations. This simplifies programming, as data structures can be accessed directly, but requires mechanisms like cache coherence to ensure consistency.
    - **Distributed Memory Systems:** Cores communicate explicitly by sending messages or using functions to access another core's memory. This requires programmers to manage data exchange, which can be complex but avoids contention for shared resources.
- **Suitability:**
    - **Shared Memory Systems:** Ideal for problems where tasks frequently share data, as implicit communication is straightforward. They are easier to program but scale poorly due to interconnect costs (e.g., buses cause contention, and large crossbars are expensive). They suit smaller systems or applications like real-time graphics processing.
    - **Distributed Memory Systems:** Better for large-scale problems requiring vast computation or data, as their interconnects (e.g., hypercubes) are cost-effective and scalable to thousands of processors. They are common in scientific computing and big data applications but require more programming effort for communication.

In summary, shared memory systems offer programming simplicity but limited scalability, while distributed memory systems provide scalability at the cost of communication complexity.

Q3: Discuss the advantages and limitations of SIMD systems, using the vector addition example to illustrate your points.

A3: SIMD (Single Instruction, Multiple Data) systems execute a single instruction across multiple data streams simultaneously, offering significant advantages but also facing notable limitations. The vector addition example (x[i]+=y[i]) illustrates these points.

- **Advantages:**
    - **Efficiency for Data-Parallel Tasks:** SIMD systems excel at data-parallelism, where the same operation is applied to large datasets. In the vector addition example, if the system has n datapaths, each can load x[i] and y[i], perform the addition, and store the result in one cycle, achieving high throughput. For m<n datapaths, the operation processes blocks of m m m elements, still maintaining efficiency.

- **Simplified Control:** A single control unit broadcasts instructions, reducing complexity. In the example, one instruction to add is sent to all datapaths, eliminating the need for individual instruction management.
- **High Performance in Specific Domains:** SIMD is used in vector processors and GPUs, which handle large arrays efficiently. The document notes their resurgence in desktop CPUs and GPUs for tasks like graphics processing.

- **Limitations:**
  - **Performance Degradation with Conditionals:** SIMD requires all datapaths to execute the same instruction or idle, which can degrade performance. For instance, modifying the vector addition to only add if y[i]>0 (if(y[i]>0.0)x[i]+=y[i]) forces datapaths with non-positive y[i] to idle, reducing efficiency as some datapaths wait while others compute.
  - **Synchronous Operation:** Datapaths operate synchronously without instruction storage, so they cannot delay or reorder instructions. In the example, all additions must occur in lockstep, limiting flexibility for irregular tasks.
  - **Limited Applicability:** SIMD systems are less effective for non-data-parallel problems. The document notes their historical decline (e.g., Thinking Machines in the 1990s) due to poor performance on tasks requiring diverse operations, with only vector processors and GPUs maintaining relevance.

In the vector addition example, a SIMD system with n datapaths completes the task in one step if m=n, or in ⌈n/m⌉ steps if m<n, showcasing efficiency. However, introducing conditionals or irregular data access would highlight its limitations, as idle datapaths reduce performance. Thus, SIMD systems are powerful for regular, data-intensive tasks but struggle with complex, conditional, or irregular computations.

Q4: Why do distributed memory systems scale better than shared memory systems for largescale parallel computing, and what challenges do programmers face when using them?

A4: Distributed memory systems scale better than shared memory systems for large-scale parallel computing due to their interconnect architecture, but they pose significant programming challenges.

Reasons for Better Scalability:

- **Cost-Effective Interconnects:** Distributed memory systems use interconnects like toroidal meshes or hypercubes, which are relatively inexpensive compared to shared memory interconnects. For example, a toroidal mesh with p processors requires 2p links, while a large crossbar for shared memory requires p2 switches, making it prohibitively expensive for many processors.

- **Reduced Contention:** In shared memory systems, buses face increased contention as processors are added, slowing memory access. For instance, the document notes that buses are unsuitable for systems with many processors due to frequent conflicts. Distributed memory systems avoid this by giving each processor private memory, with communication handled via a network.

- **Support for Large Systems:** Distributed memory systems, such as clusters with thousands of nodes, are common in supercomputing. The document highlights that these systems, often hybrid with shared-memory nodes, scale to vast computational tasks, unlike shared memory systems, which are limited to fewer processors due to interconnect costs.

- **Programming Challenges:**
  - **Explicit Communication:** Unlike shared memory systems, where cores communicate implicitly via shared data, distributed memory systems require explicit message passing or special functions to access remote memory. Programmers must design communication protocols, as seen in the document's description of processor-memory pairs communicating over a network.
  - **Complexity in Coordination:** Coordinating tasks is more complex, as programmers must manage data distribution and synchronization. For example, in a distributed memory system, a task like vector addition requires dividing arrays across processors and exchanging results, increasing code complexity.
  - **Debugging and Optimization:** Explicit communication introduces challenges in debugging and optimizing performance. Network latency and bandwidth, as described in the document (message transmission time=l+n/b), must be considered, and programmers need to minimize communication overhead.

In conclusion, distributed memory systems scale better due to cost-effective, contention-free interconnects, enabling large-scale systems like clusters. However, programmers face challenges in managing explicit communication, coordination, and optimization, requiring more effort compared to the implicit communication of shared memory systems.

SIMD systems,

SIMD (Single Instruction, Multiple Data) systems are parallel computing architectures that execute the same instruction on multiple data elements simultaneously. Key features:

Advantages:

1. Improved performance: Increased throughput for specific tasks.
2. Efficient processing: Reduced instruction overhead.

Applications:

1. Graphics processing: Accelerating graphics rendering.
2. Scientific simulations: Efficiently processing large datasets.
3. Machine learning: Accelerating neural network computations.

Characteristics:

1. Data parallelism: Same operation on multiple data elements.
2. Lockstep execution: All processing units execute the same instruction.

Examples:

1. GPU architectures: Many modern GPUs are SIMD-based.
2. Vector processors: Designed for high-performance computing.

SIMD systems excel in applications with inherent data parallelism, making them a valuable tool for accelerating specific tasks.

Q1: Describe the architecture and operation of a SIMD system, using the vector addition example illustrate how it processes data.

A1: A SIMD (Single Instruction, Multiple Data) system is designed to execute a single instruction across multiple data streams simultaneously, making it ideal for data-parallel tasks. Architecturally, a SIMD system consists of a single control unit and multiple datapaths. The control unit broadcasts one instruction to all datapaths, which either apply it to their data or remain idle. This synchronous operation ensures uniformity but limits flexibility.

In the vector addition example, consider two arrays x and y, each with n elements, where the task is to compute x[i]+=y[i] for i=0 to n−1. If the SIMD system has n datapaths, each datapath i loads x[i] and y[i], adds them, and stores the result in x[i], completing the task in one cycle. If there are m<n datapaths (e.g., m=4, n=15), the system processes the arrays in blocks: elements 0–3, 4–7, 8– 11, and 12–14. In the final block, only three elements are processed, leaving one datapath idle. This illustrates SIMD's efficiency for uniform operations on large datasets but also highlights potential inefficiencies when data sizes do not align with the number of datapaths, as idle datapaths reduce performance.

Q2: Explain the advantages of SIMD systems for data-parallel problems and the challenges they face with conditional operations, using examples.

A2: SIMD systems are highly efficient for data-parallel problems but face challenges with conditional operations due to their synchronous architecture.

- **Advantages for Data-Parallel Problems:**
- **High Throughput:** SIMD systems excel at tasks where the same operation is applied to large datasets. In the document's vector addition example (x[i]+=y[i]), a SIMD system with n datapaths can perform all additions in one cycle, significantly reducing execution time compared to a sequential system, which requires n iterations.
- **Simplified Control:** A single control unit broadcasts instructions, reducing overhead. For the vector addition, one instruction is sent to all datapaths, streamlining execution.
- **Optimized Hardware:** SIMD systems, like vector processors, use vector registers and pipelined functional units to process entire arrays efficiently. The document notes that a loop

like vector addition requires only one load, add, and store per block of vector_length elements, unlike scalar systems requiring operations for each element.

- **Challenges with Conditional Operations:**
- **Idle Datapaths:** SIMD systems require all datapaths to execute the same instruction or remain idle, which is problematic for conditional operations. In the document's example, modifying vector addition to if(y[i]>0.0)x[i]+=y[i] requires each datapath to check if y[i] is positive. Datapaths with non-positive y[i] remain idle while others perform the addition, reducing efficiency.
- **Synchronous Execution:** The lack of instruction storage means datapaths cannot delay or reorder instructions, forcing synchronous operation. This rigidity exacerbates performance degradation when only some datapaths execute the instruction.
- **Performance Impact:** The conditional operations can "seriously degrade" performance, as idle datapaths waste computational resources, especially in tasks with frequent or complex conditionals.

Thus, SIMD systems are powerful for uniform, data-parallel tasks but struggle with conditional operations due to enforced synchronicity and idle datapaths.

Q3: Discuss the role of vector processors in SIMD systems, including their key features and limitations.

A3: Vector processors are a key implementation of SIMD systems, designed to operate on arrays or vectors of data, unlike conventional CPUs that process scalars. The features and limitations:

- **Key Features:**
  - **Vector Registers:** These store vectors of operands (4 to 256 64-bit elements) and perform operations on all elements simultaneously. For example, in the loop x[i]+=y[i], a vector register loads a block of x and y, adds them, and stores the result in one operation.
  - **Vectorized Functional Units:** These units apply the same operation to all elements in a vector (e.g., adding corresponding elements of two vectors), adhering to SIMD principles.
  - **Vector Instructions:** These operate on entire vectors, reducing the number of instructions. For the loop above, only one load, add, and store is needed per block of vector_length elements, unlike scalar systems requiring one per element.
  - **Interleaved Memory:** Memory banks allow near-simultaneous access, minimizing delays when loading/storing vector elements across banks.
  - **Strided and Scatter/Gather Access:** Hardware supports accessing vector elements at fixed (strided) or irregular (scatter/gather) intervals, accelerating operations like accessing every fourth element.

- **Vectorizing Compilers:** These identify vectorizable code and provide feedback on nonvectorizable loops, aiding programmers in optimizing code.
- **Limitations:**
- **Poor Handling of Irregular Data:** Vector processors struggle with irregular data structures, as their design assumes uniform operations on arrays, limiting their versatility.
- **Scalability Constraints:** Scaling vector processors by increasing vector length is challenging, as it requires custom hardware. The document notes that current systems scale
  by adding more vector processors, not longer vectors, and long-vector processors are expensive.
- **Limited Commodity Support:** While commodity systems support short vectors, longvector processors are custom-made, increasing costs and reducing accessibility.

Vector processors are fast and user-friendly for data-parallel tasks due to their optimized hardware and compilers but are less effective for irregular or non-parallel tasks and face scalability and cost challenges.

Q4: How do GPUs leverage SIMD parallelism, and why are they not considered pure SIMD systems? What are their strengths and weaknesses for general computing?

A4: Graphics Processing Units (GPUs) leverage SIMD parallelism for graphics and general computing but are not pure SIMD systems due to their hybrid architecture. The operation, strengths, and weaknesses.

- **Use of SIMD Parallelism:**
  - GPUs apply SIMD parallelism in their programmable pipeline_forest: graphics pipeline stages, executing short shader functions (often a few lines of C code) on multiple data elements (e.g., vertices) simultaneously. Each core has many datapaths (e.g., 128), enabling parallel execution of the same instruction across large datasets, such as pixel or vertex processing.
  - Hardware multithreading supports high data throughput by managing many threads, with some systems storing over a hundred suspended threads per active thread, minimizing memory access stalls.
- **Not Pure SIMD Systems:**
  - GPUs are not purely SIMD because, while datapaths within a core use SIMD parallelism, each core can run multiple instruction streams. Additionally, GPUs typically have dozens of cores, each capable of independent instruction streams, blending SIMD and MIMD characteristics.
  - The document notes that GPUs may use both shared and distributed memory, with cores accessing different memory blocks over a network, further diverging from a pure SIMD model.
- **Strengths for General Computing:**
  - **High Performance:** GPUs handle massive data-parallel tasks efficiently, making them popular for high-performance computing, such as scientific simulations and machine learning.
  - **High Memory Bandwidth:** GPUs maintain high data movement rates, ideal for large datasets like images (hundreds of megabytes).

- **Programming Support:** Languages developed for GPUs (e.g., CUDA for Nvidia) enable general-purpose computing, expanding their utility beyond graphics.
- **Weaknesses for General Computing:**
- **Poor Performance on Small Problems:** GPUs require many threads and large datasets to keep datapaths busy, leading to suboptimal performance on smaller tasks.
- **Complex Programming:** Despite language support, programming GPUs requires managing parallel threads and memory hierarchies, which is more complex than sequential programming.
- **Hybrid Architecture Challenges:** The mix of SIMD and MIMD features, along with shared and distributed memory, complicates optimization compared to pure SIMD systems. In summary, GPUs leverage SIMD parallelism for high-throughput data-parallel tasks but combine MIMD features, making them versatile yet complex for general computing. Their performance excels with large datasets but falters with smaller, less parallel tasks.

2. MIMD systems,

MIMD (Multiple Instruction, Multiple Data) systems are parallel computing architectures where multiple processors execute different instructions on different data elements simultaneously. Key features:

Advantages:

1. Flexibility: Handle diverse workloads and applications.
2. Scalability: Can be scaled up to thousands of processors.

Characteristics:

1. Multiple instruction streams: Each processor executes its own instruction stream.
2. Distributed or shared memory: Processors can have their own memory or share memory.

Types:

1. Tightly coupled: Processors share memory and communicate through shared memory.
2. Loosely coupled: Processors have their own memory and communicate through message passing.

Applications:

1. Scientific simulations: Weather forecasting, fluid dynamics.
2. Data analytics: Large-scale data processing.
3. Machine learning: Distributed training of models.

Examples:

1. Multiprocessor systems: Many modern servers and supercomputers.
2. Cluster computing: Distributed systems composed of multiple nodes.

MIMD systems are versatile and can handle a wide range of applications, making them a fundamental component of modern parallel computing.

Q1: Describe the architecture and operation of MIMD systems, highlighting how they differ from SIMD systems in terms of instruction and data stream management.

A1: MIMD (Multiple Instruction, Multiple Data) systems are designed to support multiple simultaneous instruction streams operating on multiple data streams, making them highly flexible for parallel computing. Architecturally, MIMD systems consist of a collection of fully independent processing units or cores, each with its own control unit and datapath. Unlike SIMD systems, which use a single control unit to broadcast one instruction to all datapaths synchronously, MIMD systems operate asynchronously, with no global clock. This means each processor can execute different instructions at different times, and there may be no temporal relation between processors unless the programmer imposes synchronization.

For example, in a MIMD system, one core might execute a matrix multiplication while another performs a sorting operation, each on different data sets. This contrasts with a SIMD system, where all datapaths must execute the same instruction (e.g., vector addition (x[i] += y[i])) or remain idle. The document notes that MIMD systems are divided into shared-memory systems, where cores access a common memory via an interconnect, and distributed-memory systems, where each core has private memory and communicates over a network. This flexibility allows MIMD systems to handle diverse parallel tasks, but it requires careful coordination to avoid conflicts or race conditions, unlike the simpler, synchronous control of SIMD systems.

Q2: Compare and contrast shared-memory and distributed-memory MIMD systems in terms of their architecture, communication methods, and scalability.

A2: Shared-memory and distributed-memory MIMD systems differ significantly in architecture, communication, and scalability, as outlined in the document.

- **Architecture:**
  - **Shared-Memory Systems:** These consist of multiple cores connected to a common memory via an interconnection network (e.g., bus or crossbar). The document highlights multicore processors as typical examples, with private L1 caches and possibly shared lower-level caches. They can be Uniform Memory Access (UMA), where all memory locations have equal access times, or Non-Uniform Memory Access (NUMA), where access times vary based on memory proximity.
  - **Distributed-Memory Systems:** Each core is paired with its own private memory, and processor-memory pairs communicate over a network (e.g., Ethernet in clusters). The document describes clusters as common distributed-memory systems, often hybrid, with shared-memory nodes (multicore PCs) connected via commodity networks.
- **Communication Methods:**
  - **Shared-Memory Systems:** Cores communicate implicitly by reading and writing to shared memory locations. For example, a core might update a shared variable (x), which another core reads, simplifying programming but requiring mechanisms like cache coherence to ensure consistency.

  - **Distributed-Memory Systems:** Cores communicate explicitly by sending messages or using functions to access remote memory. For instance, a core might send a data block to another core over a toroidal mesh, requiring programmers to manage communication explicitly, which increases complexity.
- **Scalability:**
  - **Shared-Memory Systems:** These scale poorly for large systems due to interconnect limitations. The document notes that buses cause contention with many processors, and large crossbars are expensive (requiring (p^2) switches for (p) processors). This limits sharedmemory systems to smaller configurations, such as multicore chips.
  - **Distributed-Memory Systems:** These scale better for large-scale problems due to costeffective interconnects like hypercubes or toroidal meshes. For example, a toroidal mesh with (p) processors requires only (2p) links, and clusters can scale to thousands of nodes, making them ideal for vast computational tasks like scientific simulations.

In summary, shared-memory systems offer simpler programming but limited scalability, while distributed-memory systems provide scalability at the cost of complex communication management.

Q3: Explain the role of interconnection networks in MIMD systems, focusing on the differences between shared-memory and distributed-memory interconnects, and provide examples.

A3: Interconnection networks are critical to the performance of MIMD systems, as they determine how efficiently cores communicate and access memory. The distinguishes between interconnects for shared-memory and distributed-memory MIMD systems, highlighting their designs and implications.

- **Shared-Memory Interconnects:**
  - **Description:** In shared-memory MIMD systems, cores connect to a common memory via an interconnect, such as a bus or crossbar. The document explains that buses were historically common due to low cost and flexibility but are unsuitable for large systems because shared communication wires lead to contention as processor count increases.
  - **Example:** A crossbar, illustrated in Fig. 2.7, uses switches to route data between processors and memory modules. For four processors ((P_1) to (P_4)) and four memory modules ((M_1) to (M_4)), a crossbar allows simultaneous accesses (e.g., (P_1) writes to (M_4), (P_2) reads from (M_3)) unless two cores access the same module. Crossbars are faster than buses but expensive, requiring (p^2) switches for (p) processors.
  - **]Impact:** Crossbars reduce contention compared to buses, improving performance, but their high cost limits scalability for large systems.

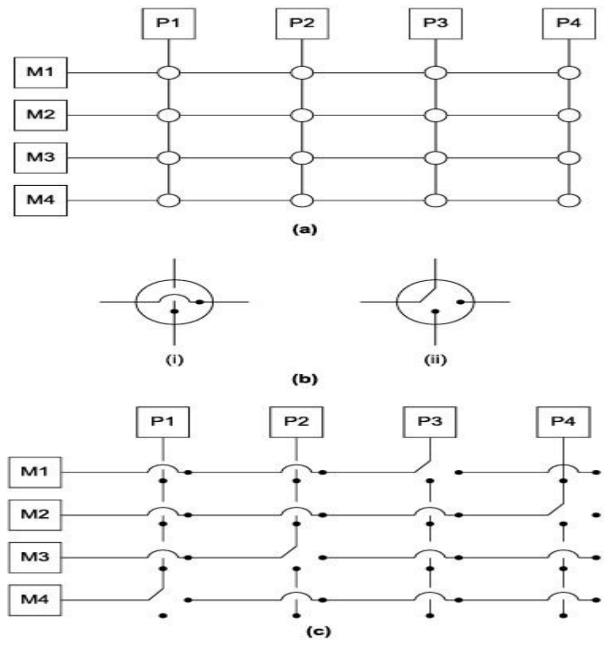


FIGURE 2.7
(a) A crossbar switch connecting four processors (P_i) and four memory modules (M_j); (b) configuration of internal switches in a crossbar; (c) simultaneous memory accesses by the processors.

[Refer: "Peter S Pacheco, Matthew Malensek – An Introduction to Parallel Programming, second edition, Morgan Kauffman]

- **Distributed-Memory Interconnects:**
  - **Description:** In distributed-memory MIMD systems, each processor-memory pair connects to a network via switches, using direct (e.g., ring, toroidal mesh, hypercube) or indirect (e.g., crossbar, omega network) interconnects. The document emphasizes that direct interconnects link switches to processors and other switches, while indirect interconnects may not connect switches directly to processors.
  - **Examples:** A toroidal mesh (Fig. 2.8b) requires (2p) links for (p) processors and supports more simultaneous communications than a ring (Fig. 2.8a), which has only (p) links. A hypercube (Fig. 2.12) offers a bisection width of (p/2), providing high connectivity but requiring complex switches. An omega network (Fig. 2.15) uses fewer switches ((2p \log_2(p))) than a crossbar ((p^2)), but some communications cannot occur simultaneously
  - **Impact:** Distributed-memory interconnects are more scalable and cost-effective, enabling systems with thousands of processors, but they require explicit communication, increasing programming complexity.

The interconnect performance, measured by metrics like bisection width (e.g., (2\sqrt{p}) for a toroidal mesh, (p/2) for a hypercube), significantly affects system efficiency. Shared-memory interconnects prioritize low-latency memory access but struggle with scalability, while distributedmemory interconnects support large-scale systems at the cost of communication overhead.

.

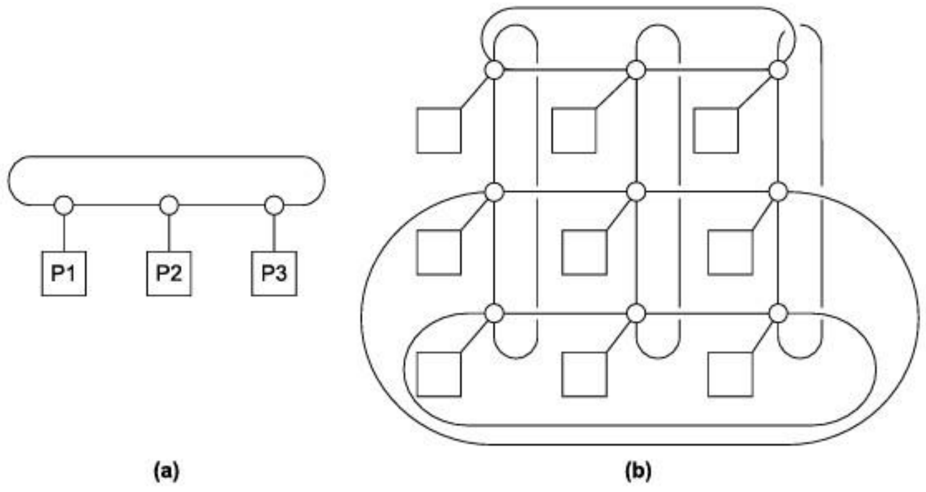


FIGURE 2.8

(a) A ring and (b) a toroidal mesh.

[Refer: "Peter S Pacheco, Matthew Malensek – An Introduction to Parallel Programming, second edition, Morgan Kauffman]

Q4: Discuss the cache coherence problem in shared-memory MIMD systems, including the approaches to address it and the issue of false sharing, with examples.

A4: The cache coherence problem in shared-memory MIMD systems arises when multiple cores cache the same variable, and an update by one core may not be reflected in others' caches, leading to inconsistent data. The illustrates this with an example and discusses solutions and the related issue of false sharing.

- **Cache Coherence Problem:**
- **Description:** In a shared-memory system with two cores, each with a private cache, consider a shared variable (x) initialized to 2. At time 0, core 0 executes (y0 = x) (setting (y0 = 2)), and core 1 executes (y1 = 3 * x) (setting (y1 = 6)). At time 1, core 0 updates (x = 7). At time 2, core 1 executes (z1 = 4 * x). Without cache coherence, core 1's cache may retain (x = 2), resulting in (z1 = 8), instead of the expected (z1 = 28) (from (4 * 7)).
- **Cause:** Caches are managed by hardware, not programmers, and updates to a cached variable in one core's cache (with write-through or write-back policies) may not propagate to other cores' caches, causing unpredictable behavior.
- **Approaches to Address Cache Coherence:**
- **Snooping Cache Coherence:** Used in bus-based systems, where cores "snoop" the bus for updates. When core 0 updates (x), it broadcasts the update, and core 1 marks its cached (x) as invalid. The document notes this works with both write-through and write-back caches but

requires a broadcast-supporting interconnect. However, snooping is not scalable for large systems due to frequent broadcasts.

- **Directory-Based Cache Coherence:** Suitable for large systems, this uses a distributed directory to track cache line statuses. When core 0 caches (x), the directory records it. Upon updating (x), only cores with (x) in their caches are notified to invalidate it, reducing communication overhead. The document highlights the trade-off of additional storage for scalability.
- **False Sharing:**
- **Description:** False sharing occurs when cores access different variables in the same cache line, causing unnecessary invalidations. The document's example involves a vector (y) of 8 doubles (64 bytes, fitting one cache line) in a two-core system. Core 0 updates (y[0]) to (y[3]), and core 1 updates (y[4]) to (y[7]). Since (y) is in one cache line, each update invalidates the other core's cache, forcing frequent memory accesses despite no actual data sharing.
- **Impact and Solution:** False sharing degrades performance by increasing main memory accesses but does not affect correctness. The document suggests using thread-local temporary storage to accumulate results, then copying to shared memory to reduce cache line conflicts.

In conclusion, cache coherence ensures data consistency in shared-memory MIMD systems, with snooping suitable for small systems and directory-based methods for larger ones. False sharing, while avoidable through careful programming, highlights the importance of aligning data with cache line boundaries to optimize performance.

3 Interconnection networks,

An interconnection network in parallel systems enables communication between multiple processors, memory modules, and other components. Key aspects:

Characteristics:

1. Topology: Physical or logical arrangement of nodes and links.
2. Bandwidth: Data transfer rate between nodes.
3. Latency: Time taken for data to travel between nodes.

Types:

1. Static networks: Fixed connections between nodes.
2. Dynamic networks: Connections can be reconfigured.

Topologies:

1. Bus: Single shared communication channel.
2. Mesh: Grid-like structure with node-to-node connections.
3. Hypercube: Highly connected, scalable network.
4. Torus: Mesh with wrap-around connections.

Performance factors:

1. Scalability: Ability to handle increased nodes and traffic.
2. Congestion: Network overload due to excessive traffic.

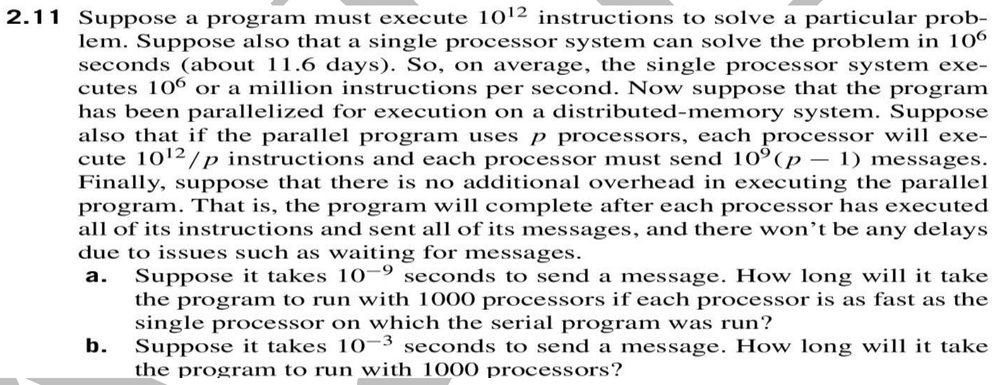
Importance:

1. Efficient data transfer: Enables parallel processing and scalability.
2. System performance: Directly impacts overall system efficiency.

A well-designed interconnection network is crucial for achieving high performance in parallel systems.

Q1: Explain the importance of interconnection networks in parallel computing systems and describe how their performance impacts parallel program efficiency.

A1: Interconnection networks are critical components in parallel computing systems, as they enable communication between processors and memory in both shared-memory and distributedmemory architectures. The document emphasizes that even with high-performance processors and memory, a slow interconnect can significantly degrade the performance of parallel programs, except for the simplest ones (see Exercise 2.11). The interconnect determines how quickly data can be transferred, affecting the overall execution time of parallel tasks.



2.11 Suppose a program must execute 10^{12} instructions to solve a particular problem. Suppose also that a single processor system can solve the problem in 10^6 seconds (about 11.6 days). So, on average, the single processor system executes 10^6 or a million instructions per second. Now suppose that the program has been parallelized for execution on a distributed-memory system. Suppose also that if the parallel program uses p processors, each processor will execute $10^{12}/p$ instructions and each processor must send $10^9(p-1)$ messages. Finally, suppose that there is no additional overhead in executing the parallel program. That is, the program will complete after each processor has executed all of its instructions and sent all of its messages, and there won't be any delays due to issues such as waiting for messages.

a. Suppose it takes 10^{-9} seconds to send a message. How long will it take the program to run with 1000 processors if each processor is as fast as the single processor on which the serial program was run?

b. Suppose it takes 10^{-3} seconds to send a message. How long will it take the program to run with 1000 processors?

[Refer: "Peter S Pacheco, Matthew Malensek – An Introduction to Parallel Programming, second edition, Morgan Kauffman]

The performance of an interconnect is characterized by two key metrics: latency and bandwidth. Latency ((l)) is the time from when a source begins transmitting data to when the destination starts receiving it, while bandwidth ((b)) is the rate at which data is received after the first byte. The total message transmission time for (n) bytes is given by (l + n/b). For example, in a distributed-memory system, a high-latency interconnect increases the time for message passing, slowing down tasks requiring frequent communication. Similarly, in a shared-memory system, a congested interconnect like a bus can delay memory accesses, reducing throughput. Metrics like bisection width and bisection bandwidth further quantify connectivity, indicating how many simultaneous communications can occur across network halves. Poor interconnect performance leads to bottlenecks, making efficient network design essential for scalable parallel systems.

Q2: Compare and contrast shared-memory and distributed-memory interconnects, focusing on their designs, advantages, and limitations.

A2: Shared-memory and distributed-memory interconnects serve distinct roles in parallel computing, with different designs, advantages, and limitations as outlined in the document.

- **Shared-Memory Interconnects:**
  - **Design:** These connect processors to a common memory. Historically, buses were used, consisting of shared communication wires controlled by hardware. Modern systems use switched interconnects like crossbars (Fig. 2.7 Refer Above), where switches route data between processors and memory modules, allowing simultaneous accesses unless two cores target the same module. o **Advantages:** Buses are low-cost and flexible, easily connecting multiple devices. Crossbars enable faster communication by supporting simultaneous accesses, reducing contention compared to buses. For example, a crossbar allows (P_1) to write to (M_4) while (P_2) reads from (M_3).
  - **Limitations:** Buses suffer from contention as processor count increases, causing delays in memory access. Crossbars are faster but expensive, requiring (p^2) switches for (p) processors, limiting scalability for large systems.
- **Distributed-Memory Interconnects:**
  - **Design:** These connect processor-memory pairs, divided into direct (e.g., ring, toroidal mesh, hypercube) and indirect (e.g., crossbar, omega network) types. Direct interconnects link switches to processors and other switches (Fig. 2.8), while indirect interconnects use switching networks with unidirectional links (Fig. 2.13). For instance, a toroidal mesh uses (2p) links for (p) processors, and a hypercube has (p=2^d) nodes with (d+1) links per switch.
  - **Advantages:** Distributed-memory interconnects are more scalable and cost-effective. A toroidal mesh supports more simultaneous communications than a ring's (p) links, and a hypercube's bisection width of (p/2) offers high connectivity. Indirect interconnects like omega networks use fewer switches ((2p \log_2(p))) than crossbars ((p^2)), reducing costs.
  - **Limitations:** Direct interconnects like rings have lower connectivity (bisection width of 2 for 8 nodes), causing delays in some communication patterns. Indirect interconnects like omega networks cannot support all simultaneous communications (e.g., processor 0 to 6 blocks 1 to 7), reducing flexibility compared to crossbars.

In summary, shared-memory interconnects prioritize low-latency memory access but are less scalable, while distributed-memory interconnects offer scalability and cost-effectiveness for large systems, albeit with complex communication management.

Q3: Discuss the concept of bisection width as a measure of interconnect connectivity, and compare the bisection widths of a ring, toroidal mesh, hypercube, and crossbar.

A3: Bisection width is a key metric for evaluating the connectivity of an interconnection network, defined as the minimum number of links that must be removed to divide the network into two equal halves, representing the worst-case number of simultaneous communications possible between halves. The bisection widths for several interconnects, allowing comparison of their connectivity.

- **Ring:**
  - **Bisection Width:** For a ring with (p) nodes (e.g., 8 nodes, Fig. 2.9a), the bisection width is 2, as only two links connect the two halves of four nodes each in the worst-case scenario. Even if another bisection allows four communications (Fig. 2.9b), the worst-case estimate prevails.
  - **Implication:** Low bisection width indicates limited connectivity, making rings prone to communication bottlenecks in large systems.
- **Toroidal Mesh:**
  - **Bisection Width:** For a square two-dimensional toroidal mesh with (p = q^2) nodes (where (q) is even), the bisection width is (2q = 2\sqrt{p}) (Fig. 2.10). For example, with (p = 16) ((q = 4)), the bisection width is 8.
  - **Implication:** Higher than a ring, the toroidal mesh supports more simultaneous communications, improving scalability for distributed-memory systems.
- **Hypercube:**
  - **Bisection Width:** For a hypercube with (p = 2^d) nodes, the bisection width is (p/2).

    For (p = 8) ((d = 3)), the bisection width is 4.
  - **Implication:** The hypercube offers high connectivity, supporting more communications than a toroidal mesh or ring, but requires complex switches ((1+\log_2(p)) links), increasing costs.

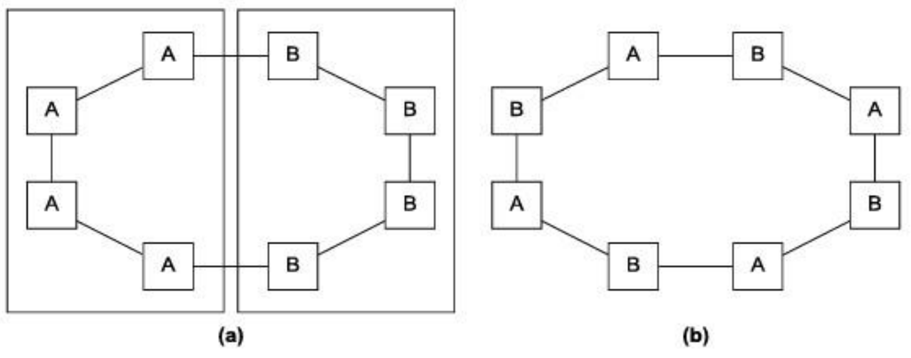


FIGURE 2.9

Two bisections of a ring: (a) only two communications can take place between the halves and (b) four simultaneous connections can take place.

[Refer: "Peter S Pacheco, Matthew Malensek – An Introduction to Parallel Programming, second edition, Morgan Kauffman]

- **Crossbar:**
  - **Bisection Width:** For a (p \times p) crossbar, the bisection width is (p). For (p = 8), the bisection width is 8, matching the toroidal mesh for (p = 16) but achieved with fewer nodes.
  - **Implication:** Crossbars provide excellent connectivity, allowing all processors to communicate simultaneously unless targeting the same module, but their (p^2) switch requirement makes them expensive.

Comparison: The crossbar has the highest bisection width ((p)), followed by the hypercube ((p/2)), toroidal mesh ((2\sqrt{p})), and ring (2). Higher bisection width indicates better support for simultaneous communications, but cost and complexity increase. Rings are cheapest but least connected, while crossbars are highly connected but impractical for large systems. Hypercubes and toroidal meshes balance connectivity and cost, making them suitable for large distributedmemory systems.

Q4: Describe the latency and bandwidth metrics for interconnection networks, and explain how they influence the performance of message transmission in distributed-memory systems, including potential variations in the definition of latency.

A4: Latency and bandwidth are fundamental metrics for assessing the performance of interconnection networks, particularly in distributed-memory systems where processors communicate explicitly via messages. The document provides detailed definitions and highlights their impact on message transmission.

- **Latency:**
  - **Definition:** Latency ((l)) is the time elapsed from when the source begins transmitting data to when the destination starts receiving the first byte. However, the document notes that latency is sometimes used to mean total message transmission time or the fixed overhead for assembling and disassembling messages (e.g., including destination addresses, size, and error correction data).
  - **Impact:** High latency increases the initial delay before data transfer begins, critical in applications with frequent small messages. For example, in a distributed-memory system, a latency of 10 microseconds delays each message, accumulating significantly in tasks requiring thousands of communications.
- **Bandwidth:**
  - **Definition:** Bandwidth ((b)) is the rate at which the destination receives data after receiving the first byte, typically measured in megabytes per second. It determines how quickly large data volumes are transferred once transmission starts.
  - **Impact:** Low bandwidth slows the transfer of large messages, creating bottlenecks in data-intensive tasks. For instance, a bandwidth of 1 GB/s transfers a 1 MB message in 1 millisecond, but a 100 MB message takes 100 milliseconds, dominating performance in big data applications.

- **Message Transmission Time:**
  - **Formula:** The time to transmit a message of (n) bytes is (l + n/b). For example, with (l = 10 \mu s), (b = 1 GB/s = 10^9 B/s), and (n = 10^6 B), the time is (10 \times 10^{6} + 10^6 / 10^9 = 10 \times 10^{-6} + 10^{-3} = 0.00001 + 0.001 = 0.00101) seconds (1.01 ms).
  - **Influence:** In distributed-memory systems, message passing is common, and the transmission time directly affects performance. Small messages are latency-dominated (since (n/b) is small), while large messages are bandwidth-dominated (since (n/b) is large). For instance, in a cluster using a toroidal mesh, frequent small messages (e.g., synchronization signals) suffer from high latency, while large data transfers (e.g., array blocks) are limited by bandwidth.
- **Variations in Latency Definition:**
  - The document warns that latency may include overheads like message assembly/disassembly, which adds time for packaging data with metadata (e.g., destination address, error correction). For example, assembling a message might take 5 microseconds, increasing effective latency. This variability complicates performance predictions, as latency might be reported as total transmission time in some contexts, misleading comparisons.

In conclusion, latency and bandwidth govern message transmission efficiency in distributedmemory systems, with latency critical for small, frequent messages and bandwidth for large data transfers. Understanding latency's variable definitions ensures accurate performance analysis and optimization in parallel programming.

4 Cache coherence,

The cache coherence problem in shared-memory systems occurs when multiple processors or cores have their own caches, leading to inconsistencies in data.

Key Issues:

1. Data inconsistency: Multiple caches have different values for the same memory location.
2. Stale data: A processor uses outdated data from its cache.

Causes:

1. Multiple caches: Each processor/core has its own cache.
2. Shared memory: Multiple processors access shared data.

Solutions:

1. Cache coherence protocols: MESI (Modified, Exclusive, Shared, Invalid) and others.
2. Cache invalidation: Invalidating cache lines when data is modified.
3. Cache update: Updating all caches when data is modified.

Importance:

1. Correctness: Ensures data consistency and accuracy.
2. Performance: Affects system performance and scalability.

Maintaining cache coherence is crucial in shared-memory systems to ensure data consistency and correctness.

Q1: Explain the cache coherence problem in shared-memory systems with an example, and discuss why it poses a challenge for programmers.

A1: The cache coherence problem in shared-memory systems occurs when multiple cores maintain private caches that store the same variable, and an update by one core is not immediately reflected in the caches of other cores, leading to data inconsistency. The illustrates this with an example involving a shared-memory system with two cores, each with a private cache (Fig. 2.17). Suppose a shared variable x is initialized to 2, and private variables y0 (core 0), y1, and z1 (core 1) are used. At time 0, core 0 executes y0=x (setting y0=2), and core 1 executes y1=3*x (setting y1=6). At time 1, core 0 updates x=7. At time 2, core 1 executes z1=4*x. Without cache coherence, core 1's cache may retain x=2, so z1=4*2=8, instead of the expected z1=4*7=28.

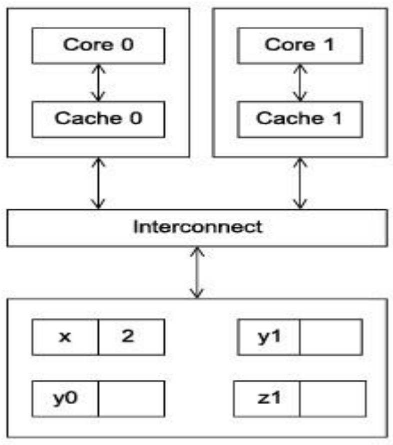


FIGURE 2.17
A shared-memory system with two cores and two caches.

[Refer: "Peter S Pacheco, Matthew Malensek – An Introduction to Parallel Programming, second edition, Morgan Kauffman]

This poses a challenge for programmers because caches are managed by hardware, not software, so programmers lack direct control over cache updates. The unpredictability persists regardless of cache policies (write-through or write-back). In a write-through policy, main memory is updated with x=7, but core 1's cache retains x=2. In a write-back policy, core 0's cache update may not be visible to core 1. This inconsistency makes it difficult for programmers to ensure correct program behavior, as the outcome of simple operations like reading x becomes unpredictable, necessitating hardware mechanisms to maintain coherence.

Q2: Compare and contrast snooping cache coherence and directory-based cache coherence, highlighting their mechanisms, advantages, and limitations.

A2: Snooping and directory-based cache coherence are two primary approaches to ensuring cache consistency in shared-memory systems, each with distinct mechanisms, advantages, and limitations.

 Snooping Cache Coherence:

- **Mechanism:** In snooping, cores monitor a shared interconnect (typically a bus) for cache updates. When a core (e.g., core 0) updates a variable like x in its cache, it broadcasts this update across the interconnect. Other cores (e.g., core 1) "snoop" the interconnect, detect the update to the cache line containing x , and mark their own copies as invalid. The document notes that broadcasts inform about cache line updates, not specific variables.
- **Advantages:** Snooping is effective for small systems with a shared interconnect like a bus. With write-through caches, no additional traffic is needed, as cores can watch for writes. It supports both write-through and write-back caches, though write-back requires extra communication. The simplicity of snooping makes it suitable for systems with few cores. ○ **Limitations:** Snooping is not scalable for large systems because broadcasts are required for every cache update, leading to performance degradation. The frequent broadcasts overwhelm the interconnect in larger networks, slowing down the system relative to local memory access speeds.

 Directory-Based Cache Coherence:

- **Mechanism:** This approach uses a distributed directory data structure to track the status of each cache line across cores. For example, when core 0 caches a line, the directory records this. Upon updating a variable in that line, the directory is consulted, and only cores with copies of the line are notified to invalidate them. The document describes a scenario where a system with distributed-memory architecture but a single address space (e.g., core 0 accessing core 1's memory) benefits from this method.
- **Advantages:** Directory-based coherence is scalable, as it avoids broadcasting by targeting only affected cores, reducing communication overhead. It supports large systems with many cores, making it suitable for complex architectures where accessing remote memory is slower but feasible.
- **Limitations:** The directory requires substantial additional storage to maintain cache line statuses, increasing hardware complexity and cost. The document notes this trade-off, emphasizing that the storage overhead is justified by improved scalability compared to snooping.

Comparison: Snooping is simpler and effective for small, bus-based systems but fails in large networks due to broadcast overhead. Directory-based coherence scales better by minimizing communication but demands more storage. Both ensure coherence, but directory-based is preferred for large-scale shared-memory systems.

Q3: Describe the concept of false sharing in shared-memory systems, including an example, and explain how it impacts performance and how it can be mitigated.

A3: False sharing is a performance issue in shared-memory systems where multiple cores access different variables stored in the same cache line, causing unnecessary cache invalidations and memory accesses, even though the variables are not actually shared. The example to illustrate this problem. Consider a program that computes values using a function f(i,j) and accumulates them into a vector

```
y[n]: for (i = 0; i < n; i++) {
          for (j = 0; j < m; j++) {
                    y[i] += f(i, j);
                 }
          }
```

This is parallelized on two cores, with core 0 handling iterations i=0 to n/2−1, and core 1 handling i=n/2 to n−1. Suppose m=8, doubles are 8 bytes, cache lines are 64 bytes (holding 8 doubles), and y[0] starts a cache line. Thus, y[0] to y[7] occupy one cache line. When core 0 updates y[0] to y[3], and core 1 updates y[4] to y[7], each update invalidates the entire cache line in the other core's cache, forcing repeated memory fetches, despite no actual data sharing.

- **Performance Impact:** False sharing degrades performance by increasing main memory accesses, as each core's update triggers cache invalidations in the other. The document notes that for large n, most y[i]+=f(i,j) assignments access main memory, significantly slowing execution. Unlike coherence issues, false sharing does not affect program correctness but ruins efficiency.
- **Mitigation:** The thread-local temporary storage to accumulate results, then copying them to shared memory. For example, each core could use a local array to store partial sums for its assigned iterations, updating y only at the end. This reduces cache line conflicts by minimizing simultaneous accesses to the same cache line, improving performance.

False sharing highlights the importance of aligning data structures with cache line boundaries and careful parallelization to avoid unintended performance penalties in shared-memory systems.

Q4: Discuss how cache coherence mechanisms interact with write-through and write-back cache policies, and explain the implications for shared-memory system.

A4: Cache coherence mechanisms, such as snooping and directory-based protocols, must account for the cache policies (write-through or write-back) used in shared-memory systems to ensure data consistency. The interactions and their implications for system design, using the cache coherence problem example.

- **Write-Through Cache Policy:**
  - **Interaction with Coherence Mechanisms:** In a write-through policy, when a core updates a cached variable (e.g., core 0 sets x=7), the update is immediately written to main memory.

    However, this does not update other cores' caches (e.g., core 1's cache retains x=2). Snooping coherence allows other cores to monitor the interconnect for such updates and invalidate their cache lines. The shared bus and write-through caches, no additional traffic is needed, as cores can "watch" for writes. Directory-based coherence records the update in the directory, notifying other cores to invalidate their copies.

- **Implications:** Write-through simplifies snooping in small systems, as main memory is always updated, reducing the need for extra communication. However, it generates more memory traffic, as every write goes to main memory, which can bottleneck large systems. Directory-based coherence mitigates this by targeting specific cores, but the policy still increases memory bandwidth demands, impacting scalability.

- **Write-Back Cache Policy:**
  - **Interaction with Coherence Mechanisms:** In a write-back policy, updates (e.g., x=7 in core 0's cache) are stored only in the cache until the cache line is evicted, delaying main memory updates. This exacerbates the coherence problem, as other cores (e.g., core 1) may not see the update when accessing x. Snooping requires an extra communication to broadcast the update, as the document states, since cache updates are not sent to memory immediately. Directory-based coherence tracks these updates in the directory, notifying affected cores to invalidate their cache lines when the update occurs.
  - **Implications:** Write-back reduces memory traffic by deferring writes, improving performance in systems with high write frequency. However, it complicates coherence, requiring additional communication for snooping or directory updates. This makes writeback more suitable for systems with directory-based coherence, where selective notifications reduce overhead, but it increases complexity in snooping systems due to broadcast demands.
  - **Design Implications:** The choice of cache policy influences shared-memory system design. Write-through is simpler for small, bus-based systems with snooping, as it aligns with frequent main memory updates, but it limits scalability due to memory traffic. Write-back is preferred for larger systems with directory-based coherence, as it minimizes memory accesses, but requires robust interconnects to handle coherence traffic. The example of a twocore system underscores that both policies cause coherence issues without proper mechanisms, necessitating careful design of interconnects and coherence protocols to balance performance, scalability, and complexity.

In summary, coherence mechanisms must be tailored to cache policies, with write-through favoring simpler snooping in small systems and write-back supporting scalable directory-based designs, shaping the architecture of efficient shared-memory systems.

5 Shared-memory vs. distributed-memory,

Two fundamental architectures for parallel computing:

Shared-Memory:

1. Single memory space: All processors share a common memory.
2. Easy data sharing: Processors can access shared data directly.

3. Faster communication: Less overhead for data exchange.

Distributed-Memory:

1. Multiple memory spaces: Each processor has its own private memory.
2. Message passing: Processors communicate by exchanging messages.
3. Scalability: Can handle large number of processors.

Key differences:

1. Memory access: Shared-memory allows direct access, while distributed-memory requires message passing.
2. Scalability: Distributed-memory is more scalable.

Choosing between:

1. Shared-memory: Suitable for smaller-scale parallel applications.
2. Distributed-memory: Ideal for large-scale parallel applications.

Each architecture has its strengths and weaknesses, and the choice depends on the specific application requirements.

Q1: Explain the architectural differences between shared-memory and distributed-memory systems, and discuss how these differences influence their communication mechanisms. A11: Shared-memory and distributed-memory systems differ fundamentally in their memory organization and communication approaches.

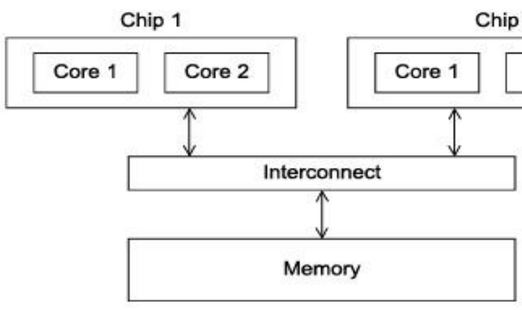


FIGURE 2.5
A UMA multicore system.

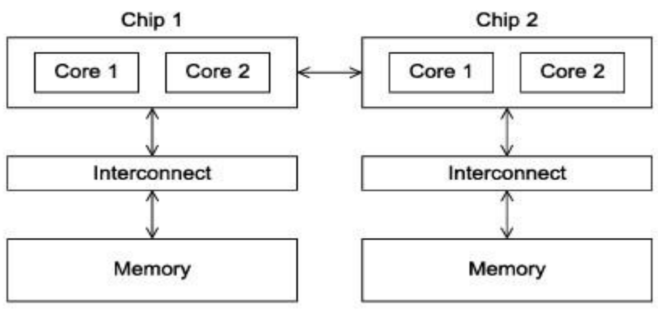


FIGURE 2.6
A NUMA multicore system.

[Refer: "Peter S Pacheco, Matthew Malensek – An Introduction to Parallel Programming, second edition, Morgan Kauffman]

- Shared-Memory Systems:
  - o Architecture: In shared-memory systems, a collection of autonomous processors (or cores) is connected to a common memory system via an interconnection network (Fig. 2.3). Typically, these systems use multicore processors with private level 1 caches and possibly shared higher-level caches . The interconnect can be a bus or a switched interconnect like a crossbar .Two types exist: Uniform Memory Access (UMA), where all cores have equal memory access times (Fig. 2.5), and Nonuniform Memory Access (NUMA), where local memory access is faster than remote (Fig. 2.6).
  - o Communication Mechanism: Processors communicate implicitly by reading from and writing to shared data structures in the common memory. For example, one core can update a shared variable, and another can read it without explicit message passing, simplifying programming.
- Distributed-Memory Systems:
  - o Architecture: Each processor is paired with its own private memory, forming a processormemory pair, and these pairs communicate over an interconnection network (Fig. 2.4). Common implementations include clusters of commodity systems (e.g., PCs) connected via networks like Ethernet, often with shared-memory nodes, termed hybrid systems .Direct interconnects (e.g., ring, toroidal mesh, hypercube) and indirect interconnects (e.g., crossbar, omega network) facilitate communication . o Communication Mechanism: Processors communicate explicitly by sending messages or using special functions to access another processor's memory. For instance, to share data, a processor must package it into a message and transmit it over the network, requiring explicit coordination.
  - Influence on Communication: The shared-memory architecture enables seamless data sharing through memory accesses, reducing programmer effort but introducing challenges like cache coherence .Distributed-memory systems require explicit message passing, which is more complex to program but avoids coherence issues since each processor's memory is private. The document emphasizes that shared-memory's implicit communication is appealing, but distributed-memory's explicit approach scales better for large systems due to cost-effective interconnects .

These architectural differences shape the trade-offs between programming ease and scalability, with shared-memory favoring simplicity and distributed-memory excelling in large-scale applications.

Q2: Compare and contrast the advantages and limitations of shared-memory and distributed-memory systems, focusing on their suitability for different types of parallel computing problems.

A2: Shared-memory and distributed-memory systems each have distinct advantages and limitations, influencing their suitability for various parallel computing problems.

- Shared-Memory Systems:
  - Advantages:
    - Programming Ease: Implicit communication via shared data structures is conceptually simpler, making shared-memory systems appealing to programmers. The document notes that most programmers prefer this model over explicit message passing .
    - UMA and NUMA Variants: UMA systems offer uniform memory access times, simplifying programming, while NUMA systems provide faster local memory access and support larger memory capacities .
    - Efficient for Small Systems: Shared-memory systems, especially with buses or crossbars, are cost-effective and flexible for systems with few processors, supporting simultaneous communications .
  - Limitations:
    - Scalability Issues: The document highlights that scaling shared-memory systems is costly due to interconnect limitations. Buses suffer from contention as processor count increases, and crossbars require expensive p2 switches for p processors .
    - Cache Coherence Overhead: Maintaining cache coherence in multi-core systems introduces complexity and performance overhead, as updates to shared variables must propagate across caches .
    - Suitability: Best for problems with frequent data sharing and moderate processor counts, such as multi-threaded applications on multicore CPUs, but less effective for large-scale, data-intensive tasks due to interconnect costs.
- Distributed-Memory Systems:
  - Advantages:
    - Scalability: Distributed-memory interconnects like hypercubes and toroidal meshes are relatively inexpensive and support thousands of processors, making them ideal for large systems . For example, a toroidal mesh uses 2p links versus a ring's p, enhancing connectivity .
    - No Cache Coherence Issues: Since each processor has private memory, there's no need for cache coherence mechanisms, simplifying hardware design .
    - Suitability for Large Problems: The document states that distributed-memory systems are better suited for problems requiring vast data or computation, such as scientific simulations on clusters .

- Limitations:
  - Programming Complexity: Explicit communication via message passing or special functions is more challenging, requiring programmers to manage data transfers explicitly .
  - Communication Overhead: Message transmission time, given by l+n/b (latency l bandwidth b message size n), can introduce delays, especially for frequent small messages .
  - Suitability: Ideal for large-scale, loosely coupled problems where data can be partitioned across nodes, but less efficient for tasks requiring frequent, fine-grained communication.

•Comparison: Shared-memory systems excel in ease of programming and small-tomedium-scale applications but are limited by interconnect scalability and coherence overhead. Distributed-memory systems offer superior scalability and are suited for large, data-intensive problems but demand more programming effort. The document underscores that distributed-memory systems dominate in scenarios requiring thousands of processors due to cost-effective interconnects like hypercubes .

Q3: Discuss the role of interconnection networks in the scalability of shared-memory and distributed-memory systems, including specific examples.

A3: Interconnection networks are critical to the scalability of both shared-memory and distributedmemory systems, as they determine communication efficiency and system performance, especially as processor counts increase. The decisive role and provides specific examples .

- Shared-Memory Systems:
  - Role in Scalability: The interconnect facilitates communication between processors and shared memory. However, scalability is limited by interconnect cost and contention. The document notes that a slow interconnect can degrade performance significantly, even with high-performance processors and memory.
- Examples:
  - Bus: Historically used in shared-memory systems, a bus consists of shared communication wires. Its low cost and flexibility allow easy device connection, but contention increases with more processors, causing delays. For example, connecting many processors to a bus leads to frequent waits for memory access, making it unsuitable for large systems.
  - Crossbar: A switched interconnect like a crossbar (Fig. 2.7) uses switches to route data, enabling simultaneous communications (e.g., P1 writes to M4, P2 reads from M3) unless cores access the same memory module. Crossbars are faster than buses but require p2 switches for p processors, making them expensive and limiting scalability for large systems.

- Implications: The document highlights that buses are suitable only for small systems, and large crossbars are rare due to cost, restricting shared-memory scalability.

- Distributed-Memory Systems:
  - Role in Scalability: Interconnects enable communication between processor-memory pairs, and their design significantly enhances scalability. Distributed-memory interconnects like hypercubes and toroidal meshes are cost-effective and support thousands of processors, making them ideal for large-scale systems.
  - Examples:
    - Toroidal Mesh: A direct interconnect with 2p links for p processors, compared to a ring's p, supports more simultaneous communications. Its bisection width is 2p for a square mesh, indicating better connectivity than a ring's bisection width of 2. This makes it scalable for clusters.
    - Hypercube: With p=2d nodes, a hypercube has a bisection width of p/2, offering high connectivity (page 41). Each switch supports 1+log2(p) links, more complex than a toroidal mesh's five, but its scalability has been proven in actual systems.
    - Clusters: These use commodity networks like Ethernet to connect shared-memory nodes, forming hybrid systems. The document notes their scalability for large, distributed-memory systems.
  - Implications: Distributed-memory interconnects are less costly and more scalable, enabling systems with thousands of processors for vast computational tasks.

•Comparison: Shared-memory interconnects (buses, crossbars) suffer from contention or high costs, limiting scalability to small systems. Distributed-memory interconnects (toroidal meshes, hypercubes) offer cost-effective, high-connectivity solutions, supporting large-scale applications. The distributed-memory systems are better suited for massive data or computation due to these scalable interconnects.

Q4: Analyze why distributed-memory systems are preferred for large-scale parallel computing applications over shared-memory systems, on hardware and interconnects.

A4: Distributed-memory systems are preferred for large-scale parallel computing applications due to their superior scalability, cost-effective interconnects, and ability to handle vast data and computation.

- Scalability Advantages:
  - Interconnect Design: The distributed-memory systems use interconnects like hypercubes and toroidal meshes, which are relatively inexpensive and scale to thousands of processors . For example, a toroidal mesh with p processors requires 2p links and has a bisection width of 2p

    , supporting more simultaneous communications than a ring's p links and bisection width of 2. Hypercubes, with a bisection width of p/2, offer even higher connectivity, proven in real systems .
  - Contrast with Shared-Memory: Shared-memory systems rely on buses or crossbars, which face scalability challenges. Buses suffer from contention as processor count grows, leading to delays .

Crossbars, while faster, require p2 switches, making them prohibitively expensive for large systems. The limitations make shared-memory systems suitable only for small processor counts.

- Cost-Effectiveness:
  - Distributed-Memory Interconnects: Interconnects like toroidal meshes and hypercubes are cost-effective due to fewer links and simpler switches compared to crossbars. For instance, an omega network uses 2plog2(p) switches versus a crossbar's p2, reducing costs . Clusters, using commodity networks like Ethernet, further lower costs by leveraging standard hardware.
  - Shared-Memory Costs: The high cost of crossbars and the contention in buses make sharedmemory systems less viable for large-scale applications. The large crossbars are unusual due to expense.
- Handling Large-Scale Problems:
  - Distributed-Memory Suitability: The distributed-memory systems are better suited for problems requiring vast amounts of data or computation. Clusters, often hybrid systems with shared-memory nodes, can partition large datasets across private memories, minimizing contention and leveraging scalable interconnects. Explicit communication, though complex, allows fine-tuned data management for large-scale tasks like scientific simulations.
  - Shared-Memory Limitations: Shared-memory systems struggle with large-scale problems due to interconnect bottlenecks and cache coherence overhead. The maintaining coherence across many cores adds complexity and performance costs , and interconnect contention limits scalability .
- Hardware Simplicity:
  - No Cache Coherence in Distributed-Memory: Since each processor has private memory, distributed-memory systems avoid cache coherence issues, simplifying hardware design and reducing overhead. Shared-memory systems, conversely, require complex coherence mechanisms like snooping or directory-based protocols.
  - Hybrid Systems: The document notes that modern clusters often use shared-memory nodes, combining local shared-memory benefits with distributed-memory scalability, enhancing their suitability for large applications .

In conclusion, distributed-memory systems are preferred for large-scale parallel computing due to their scalable, cost-effective interconnects (e.g., toroidal meshes, hypercubes), lack of cache coherence overhead, and ability to handle massive data, as evidenced by their use in clusters and grids. Shared-memory systems, limited by expensive or contention-prone interconnects, are less suitable for such applications.

6. Coordinating the processes/threads,

Coordinating processes/threads involves managing interactions between concurrent tasks to ensure correct execution and efficient resource utilization.

Key Aspects:

1. Synchronization: Coordinating access to shared resources.
2. Communication: Exchanging data between processes/threads.
3. Mutual Exclusion: Ensuring exclusive access to shared resources.

Techniques:

1. Locks: Synchronizing access to shared resources.
2. Semaphores: Controlling access to resources.
3. Monitors: High-level synchronization constructs.

Importance:

1. Correctness: Ensures data consistency and program correctness.
2. Efficiency: Optimizes resource utilization and minimizes overhead.

Challenges:

1.Deadlocks: Processes/threads waiting for each other to release resources.

2. Starvation: Processes/threads unable to access resources due to others' monopolization. Effective coordination of processes/threads is crucial for building reliable and efficient concurrent systems.

1. **Explain the process of dividing work among processes/threads in parallel programs, including the concept of load balancing**

Answer: the process of dividing work among processes/threads as a fundamental step in parallelizing programs, particularly in single program, multiple data (SPMD) programs. The goal is to distribute the workload effectively to achieve optimal performance. For example, in an embarrassingly parallel task like adding two arrays (x[i] += y[i] for arrays x and y of size n), the programmer assigns subsets of array elements to each of the p processes/threads. Process/thread 0 might handle elements 0 to n/p-1, process/thread 1 handles n/p to 2n/p-1, and so on. This division requires two key considerations:

- **Load Balancing**: The work must be divided so that each process/thread receives roughly the same amount of work. Load balancing ensures no process/thread is idle while others are overloaded, maximizing resource utilization. In the array addition example, dividing n elements evenly across p processes/threads (approximately n/p elements each) achieves load balancing, as each element addition is computationally equivalent.
- **Minimizing Communication**: The division should reduce the need for communication among processes/threads, as communication (e.g., message-passing in distributed-memory systems or shared variable access in shared-memory systems) is typically expensive. In the array addition case, no communication is needed after the initial assignment, as each process/thread operates independently on its assigned elements.

The document notes that load balancing becomes challenging when the amount of work is not known in advance, such as in dynamically generated tasks. In such cases, programmers must design strategies to redistribute work during execution to maintain balance. The array addition example is "embarrassingly parallel" because the division is straightforward, requiring minimal coordination, but most problems demand additional synchronization and communication, complicating the parallelization process.

2. **Discuss why most parallel programs require synchronization and communication among processes/threads, and how these are interrelated**

   Answer parallel programs, most parallel programs require coordination through **synchronization** and **communication** among processes/threads to ensure correct execution and efficient performance. These tasks are critical because parallel programs often involve interdependent tasks, where the output of one process/thread affects another, necessitating coordination to avoid errors or inefficiencies.

   Need for Synchronization: Synchronization ensures that processes/threads execute in a coordinated manner, preventing issues like race conditions or incorrect data dependencies. For example, if multiple threads need to update a shared variable, synchronization ensures that updates occur in a controlled order to avoid data corruption. Without synchronization, the relative execution speeds of asynchronous processes/threads could lead to unpredictable outcomes.

   Need for Communication: Communication allows processes/threads to share data or results necessary for task completion. In distributed-memory programs, processes with private memories must explicitly exchange data (e.g., via message-passing). In shared-memory programs, threads communicate implicitly through shared variables, but this still requires coordination to ensure data consistency.

   Interrelation: The synchronization and communication are often interrelated. In **distributedmemory programs**, communication (e.g., a send/receive operation) implicitly synchronizes processes, as a receiving process waits for the sender's message, aligning their execution. For instance, in the message-passing example, process 0's Receive call synchronizes with process 1's Send, ensuring process 0 does not proceed until the message arrives. In **shared-memory programs**, synchronization facilitates communication by controlling access to shared variables. For example, a mutex synchronizes threads to ensure only one updates a shared variable (x += my_val) at a time, enabling safe communication of results. This interrelation complicates parallel program design, as programmers must balance synchronization to prevent errors with communication efficiency to minimize overhead, tailoring coordination strategies to the program's architecture (shared or distributed memory).

3. **Using the array addition example, illustrate how an embarrassingly parallel program achieves coordination, and contrast this with the coordination needs of more complex parallel programs.**

   Answer: **parallel** program for adding two arrays (x and y of size n) as an example of minimal coordination needs. The serial code is: double x[n], y[n]; for (int i = 0; i < n; i++)   x[i] += y[i];

To parallelize this with p processes/threads, the work is divided by assigning elements to each process/thread: process/thread 0 handles indices 0 to n/p-1, process/thread 1 handles n/p to 2n/p-1, and so on. The coordination involves:

- **Work Division**: The programmer ensures **load balancing** by dividing n elements evenly (approximately n/p per process/thread), as each addition operation is uniform in cost.
- **Minimal Communication**: No communication is needed during execution, as each process/thread operates independently on its assigned elements, accessing disjoint portions of x and y. In shared-memory systems, threads access shared arrays without conflict; in distributed-memory systems, arrays are pre-distributed to private memories.
- **No Synchronization**: Since there are no dependencies between tasks (each element addition is independent), synchronization is unnecessary, avoiding mechanisms like mutexes or barriers.

Contrast with Complex Programs: The document notes that most parallel programs are not embarrassingly parallel and require significant coordination . For example:

- **Synchronization Needs**: In the shared-memory example , updating a shared variable x (x += my_val) by multiple threads creates a race condition, requiring a **mutex** to ensure atomic updates. This serializes the critical section, increasing overhead and complexity compared to the array addition's lack of synchronization.
- **Communication Needs**: In the distributed-memory message-passing example , process 1 sends a message to process 0, which requires explicit Send and Receive calls. This communication synchronizes the processes and involves overhead, unlike the array addition's communication-free execution.
- **Dynamic Workloads**: Complex programs may generate work dynamically , requiring adaptive load balancing and frequent communication/synchronization to redistribute tasks, contrasting with the static, predictable division in array addition.

The array addition example achieves coordination through simple, static work division, making it "embarrassingly parallel" and efficient. Complex programs, however, demand intricate synchronization and communication strategies to manage dependencies and shared resources, significantly increasing design and performance challenges.

4. **Analyze the role of SPMD programs in coordinating processes/threads, and discuss how they support both data-parallelism and task-parallelism**

   Answer: **single program, multiple data (SPMD)** programs as a key paradigm for parallel computing, where a single executable runs on all cores but behaves differently based on conditional branches, such as if (I'm thread/process 0) do this; else do that. role in coordinating processes/threads, emphasizing their flexibility in implementing parallelism.

Role in Coordination:

- SPMD programs facilitate coordination by allowing a single codebase to manage diverse process/thread behaviors through rank-based logic. For example, in the message-passing pseudocode , processes use my_rank = Get_rank() to determine their actions: process 1 sends a message, while process 0 receives and prints it. This rank-based branching simplifies coordination, as all processes execute the same program but perform distinct tasks.
- Coordination involves dividing work, synchronizing, and communicating. SPMD supports work division by assigning tasks based on process/thread IDs, as in the array addition example , where each process/thread handles a subset of array elements. Synchronization and communication are managed through conditional logic, such as using Send/Receive in distributed-memory or mutexes in shared-memory to control shared variable access.

Support for Data-Parallelism:

- SPMD programs implement **data-parallelism** by dividing a dataset across processes/threads, each performing the same operation on different data. The array addition example demonstrates this: if process/thread 0 operates on the first half of the array (x[0:n/p1] += y[0:n/p-1]) and process/thread 1 on the second half (x[n/p:2n/p-1] += y[n/p:2n/p-1]), the same addition operation is applied to different data subsets, achieving parallelism with minimal coordination.

Support for Task-Parallelism:

- SPMD programs also support **task-parallelism** by assigning different tasks to processes/threads via conditional branches. The first SPMD example shows this: if (I'm thread/process 0) do this; else do that, where process/thread 0 performs one task and others perform another. In the message-passing example, process 1 creates and sends a message, while process 0 receives and prints it, illustrating distinct tasks coordinated within the same executable.

Analysis: ○ SPMD's flexibility in coordinating processes/threads stems from its ability to use a single program to handle both data-parallel and task-parallel workloads, simplifying development and maintenance compared to multiple executables. It supports load balancing by dividing data or tasks based on ranks and enables synchronization/communication through rank-specific logic.

- However, coordination complexity increases for non-embarrassingly parallel tasks, requiring careful management of synchronization (e.g., mutexes, barriers) and communication (e.g., message-passing), as seen in the shared-memory race condition example or distributedmemory communication.

SPMD's unified framework thus streamlines coordination while accommodating diverse parallelism models, making it a powerful approach for parallel program design in both shared- and distributed-memory systems.

7. Shared-memory,

Shared-memory is a parallel computing architecture where multiple processors or cores share a common memory space, allowing direct access to shared data.

Key Features:

1. Single memory space: All processors access the same memory.
2. Fast data access: Processors can directly access shared data.
3. Easy data sharing: Simplifies communication between processors.

Advantages:

1. Efficient data exchange: Reduced overhead for data transfer.
2. Easy programming: Familiar programming model for shared-memory access.

Challenges:

1. Cache coherence: Maintaining data consistency across multiple caches.
2. Synchronization: Coordinating access to shared resources.

Applications:

1. Multithreaded applications: Shared-memory suitable for multithreaded programs.
2. Multicore systems: Shared-memory architecture in multicore processors.

Shared-memory architecture enables efficient data sharing and communication between processors, making it suitable for various parallel computing applications.

1. **Explain the difference between dynamic and static thread paradigms in shared-memory programming, including their advantages and disadvantages.**

Answer:In shared-memory programs, two thread paradigms are discussed: dynamic and static.

Dynamic Thread Paradigm: In this approach, a master thread typically waits for work requests (e.g., over a network). When a request arrives, it forks a worker thread to handle the task. Once the task is completed, the worker thread terminates and joins the master thread.

- **Advantages**: This paradigm is resource-efficient because thread resources (e.g., stack, program counter) are only allocated while the thread is active, freeing them up when the thread terminates.
- **Disadvantages**: Forking and joining threads can be time-consuming, potentially impacting performance if threads are frequently created and destroyed.

Static Thread Paradigm: Here, all threads are forked after initial setup by the master thread and run until all work is completed. After completion, the threads join the master thread, which may perform cleanup before terminating.

- **Advantages**: This paradigm can offer better performance by avoiding the overhead of repeated forking and joining. It also aligns closely with distributed-memory programming paradigms, making it easier to adapt code across systems. ○ **Disadvantages**: It is less resource-efficient, as idle threads still hold resources (e.g., stack, program counter) that cannot be freed until the program ends.

The choice between paradigms depends on the application's needs, balancing resource efficiency against performance requirements.

2. **Discuss the concept of nondeterminism in shared-memory programs and provide an example of how it can lead to issues like race conditions. Explain how a mutex can address this issue.** **Answer**:Nondeterminism in shared-memory programs arises in MIMD (Multiple Instruction, Multiple Data) systems where threads execute asynchronously, causing the relative completion times of statements to vary across runs. This can lead to different outputs for the same input, as the order of thread execution is unpredictable.

Example of Nondeterminism and Race Condition: Consider two threads (Thread 0 and Thread 1) where Thread 0 has a private variable my_x = 7 and Thread 1 has my_x = 19. If both execute printf("Thread %d > my_x = %d\n", my_rank, my_x);, the output order is unpredictable (e.g., Thread 0's output may appear before or after Thread 1's). A more critical issue occurs with a shared variable x initialized to 0, where both threads execute my_val = Compute_val(my_rank); x += my_val;. If Thread 0's my_val = 7 and Thread 1's my_val = 19, a race condition can occur if both threads attempt to update x simultaneously. For instance, if Thread 0 loads x = 0, adds my_val = 7, and stores x = 7, but Thread 1 simultaneously loads x = 0, adds my_val = 19, and stores x = 19, the final value of x might be 19 instead of the correct sum (26), as one update overwrites the other.

Solution with Mutex: To prevent this race condition, a mutual exclusion lock (mutex) can be used to ensure the operation x += my_val is atomic, meaning only one thread can execute it at a time. The modified code would be: my_val = Compute_val(my_rank); Lock(&add_my_val_lock); x += my_val;
Unlock(&add_my_val_lock);

Here, a thread must acquire the mutex (Lock) before updating x and release it (Unlock) afterward. If one thread holds the lock, the other waits, ensuring that updates are serialized and the final value of x is correct (e.g., 7 + 19 = 26). However, this serialization reduces parallelism, so critical sections should be minimized and kept short.

3. **Describe the role of mutual exclusion in shared-memory programming and discuss alternatives to mutexes for ensuring thread synchronization.**

Answer:Mutual exclusion in shared-memory programming ensures that only one thread at a time can execute a critical section—a block of code accessing a shared resource—to prevent errors like race conditions. For example, if multiple threads attempt to update a shared variable x, mutual exclusion ensures each update is atomic, preserving data consistency.

The most common mechanism for mutual exclusion is a **mutex** (mutual exclusion lock). A mutex protects a critical section by requiring a thread to acquire the lock before entering the section and release it afterward. While a thread holds the lock, others attempting to access the critical section wait, ensuring exclusive access. For instance, to safely update a shared variable x, a thread executes: Lock(&add_my_val_lock);

```
x += my_val;
```

Unlock(&add_my_val_lock);

This prevents simultaneous updates that could lead to incorrect results. However, mutexes enforce serialization, which can reduce parallelism, so critical sections should be as short as possible.

Alternatives to Mutexes:

- **Busy-Waiting**: A thread enters a loop to check a condition, such as a shared variable ok_for_1. For example, Thread 1 waits until Thread 0 sets ok_for_1 = true before updating x. This is simple but wasteful, as the waiting thread consumes CPU resources without performing useful work.
- **Semaphores**: Similar to mutexes, semaphores provide synchronization but offer more flexibility for certain scenarios, such as signaling between threads. They are discussed further in Chapter 4 of the document.
- **Monitors**: These are higher-level constructs where an object's methods can only be executed by one thread at a time, providing mutual exclusion implicitly.

Each alternative has trade-offs: busy-waiting is resource-intensive, semaphores require careful management, and monitors may not be available in all environments. The choice depends on the specific synchronization needs and system constraints.

8. Distributed-memory

Distributed-memory is a parallel computing architecture where each processor or node has its own private memory, and data is exchanged through message passing.

Key Features:

1. Private memory: Each processor has its own memory space.
2. Message passing: Processors communicate by exchanging messages.
3. Scalability: Can handle large number of processors.

Advantages:

1. Scalability: Suitable for large-scale parallel applications.
2. Flexibility: Can handle diverse workloads and applications.

Challenges:

1. Communication overhead: Message passing can introduce latency.
2. Data consistency: Ensuring data consistency across nodes.

Applications:

1. High-performance computing: Distributed-memory suitable for large-scale simulations.
2. Cluster computing: Distributed-memory architecture in cluster systems.

Distributed-memory architecture enables scalability and flexibility, making it suitable for large-scale parallel computing applications.

1. **Explain the concept of message-passing in distributed-memory programming, including how it works and its key features, using the example.**

Answer: In distributed-memory programming, cores access only their private memories, necessitating explicit communication between processes. The most widely used approach is message-passing, which involves a send and a receive function to exchange data between processes identified by ranks (0, 1, ..., p-1, where p is the number of processes). The Message-Passing Interface (MPI) is the dominant API for this purpose.

How It Works: The document provides an example in pseudocode: char message[100]; my_rank = Get_rank(); if (my_rank == 1) {

```
        sprintf(message, "Greetings from process 1");
        Send(message, MSG_CHAR, 100, 0);
    } else if (my_rank == 0) {
        Receive(message, MSG_CHAR, 100, 1);
        printf("Process 0 > Received: %s\n", message); }
```

Here, process 1 creates a message ("Greetings from process 1") and sends it to process 0 using the Send function, specifying the message, its element type (MSG_CHAR), the number of elements (100), and the destination process rank (0). Process 0 calls Receive, specifying the storage for the message, the element type, the number of elements, and the sender's rank (1), then prints the received message. This example illustrates an SPMD (Single Program, Multiple Data) approach, where processes execute the same program but perform different actions based on their ranks.

Key Features:

- **SPMD Nature**: The same executable runs on all processes, with behavior determined by rank-based branching.
- **Local Memory**: Variables like message refer to distinct memory blocks on each process, emphasizing the private nature of memory.

- **Blocking Behavior**: Typically, Send blocks until the corresponding Receive begins, and Receive blocks until the message is received, though other behaviors are possible (e.g., Send copying data to a buffer and returning immediately).
- **Additional Functions**: Message-passing APIs like MPI provide collective communications (e.g., broadcast, reduction) and functions for managing processes or complex data structures. ○ **I/O Support**: Most implementations allow all processes to access stdout and stderr.

Message-passing is powerful and versatile, used in the world's most powerful computers, but it is low-level, requiring significant programmer effort to manage data distribution and program rewriting.

2. **Discuss the advantages and challenges of one-sided communication compared to messagepassing in distributed-memory systems.**

Answer: One-sided communication, or remote memory access, in distributed-memory systems allows a single process to update either its local memory with a value from another process or a remote process's memory with a local value, without requiring explicit participation from the other process. This contrasts with message-passing, where both a sender and a receiver must actively participate via Send and Receive functions.

Advantages of One-Sided Communication:

a **Simplified Communication**: Only one process needs to initiate the operation, reducing the coordination required compared to message-passing, where both processes must call functions.

b **Reduced Overhead**: By eliminating the need for a matching Receive or Send call, one-sided communication can reduce synchronization overhead and the cost of one function call. This can lead to faster communication in scenarios where only one process needs to act.

Challenges of One-Sided Communication:

c **Synchronization Needs**: For safe memory updates, the initiating process must know when it is safe to overwrite remote memory, and the receiving process must know when the update is complete. This may require synchronization before and after the operation or polling a flag variable set by the initiating process, which increases overhead. For example, if process 0 copies a value to process 1's memory, process 1 may need to repeatedly check a flag to detect the update, adding complexity.

d **Error-Prone**: Without explicit interaction between processes, errors in one-sided communication (e.g., overwriting the wrong memory location) can be difficult to debug, as there is no direct handshake to ensure correct operation.

e **Potential Overhead**: The need for synchronization or polling can negate some performance benefits, making the advantages harder to realize in practice compared to the structured communication of message-passing.

In contrast, message-passing, while more complex due to its explicit two-process interaction, provides a clearer structure for communication and synchronization, reducing the risk of subtle errors but increasing programmer effort. One-sided communication is thus appealing for its simplicity but requires careful management to achieve its potential efficiency.

3. **Describe how partitioned global address space (PGAS) languages attempt to bridge sharedmemory and distributed-memory programming, including the benefits and challenges. Answer**: Partitioned Global Address Space (PGAS) languages aim to combine the programmerfriendly aspects of shared-memory programming with the architecture of distributed-memory systems, where each core accesses only its private memory. They allow programmers to use sharedmemory techniques, such as accessing shared variables, while running on distributed-memory hardware.

How PGAS Works: The document illustrates this with a vector addition example:

```
shared int n = ...; shared double x[n], y[n];
private   int   i,   my_first_element,   my_last_element;
my_first_element  =  ...;  my_last_element  =  ...;  /*
Initialize x and y */ for (i = my_first_element; i <=
my_last_element; i++) {     x[i] += y[i];
}
```

Here, shared arrays x and y are declared, but each process is assigned specific elements based on its rank. PGAS languages allow programmers to control the distribution of shared data structures, ensuring that private variables (e.g., i) are allocated in local memory, and shared data is distributed to optimize local memory access.

Benefits:

- **Programmer Familiarity**: PGAS provides a shared-memory-like interface, which is more intuitive for many programmers compared to message-passing or one-sided communication.
- **Controlled Data Distribution**: Programmers can specify which elements of shared data reside in each process's local memory, avoiding the performance pitfalls of accessing remote memory, which can be hundreds or thousands of times slower than local access.
- **Improved Performance**: When shared data is allocated to the local memory of the executing core, operations like vector addition can be very fast, as shown in the example if x and y elements are locally assigned.

Challenges:

- **Performance Risks**: If data is poorly distributed—for instance, if all of x is on one core and all of y is on another—performance can be severely degraded due to frequent remote memory accesses.

- **Programmer Responsibility**: Unlike pure shared-memory systems, PGAS requires programmers to manage data distribution explicitly, increasing complexity compared to shared-memory programming where memory access is uniform.
- **Implementation Complexity**: Developing compilers and tools for PGAS languages is challenging, as they must balance shared-memory abstractions with the realities of distributed-memory hardware.

PGAS languages thus offer a promising compromise, providing shared-memory ease with distributed-memory efficiency, but they require careful data management to avoid performance issues, unlike the explicit but more predictable communication in messagepassing.

GPU programming, Programming hybrid systems, MIMD systems, GPUs, Performance – Speedup and efficiency in MIMD systems, Amdahl's law, Scalability in MIMD systems, Taking timings of MIMD programs, GPU performance.

2 Module

Module -2 Lecturer Notes: GPU programming, Programming hybrid systems, MIMD systems, GPUs, Performance

This Lecture Notes provides a comprehensive overview of **GPU** programming, Programming hybrid systems, MIMD systems, GPUs, Performance, based on Chapter 2 of "Peter S Pacheco, Matthew Malensek – An Introduction to Parallel Programming, second edition, Morgan Kauffman , VTU syllabus based BCS702 Parallel Computing" It is designed to serve as a detailed lecture presentation for a classroom setting, covering **GPU** programming, Programming hybrid systems, MIMD systems, GPUs, Performance, and practical examples, while ensuring clarity for students new to parallel computing.

MODULE-2: GPU programming, Programming hybrid systems, MIMD systems, GPUs, Performance

1. Speedup and efficiency in MIMD systems,
2. Amdahl's law,
3. Scalability in MIMD systems,
4. Taking timings of MIMD programs,
5. GPU performance.

1. Speedup and efficiency in MIMD systems,

In MIMD (Multiple Instruction, Multiple Data) systems, speedup and efficiency are key performance metrics.

Speedup:

1. Definition: Ratio of sequential execution time to parallel execution time.
2. Ideal speedup: Linear speedup, where speedup equals the number of processors.
3. Actual speedup: Limited by overheads, communication, and synchronization.

Efficiency:

1.Definition: Ratio of actual speedup to ideal speedup.
2.Measures utilization: Percentage of time processors spend on useful work. Factors affecting speedup and efficiency:

1. Parallelization overhead: Communication, synchronization, and task creation.
2. Load balancing: Even distribution of work among processors.
3. Scalability: Ability to handle increased workload.

Optimizing speedup and efficiency:

1. Minimize overhead: Reduce communication and synchronization.
2. Improve load balancing: Distribute work evenly among processors.
3. Optimize algorithms: Use parallel algorithms and data structures.

By understanding speedup and efficiency, developers can optimize MIMD systems for better performance and scalability.

1. **Explain the concepts of speedup and efficiency in MIMD systems, including their formulas and how they are affected by parallel overhead**
   Answer: speedup and efficiency are defined as key metrics for evaluating the performance of parallel programs in homogeneous MIMD (Multiple Instruction, Multiple Data) systems, where all cores have the same architecture. **Speedup** (S) is the ratio of the serial run-time (Tserial), the time taken by a serial program on a single core, to the parallel run-time (Tparallel), the time taken by the parallel program on p cores:
   S=TserialTparallel
   The ideal speedup, called **linear speedup**, occurs when S=p, meaning the parallel program runs p times faster than the serial program, achieved by dividing the serial work equally among p cores without introducing additional work.
   Efficiency (E) measures how well the cores are utilized and is defined as the speedup per core:
   E=Sp=Tserialp.Tparallel
   E=pS=p.TparallelTserial
   The ideal efficiency is 1.0, indicating that each core spends all its time solving the original problem. Efficiency can be interpreted as the fraction of the parallel run-time each core spends on the original problem, with the remainder being parallel overhead. For example, if Tserial=24 ms T, p=8, and Tparallel=4 ms, then:
   E=248.4=34
   E=8.424=43
   This means each core spends 34.4=3 ms on the problem and 4−3=1 ms on overhead.
   Parallel overhead prevents linear speedup because parallel programs introduce additional work not present in serial programs. In shared-memory systems, overhead includes mutex calls for critical sections, which serialize execution. In distributed-memory systems, overhead includes network communication, which is slower than local memory access. As the number of cores (p) increases, overhead typically grows—more threads compete for critical sections, or more processes transmit data—causing S to be less than p and E to decrease. Table 2.4 illustrates this: for p=2, S=1.9, E=0.95; for p=16, S=10.8, E=0.68, showing that efficiency drops as overhead increases with more cores.
   These metrics highlight the trade-off between parallelization benefits and overhead costs, guiding programmers to optimize work distribution and minimize overhead to approach linear speedup and high efficiency.

2. **Analyze the relationship between problem size and efficiency in MIMD systems, using the data from Table 2.5 and Figures 2.18 and 2.19**
   Answer: efficiency in MIMD systems is influenced by problem size, using Table 2.5 and Figures 2.18 and 2.19 to illustrate the relationship. Efficiency (E=Tserial/(p.Tparallel) represents the fraction of parallel run-time spent solving the original problem, with the rest being parallel overhead. Table 2.5 shows speedups (S) and efficiencies (E) for a parallel program run with different problem sizes (half, original, double) and core counts (p=1,2,4,8,16).

Table: Speedups and Efficiencies of parallel program on different problem sizes

	P	**1**	**2**	**4**	**8**	**16**
Half	S E	1.0 1.0	1.9 0.95	3.1 0.78	3.1 0.78	6.2 0.39
Original	S E	1.0 1.0	1.9 0.95	3.6 0.90	6.5 0.81	10.8 0.68
Double	S E	1.0 1.0	1.9 0.95	3.9 0.98	7.5 0.94	14.2 0.89

For the **original problem size**, efficiencies are 1.0 (p=1), 0.95 (p=2), 0.90 (p=4), 0.81 (p=8), and 0.68 (p=16), showing a decline as p increases due to growing parallel overhead (e.g., mutex contention or network communication). When the problem size is **halved**, efficiencies drop further: 0.95 (p=2), 0.78 (p=4), 0.60 (p=8), 0.39 (p=16), indicating that a smaller problem size exacerbates overhead's impact, as the fixed overhead consumes a larger proportion of the reduced computational work. Conversely, **doubling the problem size** improves efficiencies: 0.95 (p=2), 0.98 (p=4), 0.94 (p=8), 0.89 (p=16), approaching 1.0 for larger p, as the increased computational work (Tserial) grows faster than the overhead.

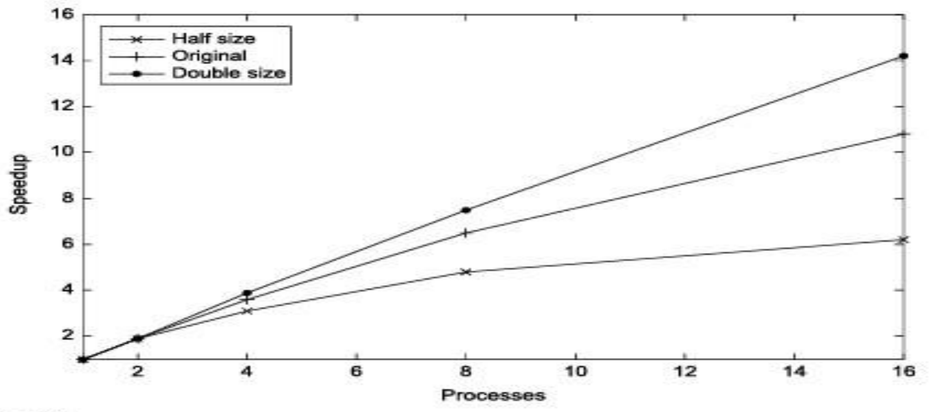


FIGURE 2.18

Speedups of parallel program on different problem sizes.

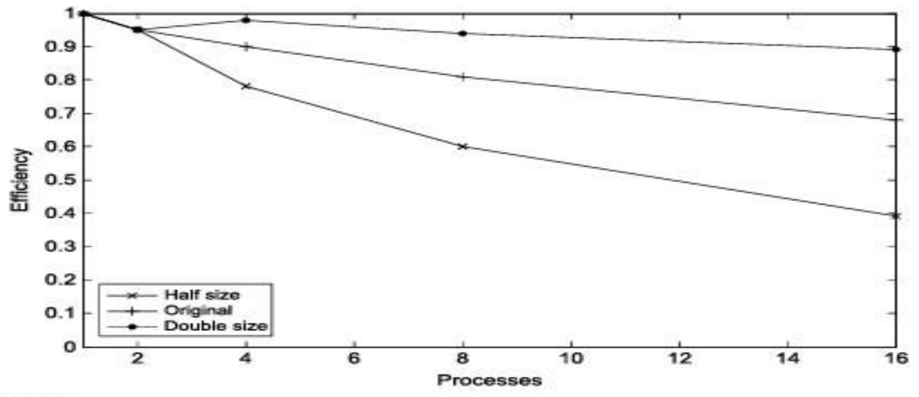


FIGURE 2.19

Efficiencies of parallel program on different problem sizes.

[Refer: "Peter S Pacheco, Matthew Malensek – An Introduction to Parallel Programming, second edition, Morgan Kauffman]

Figures 2.18 (speedups) and 2.19 (efficiencies) visually confirm this trend. In Figure 2.18, the speedup curves for larger problem sizes (double) are closer to the ideal linear speedup (S=p) than for smaller sizes (half), especially for higher p. Figure 2.19 shows efficiency curves where the double-size curve remains closer to 1.0, while the half-size curve drops sharply, particularly for p=8,16.
The behavior is common because, with a fixed number of cores, increasing problem size causes Tserial (time spent on the original problem) to grow much faster than Toverhead (e.g., communication or synchronization). For example, in matrix computations, Tserial may grow quadratically, while communication grows linearly, reducing overhead's relative impact. Thus, larger problem sizes enhance efficiency by amplifying the computational work relative to overhead, making parallel MIMD programs more effective for larger datasets.

3. **Discuss how parallel overhead impacts the performance of MIMD systems, and illustrate this with the example provided.**

Answer: parallel overhead is identified as a primary factor limiting the performance of MIMD systems, preventing linear speedup and reducing efficiency. Parallel overhead refers to additional work introduced by parallelization that is absent in the serial program, such as synchronization in shared-memory systems or communication in distributed-memory systems. The parallel run-time can often be expressed as:

Tparallel=Tserialp+Toverhead
Here, Tserial/p represents the ideal division of serial work among p cores, and Toverhead is the extra time due to parallelization. As p increases, Toverhead typically grows, reducing speedup (S=Tserial/Tparallel) and efficiency (E=Tserial/(p.Tparallel).

Impact of Overhead:
In shared-memory MIMD systems, overhead arises from critical sections requiring mutual exclusion mechanisms like mutexes. Mutex calls introduce function call overhead, and the serialization of critical section execution forces threads to wait, increasing Tparallel . In distributed-memory systems, overhead comes from transmitting data across the network, which is significantly slower than local memory access. As p increases, more threads compete for critical sections, or more processes exchange data, amplifying overhead. This causes S to deviate from the ideal p and E to decrease, as shown in Table : for p=8, S=6.5, E=0.81. Table – Speedups and efficiencies of a parallel program

p	**1**	**2**	**4**	**8**	**16**
S (Speedup)	1.0	1.9	3.6	6.5	10.8
E = S/p (Efficiency)	1.0	0.95	0.90	0.81	0.68

Illustrative Example:
An example to clarify overhead's impact. Suppose Tserial=24 ms, p=8, and Tparallel=4 ms. The efficiency is:

E=Tserialp.Tparallel=248.4=34=0.75

The time spent solving the original problem per core is:

E.Tparallel=34.4=3 ms

The overhead per core is:

Tparallel−E.Tparallel=4−3=1 ms

Thus, each core spends 3 ms on the original problem and 1 ms on overhead (e.g., mutex calls or communication). The total parallel run-time is:

Tparallel=Tserialp+Toverhead=248+1=3+1=4 ms

Tparallel=pTserial+Toverhead=824+1=3+1=4ms

This confirms that overhead (1 ms per core) increases Tparallel beyond the ideal

Tserial/p=3 ms, reducing speedup from 8 to:

S=TserialTparallel=244=6

This example illustrates how overhead, such as synchronization or communication, limits performance, emphasizing the need to minimize overhead to maximize speedup and efficiency in MIMD systems.

4 **Evaluate the significance of Table in understanding the performance of MIMD systems, and discuss how it reflects typical behavior in parallel programs.**

Answer: Table 2.4 (Refer Above)is a critical tool for understanding the performance of MIMD systems, as it presents speedup (S) and efficiency (E) values for a parallel program run with varying numbers of cores (p=1,2,4,8,16). The table provides concrete data to illustrate the impact of parallel overhead on performance, reflecting typical behavior in parallel programs.

Significance:

1. **Speedup Trends**: The table shows that speedup increases with p, but it falls short of the ideal linear speedup (S=p). For p=2, S=1.9≈2; for p=4, S=3.6≈4; but for p=16, S=10.8, significantly less than 16. This deviation highlights the presence of parallel overhead, which grows with more cores, reducing the performance gain from additional parallelism.
2. **Efficiency Decline**: Efficiency decreases as p increases, from 0.95 (p=2) to 0.68 (p=16). This reflects the increasing proportion of run-time spent on overhead (e.g., mutex calls in shared-memory systems or network communication in distributed-memory systems) rather than the original problem. Efficiency's decline quantifies how core utilization worsens with more parallelism.
3. **Practical Insights**: The data in Table 2.4, noted as derived from Chapter 3 (Tables 3.6 and 3.7), likely represents a real MPI program, making it a realistic example. It shows that while parallelization improves performance (higher S), the benefits diminish with larger p, a common challenge in MIMD systems where overhead limits scalability.

Typical Behavior:

The table reflects typical behavior in parallel programs. Most parallel programs introduce overhead not present in serial programs, such as:

1. **Shared-memory overhead**: Critical sections requiring mutexes serialize execution, and more threads increase contention, as seen in the efficiency drop from 0.90 (p=4) to 0.68 (p=16).
2. **Distributed-memory overhead**: Network communication slows data transfer, and more processes increase message exchanges, contributing to sublinear speedup.

As p increases, overhead typically grows, causing S/p to decrease, which Table 2.4 confirms. This behavior underscores the challenge of achieving linear speedup and high efficiency, emphasizing the need for careful program design to minimize overhead, such as reducing critical sections or optimizing communication patterns. Table 2.4 serves as a concise, datadriven illustration of these principles, guiding programmers in evaluating and optimizing MIMD system performance.

2. Amdahl's law,

Amdahl's Law is a theoretical bound on the maximum speedup that can be achieved by parallel processing. It states that the maximum speedup is limited by the fraction of the program that cannot be parallelized.

Key Points:

1. Sequential portion: The part of the program that cannot be parallelized.

2. Parallelizable portion: The part that can be divided among multiple processors.
3. Speedup limit: Determined by the sequential portion.

Formula:

Speedup = 1 / (1 - P + P/S) Where:

P = parallelizable portion

S = number of processors Implications:

1. Limitations: Even with infinite processors, speedup is limited by the sequential portion.
2. Optimization: Focus on reducing the sequential portion to achieve better speedup.

Amdahl's Law provides a fundamental understanding of the limits of parallel processing and guides optimization efforts.

1. **Explain Amdahl's law as described, including the derivation of the speedup formula and its implications for parallel programming.**

Answer: Amdahl's law, is an observation from the 1960s by Gene Amdahl that highlights the limitations of parallelization due to the unparallelizable portion of a program. It states that unless nearly all of a serial program is parallelized, the speedup achievable is significantly constrained, regardless of the number of cores used. The law is derived by considering the runtime contributions of parallelized and unparallelized parts of a program.

Suppose a serial program has a run-time Tserial, and a fraction 1−r (e.g., 90% or 0.9) is perfectly parallelized, meaning its run-time scales inversely with the number of cores p. The parallelized part's run-time becomes (1−r)×Tserial/p. The unparallelized part, fraction r (e.g., 10% or 0.1), cannot be sped up and retains its run-time r×Tserial. The total parallel run-time is:

Tparallel=(1−r)×Tserial/p+r×Tserial

Speedup (S) is the ratio of serial to parallel run-time:

S=TparallelTserial=(1−r)×Tserial/p+r×TserialTserial

Simplifying:

S=(1−r)/p+r1

As p becomes very large, (1−r)/p→0, so:

S≤r1

For example, if Tserial=20 seconds and 90% is parallelized (r=0.1), the parallel run-time is: Tparallel=0.9×20/p+0.1×20=18/p+2

Speedup is:

S=2018/p+2

S=18/p+220

As p→∞, 18/p→0, so Tparallel→2, and: S≤202=10

This means the speedup is capped at 10, even with thousands of cores, due to the 2-second unparallelized part.

Implications: Amdahl's law suggests that even small unparallelized fractions (e.g., r=1/100) limit speedup (e.g., to 100), posing a challenge for parallel programming. However, the document notes reasons to mitigate concern: (1) increasing problem size often reduces r, per

Gustafson's law; (2) many scientific programs achieve large speedups on distributed-memory systems; and (3) modest speedups (e.g., 5 or 10) may suffice if parallelization effort is low. These factors encourage continued development of parallel programs despite Amdahl's constraints.

2. **Using the example, illustrate how Amdahl's law limits speedup, and discuss the factors that mitigate its impact.**

Answer: an example to illustrate Amdahl's law, demonstrating how the unparallelized portion of a program caps speedup. In the example, a serial program has a run-time Tserial=20 seconds, with 90% perfectly parallelized and 10% unparallelized. The parallelized part's run-time is 0.9×20/p=18/p, and the unparallelized part's run-time is 0.1×20=2 seconds. The parallel run-time is:

Tparallel=18/p+2 Speedup

is:

S=TserialTparallel=2018/p+2

S=TparallelTserial=18/p+220

As the number of cores (p) increases, 18/p approaches 0, so Tparallel→2.

Thus, the speedup is bounded: S≤220=10

This shows that, despite perfect parallelization of 90% of the program, the speedup cannot exceed 10, even with 1000 cores, because the 2-second unparallelized part remains fixed. More generally, if the unparallelized fraction is r, the maximum speedup is 1/r. Here, r=0.1, so 1/0.1=10.

Limiting Mechanism: The unparallelized part (r×Tserial) creates a lower bound on Tparallel
, as it cannot be reduced regardless of p. As p grows, the parallelized part's contribution diminishes, but the unparallelized part dominates, capping speedup at 1/r. For example, even with infinite cores, the 2-second serial component ensures Tparallel≥2, limiting speedup.

Mitigating Factors: The three reasons not to be overly discouraged by Amdahl's law:

- **Problem Size Dependency**: The inherently serial fraction (r) often decreases as problem size increases, as described by Gustafson's law. For larger problems, the parallelizable work may grow disproportionately, reducing r and increasing the potential speedup beyond 1/r .
- **Real-World Success**: Thousands of scientific and engineering programs achieve significant speedups on large distributed-memory systems, suggesting that practical applications often overcome Amdahl's limitations through effective parallelization or problem-specific optimizations. ○ **Acceptable Speedups**: Small speedups (e.g., 5 or 10) can be sufficient, especially if parallelization requires minimal effort. In the example, a speedup of 10 is substantial for many applications, making parallelization worthwhile despite the cap.

These factors highlight that while Amdahl's law imposes theoretical limits, practical considerations and problem characteristics can mitigate its impact, encouraging continued exploration of parallel programming.

3. **Discuss the general formulation of Amdahl's law and its significance for parallel program design.**

Answer: Amdahl's law is presented as a fundamental principle that quantifies the maximum speedup achievable in a parallel program due to the presence of an unparallelizable serial component. The general formulation considers a serial program with run-time Tserial, where a fraction 1−r is perfectly parallelized (its run-time scales as (1−r)×Tserial/p) and a fraction r remains unparallelized (its run-time is r×Tserial). The parallel run-time is:

Tparallel=(1−r)×Tserial/p+r×Tserial

The speedup is:

S=TserialTparallel=Tserial(1−r)×Tserial/p+r×Tserial=1(1−r)/p+r
S=TparallelTserial=(1−r)×Tserial/p+r×TserialTserial=(1−r)/p+r1
As the number of cores (p) approaches infinity, (1−r)/p→0, so: S≤r1
This means the maximum speedup is limited to 1/r, where r is the fraction of the program that is inherently serial (cannot be parallelized). For example, if r=1/100, the speedup is capped at 100, even with thousands of cores.

Significance for Parallel Program Design:

1. **Focus on Parallelization**: Amdahl's law underscores the importance of maximizing the parallelizable portion of a program. Even a small serial fraction (e.g., r=0.05) limits speedup to 1/0.05=20, emphasizing the need to minimize serial code, such as I/O, initialization, or sequential algorithms.
2. **Scalability Challenges**: The law highlights that adding more cores yields diminishing returns if r is significant. Designers must optimize algorithms to reduce r, such as by parallelizing loops or using distributed data structures.
3. **Problem Size Consideration**: The document notes that Amdahl's law does not account for problem size, which can reduce r as size increases (per Gustafson's law). Designers should target larger problem sizes where parallel work dominates, improving speedup potential.
4. **Practical Trade-offs**: The law suggests tempering expectations for massive speedups but also recognizes that modest speedups (e.g., 5-10) may suffice if development effort is low. This encourages pragmatic design decisions, balancing parallelization effort with expected gains.
5. **Real-World Applicability**: Despite the law's limits, many scientific programs achieve large speedups on distributed-memory systems, indicating that careful design (e.g., minimizing serial bottlenecks) can mitigate its impact.

Amdahl's law serves as a guiding principle for parallel program design, urging developers to minimize serial components, optimize for larger problems, and weigh the cost-benefit of parallelization efforts, while acknowledging that practical successes often surpass theoretical constraints.

4. **Evaluate the limitations of Amdahl's law, and explain how they influence the development of parallel programs.**

Answer: Amdahl's law, which limits speedup to 1/r, where r is the unparallelized fraction of a serial program, but also discusses its limitations and mitigating factors that influence parallel program development. The law's restrictive nature arises from the fixed serial fraction constraining parallel run-time, but several limitations temper its pessimism.

Limitations of Amdahl's Law:

1. **Problem Size Omission**: Amdahl's law assumes a fixed problem size, ignoring that the inherently serial fraction (r) often decreases as problem size grows. Gustafson's law formalizes this, suggesting that larger problems increase the parallelizable portion, raising the speedup ceiling. For example, in matrix computations, initialization (serial) may be fixed, while computation (parallel) grows quadratically, reducing r .
2. **Empirical Successes**: Despite theoretical caps, thousands of scientific and engineering programs achieve significant speedups on large distributed-memory systems. This indicates that real-world applications often have small or manageable r, or they employ techniques (e.g., overlapping communication) to mitigate serial bottlenecks.

3. **Acceptability of Modest Speedups**: The law's focus on maximum speedup overlooks that smaller speedups (e.g., 5 or 10) can be valuable, especially if parallelization is straightforward. A speedup of 10, as in the example where r=0.1, may justify development for time-sensitive applications.

Influence on Parallel Program Development:

1. **Targeting Larger Problems**: The problem size limitation encourages developers to design programs for larger datasets, where r is smaller. For instance, in simulations, serial setup time becomes negligible compared to parallel computation, aligning with Gustafson's law and improving speedup potential.
2. **Optimizing Serial Code**: Awareness of Amdahl's limits drives efforts to minimize r. Developers may parallelize I/O, use asynchronous communication, or redesign algorithms to reduce serial sections, such as replacing sequential reductions with treebased approaches.
3. **Leveraging Distributed Systems**: The success of large-scale programs inspires development for distributed-memory systems, where massive core counts can exploit parallelism if r is small. Techniques like data partitioning and load balancing help achieve high speedups.
4. **Pragmatic Development**: The acceptability of modest speedups encourages developers to pursue parallelization even when complete parallelization is infeasible. For low-effort parallelizations (e.g., using OpenMP for loops), a speedup of 5-10 justifies the investment, broadening parallel programming's applicability.
5. **Balancing Effort and Gains**: Amdahl's law prompts developers to evaluate the cost of reducing r. If parallelizing a small serial fraction requires significant effort for marginal speedup, they may prioritize other optimizations or accept limited gains.
6. By recognizing these limitations, developers are motivated to design scalable programs for larger problems, optimize serial components, and adopt practical strategies that align with real-world successes, ensuring that Amdahl's law informs but does not deter parallel program development.

4. Scalability in MIMD systems,

Scalability in MIMD (Multiple Instruction, Multiple Data) systems refers to the ability to handle increased workload or larger problem sizes by adding more processors or nodes.

Key aspects:

Types of Scalability:

1. Strong scaling: Increasing processors for a fixed problem size.
2. Weak scaling: Increasing problem size with proportional increase in processors. Factors affecting scalability:

1. Communication overhead: Inter-processor communication can limit scalability.
2. Load balancing: Even distribution of work among processors.
3. Synchronization: Coordinating processes can impact performance. Benefits of scalability:

1. Increased performance: Handle larger problem sizes or workloads.
2. Improved efficiency: Better resource utilization.
3. Flexibility: Accommodate growing demands.

Challenges:

1. Scalability bottlenecks: Identifying and addressing limitations.
2. Efficient algorithms: Designing algorithms that scale well.

By achieving scalability, MIMD systems can efficiently handle complex computations and large datasets.

1. **Explain the concept of scalability in MIMD systems, including the difference between its informal and formal definitions.**

Answer: scalability in MIMD systems is described as a measure of a parallel program's ability to improve performance with increased system resources. The **informal definition** states that a program is scalable if, by increasing the system's power, such as adding more cores, it achieves speedups compared to running on a less powerful system (e.g., fewer cores). This intuitive notion focuses on performance gains through hardware upgrades.

The **formal definition** is more precise: a program is scalable if, when the number of processes or threads is increased, the problem size can be adjusted at a corresponding rate to maintain constant efficiency (E). Efficiency is defined as
E=S/p=Tserial/(p.Tparallel), where S is speedup, Tserial is serial run-time, Tparallel is parallel run-time, and p is the number of processes/threads. Scalability requires that as p grows, the problem size can be scaled to keep E unchanged, ensuring consistent core utilization.
The informal definition is broader and qualitative, emphasizing performance improvement, while the formal definition is quantitative, tying scalability to efficiency and problem size adjustments. The formal approach provides a rigorous framework for evaluating parallel programs, ensuring that scalability accounts for both computational work and parallel overhead.

2. **Using the example, demonstrate how a program is determined to be scalable, and explain the mathematical derivation.**

Answer: an example to illustrate scalability in MIMD systems. Consider a program where the serial run-time is Tserial=n microseconds, with n also representing the problem size, and the parallel run-time is Tparallel=n/p+1, where p is the number of processes/threads. The efficiency is:

E=Tserialp.Tparallel=np(n/p+1)=nn+p

E=p.TparallelTserial=p(n/p+1)n=n+pn

To test scalability, the number of processes/threads is increased by a factor k, so the new number of processes is kp. The problem size is scaled by a factor x, making the new problem size xn. The goal is to find x such that the efficiency remains unchanged:

E=nn+p=xnxn+kp

E=n+pn=xn+kpxn

Solving for x: nn+p=xnxn+kp

n+pn=xn+kpxn Cross-multiplying:

n(xn+kp)=xn(n+p)

xn2+knp=xn2+xnp

Subtract xn2: knp=xnp

Divide by np (since n≠0, p≠0):

x=k

Thus, if the problem size is increased by the same factor k as the number of processes/threads, the efficiency remains n/(n+p). For example, if p doubles (k=2), doubling the problem size (x=2) yields:

E=2n2n+2p=2n2(n+p)=nn+p

E=2n+2p2n=2(n+p)2n=n+pn

This confirms the program is scalable, as efficiency is maintained. The derivation shows that scaling the problem size proportionally to the process count balances computational work and overhead, satisfying the formal definition of scalability.

3. **Discuss the concepts of strong and weak scalability in MIMD systems, and classify the example program accordingly.**

Answer: The two specific types of scalability in MIMD systems: **strong scalability** and **weak scalability**, which describe how a parallel program maintains efficiency under different conditions as the number of processes/threads increases.

Strong Scalability: A program is strongly scalable if efficiency (E) remains fixed when the number of processes/threads (p) increases without changing the problem size. This implies that the parallel program can handle more cores effectively for a fixed workload, with parallel overhead growing minimally or being offset by perfect work division. Strong scalability is challenging because overhead (e.g., communication or synchronization) often increases with p, reducing efficiency unless the program is highly optimized.

Weak Scalability: A program is weakly scalable if efficiency remains fixed when both the number of processes/threads and the problem size increase at the same rate. This means that as p grows, the problem size is scaled proportionally (e.g., by a factor k), allowing each process to handle a constant amount of work. Weak scalability is more achievable, as increasing the problem size often outpaces overhead growth, maintaining efficiency.

In the example provided, where Tserial=n, Tparallel=n/p+1, and E=n/(n+p), the program is tested for scalability by increasing processes from p to kp and problem size from n to xn. The derivation shows that setting x=k keeps efficiency constant, as E=xn/(xn+kp)=n/(n+p) when x=k. Since efficiency is maintained by scaling the problem size at the same rate as the process count, the program is **weakly scalable**. It is not strongly scalable, as the document does not demonstrate constant efficiency for a fixed problem size with increasing p, and the constant overhead term (+1 in Tparallel) suggests efficiency would decrease without scaling n.

These concepts guide program design: strong scalability is ideal for fixed-size problems but rare, while weak scalability suits applications where larger problems are feasible, aligning with many scientific computations.

4. **Analyze the significance of the example for understanding scalability in MIMD systems, and discuss its implications for parallel program design.**

Answer: The significant for understanding scalability in MIMD systems, as it provides a concrete illustration of the formal definition of scalability and its practical implications. The example assumes Tserial=n, where n is the problem size, and Tparallel=n/p+1, yielding efficiency E=n/(n+p). Scalability is tested by increasing processes from p to kp and finding the problem size scaling factor x that maintains E: n+pn=xn+kpxn

Solving shows x=k, meaning efficiency remains constant if the problem size scales with the number of processes. This demonstrates **weak scalability**, as efficiency is preserved by increasing the problem size proportionally.

Significance:

1. **Clarity of Formal Definition**: The example operationalizes the formal definition of scalability—maintaining constant efficiency by adjusting problem size—making it accessible. It shows how scalability depends on balancing computational work (n/p) and overhead (+1).
2. **Practical Illustration**: The mathematical derivation (x=k) quantifies how problem size must scale, offering a clear guideline for achieving scalability. This is realistic for applications like simulations, where problem size can grow with resources.
3. **Weak Scalability Focus**: By classifying the program as weakly scalable, the example highlights a common scenario in parallel computing, where scaling workload with resources is feasible, unlike strong scalability, which is harder to achieve.

Implications for Parallel Program Design:

1. **Problem Size Scaling**: Designers should anticipate larger problem sizes to maintain efficiency as core counts grow, aligning with weak scalability. For example, in data-intensive applications, increasing dataset size can leverage more cores effectively.
2. **Overhead Management**: The constant overhead (+1) in the example suggests that minimizing fixed overheads (e.g., synchronization or communication setup) is crucial for scalability, especially for small n.
3. **Targeting Weak Scalability**: Since weak scalability is more achievable, developers should design algorithms that scale workload with resources, such as partitioning data evenly across processes.
4. **Application Context**: The example implies that scalability is contextdependent. Programs for fixed-size problems may struggle with scalability, while those for variable-size problems (e.g., scientific modeling) are better suited for MIMD systems.

The example underscores that scalability is achievable through careful workload scaling, guiding developers to design programs that exploit increased resources efficiently, particularly for large-scale, data-driven applications in MIMD environments.

5. Taking timings of MIMD programs,

Taking timings of MIMD (Multiple Instruction, Multiple Data) programs involves measuring the execution time to evaluate performance.

Why take timings?

1. Performance evaluation: Understand program efficiency.
2. Optimization: Identify bottlenecks.
3. Comparison: Compare different algorithms or implementations.

Methods:

1. Wall clock time: Measure elapsed time.
2. CPU time: Measure processor time used.
3. Profiling tools: Utilize tools to analyze performance.

Challenges:

1. Synchronization: Coordinating timing measurements across processes.
2. Overhead: Minimizing measurement overhead.
3. Variability: Accounting for system variability. Best practices:

1. Use high-resolution timers: Precise measurements.
2. Average multiple runs: Reduce variability.

3. Profile specific code sections: Target optimization efforts.

By accurately taking timings, developers can optimize MIMD programs for better performance and efficiency.

1Explain the different purposes of taking timings in MIMD programs, and discuss how these purposes influence the timing approach.

Answer: The two primary purposes for taking timings in MIMD programs, each influencing the timing approach distinctly. The first purpose is **during program development**, where timings help verify if the program behaves as intended. For example, in a distributed-memory program, developers may measure time spent waiting for messages to identify design or implementation flaws, such as excessive communication delays. This requires **detailed timings** of specific program segments (e.g., communication vs. computation), often necessitating fine-grained instrumentation to pinpoint inefficiencies.

The second purpose is **post-development performance evaluation**, where the goal is to assess the program's overall efficiency or speedup. Here, a **single, aggregate time** is typically reported, representing the run-time of the critical section (e.g., sorting in a bubble sort program, excluding I/O). This approach focuses on a holistic metric, like wall clock time for the main computation, to compare against serial run-times or other parallel implementations.

Influence on Timing Approach:

1. **Developmental timings** demand high-resolution timers and detailed logging, possibly outputting multiple time measurements per process/thread to analyze bottlenecks. For instance, developers might use API-specific profiling tools to break down time spent in MPI communication versus local computation.
2. **Performance evaluation timings** prioritize simplicity, using a single wall clock time measurement. The document suggests a basic code modification:

double start, finish; start = Get_current_time(); /* *Code that we want to time* */ finish = Get_current_time();
printf("The elapsed time = %e seconds\n", finish-start);

However, for parallel programs, a more robust approach synchronizes processes/threads and reports the maximum elapsed time (see Question 4 below). The distinction ensures that developmental timings guide optimization, while performance timings provide a standardized metric for reporting, aligning with the program's evaluation goals.

10. **Discuss why wall clock time is preferred over CPU time for timing MIMD programs, and provide an example to illustrate this preference.**

Answer: The explains why **wall clock time** is preferred over **CPU time** for timing MIMD programs, emphasizing its ability to capture real-world execution costs. **CPU time**, as reported by the C function clock, measures only the time spent executing program code, including user code, library functions (e.g., pow, sin), and system calls (e.g., printf, scanf). It excludes **idle time**, such as when a process is waiting for external events, which can be significant in parallel programs.

In contrast, **wall clock time** measures the elapsed time from start to finish of a code segment, including idle periods. This is critical for MIMD programs, where processes/threads often wait, such as in distributed-memory systems where a process calling a receive function waits for a matching send. Excluding this idle time, as CPU time does, would underreport the actual run-time, giving a misleading performance picture. For example, if a process consistently waits for messages, ignoring this wait time would falsely suggest better performance.

Example: The distributed-memory scenario: "in a distributed-memory program, a process that calls a receive function may have to wait for the sending process to execute the matching send, and the operating system might put the receiving process to sleep while it waits." If CPU time were used, this waiting period would not be counted, as no program code is active. However, wall clock time, captured using functions like MPI_Wtime or omp_get_wtime, includes this wait, reflecting the true cost. The suggested timing code: double start, finish; start = Get_current_time(); /* *Code that we want to time* */ finish = Get_current_time();

printf("The elapsed time = %e seconds\n", finish-start);

uses wall clock time to ensure all execution costs, including waiting, are measured, aligning with the need for accurate performance evaluation in MIMD systems.

2.Describe the process for accurately timing parallel MIMD programs, including the challenges and solutions.

Answer: A method for accurately timing parallel MIMD programs, addressing challenges in capturing a single, representative run-time across multiple processes/threads. The process involves synchronizing processes, measuring individual execution times, and aggregating them to report the total elapsed time from the first process/thread's start to the last's finish. **Process**:

1. **Synchronization**: All processes/threads execute a Barrier() function to approximately align their start times, ensuring measurements begin simultaneously.
2. **Individual Timing**: Each process/thread records its start time using Get_current_time() (e.g., MPI_Wtime or omp_get_wtime), executes the code, and records its finish time. The elapsed time is calculated as my_elapsed = my_finish - my_start.
3. **Aggregation**: A global maximum function, Global_max(my_elapsed), computes the largest elapsed time across all processes/threads, representing the total run-time from the earliest start to the latest finish.
4. **Reporting**: Process/thread 0 prints the result using: shared double global_elapsed;

```
private double my_start, my_finish, my_elapsed;
Barrier(); my_start = Get_current_time(); /*
Code that we want to time */ my_finish =
Get_current_time(); my_elapsed = my_finish -
my_start;          global_elapsed          =
Global_max(my_elapsed);
if (my_rank == 0) {
   printf("The elapsed time = %e seconds\n", global_elapsed);
}
```

Challenges:

1. **Clock Synchronization**: Clocks on different nodes may not be synchronized, making exact start-to-finish timing across nodes impossible. The barrier function only ensures all processes have started the call, not simultaneous completion.
2. **Timing Variability**: Run-times vary across executions due to system factors, even with identical inputs and systems.
3. **Timer Resolution**: Some timers have millisecond resolution (10^{-3} seconds), while instructions execute in nanoseconds (10^{-9} seconds), requiring many instructions for a nonzero measurement.

Solutions:

1. **Barrier Approximation**: The barrier function provides a practical compromise for synchronization, sufficient for most performance evaluations.
2. **Minimum Time Reporting**: To address variability, the minimum time across multiple runs is reported, as external events are unlikely to improve the best possible run-time.
3. **High-Resolution Timers**: Programmers must check timer resolution using APIprovided functions or specifications, ensuring measurements capture short events. APIs like MPI and OpenMP provide high-resolution wall clock timers.
4. **Single-Thread-per-Core**: Running one thread per core minimizes scheduling overhead and variability, ensuring consistent timings.

a. This process ensures accurate, representative timings for parallel MIMD programs, addressing synchronization and variability challenges to provide reliable performance metrics.

3. Evaluate the considerations for avoiding common pitfalls in timing MIMD programs, and discuss their importance.

Answer: The several considerations for avoiding common pitfalls when timing MIMD programs, ensuring accurate and meaningful performance measurements. These considerations address inappropriate metrics, timing scope, and system-induced variability, each critical for reliable evaluation.

Considerations and Pitfalls:

5. **Avoiding Total Program Timing**:
   1. **Pitfall**: Using tools like the Unix time command measures the entire program, including irrelevant sections like I/O, which may not reflect the performance of the core computation (e.g., sorting in a bubble sort program).
   2. **Solution**: Focus on specific code segments using custom timing code with Get_current_time() (e.g., MPI_Wtime or omp_get_wtime), as shown:
      double start, finish; start = Get_current_time(); /* *Code that we want to time* */ finish = Get_current_time();
      printf("The elapsed time = %e seconds\n", finish-start);
   3. **Importance**: Excluding I/O or setup ensures timings reflect the parallelized computation, aligning with performance evaluation goals.
6. **Using Wall Clock Time Instead of CPU Time**:
   1. **Pitfall**: CPU time, reported by clock, excludes idle periods (e.g., waiting for messages in distributed-memory programs), underestimating true run-time.
   2. **Solution**: Use wall clock time, which includes all elapsed time, capturing real costs like communication waits, as supported by MPI and OpenMP timers.
   3. **Importance**: Wall clock time provides a comprehensive view of performance, critical for identifying bottlenecks like synchronization delays in parallel programs.
7. **Handling Timing Variability**:
   1. **Pitfall**: Run-times vary across executions due to system factors, and reporting mean or median times may include inflated values from external interference.
   2. **Solution**: Report the minimum time across multiple runs, as it best approximates the program's optimal performance.

3. **Importance**: Minimum time reporting ensures results are not skewed by transient system loads, providing a consistent performance baseline.

4. **Avoiding Multiple Threads per Core**:
   1. **Pitfall**: Running multiple threads per core increases scheduling overhead and timing variability, degrading performance and complicating measurements.
   2. **Solution**: Limit to one thread per core to minimize variability and overhead.
   3. **Importance**: Single-thread-per-core execution ensures stable, reproducible timings, reflecting the program's true parallel efficiency.
5. **Checking Timer Resolution**:
   1. **Pitfall**: Low-resolution timers (e.g., milliseconds) may fail to measure short events, requiring millions of instructions for a nonzero reading.
   2. **Solution**: Verify timer resolution using API functions or specifications, ensuring suitability for the program's execution scale.
   3. **Importance**: High-resolution timers ensure precise measurements, critical for evaluating fine-grained parallel tasks.
6. **Excluding High-Performance I/O**:
   1. **Pitfall**: Including I/O in timings can skew results, as programs are typically not optimized for high-performance I/O.
   2. **Solution**: Exclude I/O from reported run-times, focusing on computational performance.
   3. **Importance**: Isolating computation ensures timings reflect parallel algorithm efficiency, not I/O bottlenecks.

Overall Importance: These considerations prevent misinterpretation of performance data, ensuring timings accurately reflect the program's parallel efficiency. By avoiding irrelevant metrics (e.g., CPU time, I/O), mitigating variability, and using appropriate tools, developers obtain reliable insights for optimization and reporting, crucial for advancing parallel computing research and applications.

12. GPU performance.

GPU performance refers to the efficiency and speed of a Graphics Processing Unit (GPU) in executing tasks, particularly in parallel computing and graphics rendering.:

Factors affecting GPU performance:

1. CUDA cores/Stream processors: Number and type of processing units.
2. Memory bandwidth: Data transfer rate between GPU memory and processors.
3. Memory capacity: Available memory for data storage.

Key performance metrics:

1. Floating-point operations per second (FLOPS): Measure of computational performance.
2. Memory bandwidth: Measure of data transfer rate.
3. Frame rate: Measure of graphics rendering performance. Optimization techniques:
1. Parallelization: Utilizing multiple cores and threads.
2. Memory optimization: Minimizing data transfer and maximizing memory locality.
3. Instruction-level parallelism: Optimizing instruction execution.

Applications:

1. Deep learning: Accelerating neural network computations.
2. Scientific simulations: Simulating complex phenomena.

3. Graphics rendering: Enhancing visual performance.

By understanding and optimizing GPU performance, developers can unlock the full potential of these powerful processing units.

1. **Explain why traditional MIMD performance metrics like efficiency and linear speedup are not applicable to GPU programs,.**

Answer: traditional MIMD performance metrics, such as **efficiency** and **linear speedup**, are not typically applied to GPU programs due to fundamental differences between GPU and CPU architectures. In MIMD systems, efficiency (E=S/p=Tserial/(p.Tparallel)) assumes the serial program runs on the same type of core as the parallel system, allowing efficiency to represent core utilization. However, **GPUs use inherently parallel cores**, unlike conventional CPUs, which are designed for serial execution. Comparing a GPU program's performance to a serial program running on a GPU core is inappropriate, as GPU cores are not optimized for serial tasks. Thus, efficiency, which relies on this comparison, "usually doesn't make sense for GPUs."

Similarly, **linear speedup** (S=p) in MIMD systems assumes perfect work division across p identical cores, with the serial baseline on the same core type. Since GPU cores are "fundamentally different from conventional CPUs," a serial CPU program cannot serve as a meaningful baseline for GPU speedup. The architectural mismatch—GPUs excelling in parallel tasks versus CPUs in serial tasks—makes linear speedup relative to a CPU serial program irrelevant. Instead, GPU performance is evaluated by **speedups over serial CPU programs or MIMD programs**, focusing on practical performance gains rather than theoretical metrics tied to identical core types.

Implications: This distinction encourages developers to focus on raw speedup metrics for GPUs, comparing against CPU-based baselines, and to avoid metrics that assume architectural homogeneity, aligning evaluation with GPU's parallel nature.

2. **Discuss how Amdahl's law applies to GPU programs, including its limitations and mitigating factors.**

Answer: **Amdahl's law** applies to GPU programs when the inherently serial portion of the program (fraction r) is executed on a conventional serial processor, such as a CPU.

Amdahl's law states that the maximum speedup is limited to less than 1/r . For example, if r=0.1 (10% serial), the speedup cannot exceed 1/0.1=10, regardless of the GPU's parallel processing power. This occurs because the serial part, running on the CPU, remains a fixed bottleneck, while the parallel part, executed on the GPU, scales with available resources. **Limitations and Mitigating Factors**:

1. **Problem Size Dependency**: Amdahl's law does not account for problem size. As problem size increases, the "inherently serial" fraction (r) often decreases, reducing the bottleneck's impact. This is supported by Gustafson's law, which suggests that larger problems increase the parallelizable portion, raising the speedup ceiling.
2. **Real-World Successes**: Many GPU programs achieve "huge speedups" in practice, indicating that serial fractions are often small or manageable in applications like scientific computing or machine learning, where parallel tasks dominate.
3. **Adequacy of Small Speedups**: Even modest speedups (e.g., 5 or 10) can be sufficient if the development effort is low, making Amdahl's theoretical limit less discouraging for practical applications.

Significance: While Amdahl's law imposes a theoretical cap on GPU performance, these mitigating factors encourage developers to target larger problems, optimize serial code, and leverage GPU parallelism. The law serves as a reminder to minimize CPU-bound serial work but does not deter GPU programming, given practical successes and the acceptability of moderate speedups.

3. **Describe the timing considerations for GPU programs, and compare them to MIMD program timing.**

Answer: timing considerations for GPU programs, emphasizing simplicity for most cases and similarities with MIMD program timing. For GPU programs, timing typically involves:

1. **Standard Approach**: Since GPU programs are "ordinarily started and finished on a conventional CPU," a CPU timer is used to measure the entire GPU execution. The timer starts before the GPU part begins and stops after it completes, capturing the wall clock time of the GPU workload. This is straightforward for programs where the GPU handles the main computation.
2. **Subset Timing**: To time specific GPU code subsets, a timer defined by the GPU API (e.g., CUDA's event API) is required, as CPU timers cannot isolate GPU kernel execution.
3. **Complex Scenarios**: Programs running on multiple CPU-GPU pairs require more care due to coordination challenges, but the document excludes these from discussion. **Comparison to MIMD Program Timing**:
4. **Similarities**:
1. Both prefer **wall clock time** over CPU time to capture all execution costs, including idle periods (e.g., waiting for GPU kernels or MIMD messages).
2. Both focus on specific code segments, excluding I/O, to evaluate computational performance (e.g., sorting in MIMD, kernel execution in GPU).
3. Both face **timing variability**, suggesting the minimum time across runs as the reported value to avoid external interference.

5. **Differences**:

1. **Synchronization**: MIMD timing requires explicit synchronization (e.g., Barrier()) and aggregation (e.g., Global_max()) to measure the time from the first process/thread's start to the last's finish across multiple cores. GPU timing is simpler, as a single CPU timer often suffices for the entire GPU workload, avoiding multi-core synchronization.
2. **API Dependency**: MIMD uses API-specific timers like MPI_Wtime or omp_get_wtime, while GPU programs may use CPU timers for overall timing or GPUspecific timers for kernel-level measurements.
3. **Complexity**: MIMD timing is more complex due to distributed or shared-memory coordination, whereas GPU timing benefits from the CPU's centralized control, except in multi-GPU scenarios.

Implications: GPU timing is generally easier for single-GPU programs, leveraging CPU timers, but requires GPU-specific tools for detailed analysis. Both approaches prioritize accurate, representative measurements, but GPU timing avoids the multi-core synchronization overhead inherent in MIMD systems.

12. **Evaluate the significance of scalability and performance evaluation for GPU programs, and discuss their implications for program design.**

Answer: **scalability** and **performance evaluation** for GPU programs, highlighting their unique considerations compared to MIMD systems and their implications for program design. **Scalability**:

1. **Definition**: The document uses an **informal definition** of scalability for GPUs: a program is scalable if increasing the GPU size (e.g., more cores or larger memory) yields speedups over its performance on a smaller GPU. The **formal MIMD scalability definition**—maintaining constant

efficiency by scaling problem size with core count— does not apply, as GPU efficiency is not meaningful due to architectural differences between GPU and CPU cores.

2. **Significance**: This informal scalability emphasizes practical performance gains, encouraging developers to design programs that exploit larger GPU resources. For example, increasing GPU cores for image processing can reduce computation time if the workload is parallelized effectively.
3. **Implications**: Designers must optimize for GPU parallelism, ensuring workloads scale with GPU size. This involves maximizing data parallelism, minimizing data transfers between CPU and GPU, and using scalable algorithms like parallel reductions.

Performance Evaluation:

1. **Approach**: GPU performance is evaluated by **speedup** compared to serial CPU programs or parallel MIMD programs, rather than efficiency or linear speedup. This reflects the GPU's parallel architecture, where raw performance gains are prioritized.
2. **Amdahl's Law**: If the serial part (fraction r r r) runs on the CPU, speedup is capped at 1/r 1/r 1/r, but mitigated by decreasing r r r with larger problem sizes, real-world successes, and the adequacy of modest speedups.
3. **Timing**: Simple CPU timers suffice for overall GPU execution, but GPU-specific timers are needed for detailed kernel analysis, ensuring accurate performance measurement.
4. **Significance**: This evaluation framework highlights the importance of practical benchmarks (speedup) over theoretical metrics (efficiency), aligning with GPU's role in high-performance computing. It also underscores the need to minimize CPU bottlenecks, as per Amdahl's law.
5. **Implications**: Developers should:
    1. **Optimize Serial Code**: Minimize CPU-bound serial tasks (e.g., data preprocessing) to reduce r, enhancing speedup.
    2. **Target Large Problems**: Design for larger datasets where parallel tasks dominate, leveraging Gustafson's law to reduce r.
    3. **Use Appropriate Timers**: Employ GPU API timers for kernel-level insights, ensuring precise optimization.
    4. **Focus on Parallelism**: Structure programs to maximize GPU core utilization, using techniques like coalesced memory access and thread synchronization.

Overall Impact: The informal scalability and speedup-focused evaluation encourage GPU program designs that prioritize parallelism and scalability with hardware upgrades. By addressing Amdahl's limits and leveraging practical successes, developers can achieve significant performance gains, particularly for large-scale, data-parallel applications like machine learning or scientific simulations, aligning with GPU's architectural strengths.

Distributed memory programming with MPI – MPI functions, The trapezoidal rule in MPI, Dealing with I/O, Collective communication, MPI-derived datatypes, Performance evaluation of MPI programs, A parallel sorting algorithm.

3 Module

Module -3 Lecturer Notes: Distributed memory programming with MPI

This Lecture Notes provides a comprehensive overview of Distributed memory programming with MPI, based on Chapter 3 of "Peter S Pacheco, Matthew Malensek – An Introduction to Parallel Programming, second edition, Morgan Kauffman , VTU syllabus based BCS702 Parallel Computing" It is designed to serve as a detailed lecture presentation for a classroom setting, covering the Distributed memory programming with MPI programming basics, and practical examples, while ensuring clarity for students new to parallel computing.

MODULE-3: Distributed memory programming with MPI –

1. MPI functions,
2. The trapezoidal rule in MPI,
3. Dealing with I/O,
4. Collective communication,
5. MPI-derived datatypes,
6. Performance evaluation of MPI programs,
7. A parallel sorting algorithm.

1. MPI functions,

MPI (Message Passing Interface) functions enable parallel computing by facilitating communication between processes. Key MPI functions include:

1. MPI_Init and MPI_Finalize: Initialize and finalize MPI environment.
2. MPI_Comm_rank and MPI_Comm_size: Determine process rank and total number of processes.
3. MPI_Send and MPI_Recv: Send and receive data between processes.
4. MPI_Bcast and MPI_Scatter: Broadcast and scatter data from one process to others.
5. MPI_Gather and MPI_Reduce: Gather and reduce data from multiple processes.

These functions allow for:

1. Point-to-point communication: Direct communication between two processes.
2. Collective communication: Coordinated communication among multiple processes.

MPI functions are essential for writing scalable and efficient parallel programs.

1 **Explain the roles of MPI_Init, MPI_Comm_size, MPI_Comm_rank, and MPI_Finalize in an MPI program, using the greetings program (Program) as an example.** **Answer**: In the greetings program (Program),

```
#include <stdio.h>
#include <string.h>     /* For strlen */
#include <mpi.h>        /* For MPI functions, etc. */
```

```
const int MAX_STRING = 100;

int     main(void)     {                         char
greeting[MAX_STRING];     int comm_sz;
/* Number of processes */     int my_rank;
/* My process rank */

   MPI_Init(NULL, NULL);
   MPI_Comm_size(MPI_COMM_WORLD, &comm_sz);
   MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);

   if (my_rank != 0) {
      sprintf(greeting, "Greetings from process %d of %d!", my_rank, comm_sz);
MPI_Send(greeting, strlen(greeting)+1, MPI_CHAR, 0, 0, MPI_COMM_WORLD);
   } else {        printf("Greetings from process %d of %d!\n", my_rank,
comm_sz);        for (int q = 1; q < comm_sz; q++) {
         MPI_Recv(greeting, MAX_STRING, MPI_CHAR, q, 0, MPI_COMM_WORLD,
MPI_STATUS_IGNORE);
         printf("%s\n", greeting);
      }
   }

   MPI_Finalize();
return 0;
}  /* main */
```

MPI_Init, MPI_Comm_size, MPI_Comm_rank, and MPI_Finalize are essential for setting up and managing the MPI environment. MPI_Init is called first to initialize the MPI system, performing tasks like allocating message buffers and assigning ranks to processes. It takes pointers to argc and argv (or NULL if unused) to allow command-line argument processing.

In Program, it sets up the environment for processes to communicate. MPI_Comm_size retrieves the total number of processes in MPI_COMM_WORLD, the default communicator including all processes, storing it in comm_sz. This allows the program to know how many processes are running (e.g., 4 for four processes). MPI_Comm_rank assigns a unique rank (0 to comm_sz-1) to each process, stored in my_rank, enabling processes to identify themselves and execute rank-specific tasks. In Program , process 0 prints its greeting and receives messages, while others send messages based on their rank. MPI_Finalize is called last to clean up MPI resources, freeing allocated storage and ensuring proper termination. No MPI functions should be called after this. Together, these functions establish the framework for the SPMD (Single Program, Multiple Data) approach in Program, allowing processes to coordinate and communicate effectively to print greetings in a distributed memory system.

2 **Describe the functionality and syntax of MPI_Send and MPI_Recv, and explain how they are used in the greetings program (Above Program) to facilitate communication. Answer**: MPI_Send and MPI_Recv are point-to-point communication functions in MPI, enabling message passing between processes. MPI_Send sends a message to a specified destination process, with syntax: int MPI_Send(void* msg_buf_p, int msg_size, MPI_Datatype msg_type, int dest, int tag, MPI_Comm communicator). The arguments specify the message buffer (msg_buf_p), size (msg_size), data type (msg_type, e.g., MPI_CHAR), destination rank (dest), message tag (tag), and communicator (e.g., MPI_COMM_WORLD). MPI_Recv receives a message, with syntax: int MPI_Recv(void* msg_buf_p, int buf_size, MPI_Datatype buf_type, int source, int tag, MPI_Comm communicator, MPI_Status* status_p). It specifies the receive buffer, buffer size, data type, source rank, tag, communicator, and a status object (or MPI_STATUS_IGNORE). In Program, processes 1 to comm_sz-1 use MPI_Send to send a greeting string (strlen(greeting)+1 chars, type MPI_CHAR) to process 0 with tag 0 in MPI_COMM_WORLD. Process 0 uses MPI_Recv in a loop to receive messages from processes 1 to comm_sz-1, storing them in greeting (size MAX_STRING, type MPI_CHAR) with matching tag 0 and communicator. MPI_Recv blocks until a message is received, ensuring process 0 prints each greeting after receipt. The message matching rules ensure correct communication: the communicator, tag, source (q), and destination (0) must align, and the receive buffer must be compatible (e.g., MAX_STRING ≥ message size). This enables the greetings program to coordinate output, with process 0 collecting and printing messages in rank order, demonstrating effective use of point-to-point communication.

3 **Analyze the use of MPI_Reduce in the context of the trapezoidal rule program and explain how it improves performance compared to the original global sum approach. Answer**: In the trapezoidal rule program (Program), the original global sum approach has processes 1 to comm_sz-1 send their local_int values to process 0 using MPI_Send, and process 0 receives them with MPI_Recv in a loop, summing them into total_int .

```
int main(void) {     int my_rank, comm_sz, n
= 1024, local_n;     double a = 0.0, b = 3.0, h,
local_a, local_b;    double local_int, total_int;
int source;
     MPI_Init(NULL, NULL);
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
    MPI_Comm_size(MPI_COMM_WORLD, &comm_sz);

    h = (b - a) / n;           /* h is the same for all processes */    local_n
= n / comm_sz;       /* So is the number of trapezoids */
     local_a = a + my_rank * local_n * h;     local_b = local_a + local_n * h;     local_int
= Trap(local_a, local_b, local_n, h);
     if (my_rank != 0) {
        MPI_Send(&local_int, 1, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD);
    } else {
        total_int = local_int;
        for    (source    =    1;    source    <    comm_sz;    source++)    {
MPI_Recv(&local_int,        1,        MPI_DOUBLE,        source,        0,
MPI_COMM_WORLD, MPI_STATUS_IGNORE);              total_int +=
local_int;
        }
    }
     if (my_rank == 0) {            printf("With n = %d trapezoids, our estimate\n", n);
printf("of the integral from %f to %f = %.15e\n", a, b, total_int);
    }
     MPI_Finalize();    return 0;
} /* main */
```

This is inefficient, as process 0 performs all the addition (comm_sz-1 receives and adds), while other processes do minimal work, leading to a bottleneck. Section 3.4 introduces MPI_Reduce as a collective communication function to optimize this. Its syntax is: int MPI_Reduce(void* input_data_p, void* output_data_p, int count, MPI_Datatype datatype, MPI_Op operator, int dest_process, MPI_Comm comm). In the trapezoidal rule context, MPI_Reduce would replace the send-receive loop with a call like MPI_Reduce(&local_int, &total_int, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD), summing all local_int values across processes and storing the result in total_int on process 0. This improves performance by distributing the reduction work across processes using a tree-structured approach , where processes pair up to sum values concurrently (e.g., processes 0 and 1, 2 and 3 sum in parallel). For comm_sz=8, the original approach requires 7 receives and adds by process 0, while MPI_Reduce reduces this to 3, with other processes performing up to 2 receives and adds, cutting the time by over 50%. For larger comm_sz (e.g., 1024), it reduces operations from 1023 to 10, improving efficiency by over 100 times. The MPI implementation optimizes this tree structure, relieving programmers from coding it manually, ensuring scalability and performance.

4. **Discuss the safety issues in MPI programs related to MPI_Send and MPI_Recv, and explain how MPI_Ssend and MPI_Sendrecv address these issues in the context of the parallel odd-even transposition sort .**

Answer: safety issues in MPI programs using MPI_Send and MPI_Recv, particularly in the parallel odd-even transposition sort. An unsafe program relies on MPI_Send buffering messages, which may not occur for large messages, leading to deadlocks. In the sort, each process sends its keys to a partner and receives keys using MPI_Send and MPI_Recv . If both processes call MPI_Send first and block (as MPI_Send may do for large messages), neither reaches MPI_Recv, causing a deadlock. To check safety, MPI_Ssend (syntax: int MPI_Ssend(void* msg_buf_p, int msg_size, MPI_Datatype msg_type, int dest, int tag, MPI_Comm communicator)) can replace MPI_Send. MPI_Ssend always blocks until the matching receive starts, revealing deadlocks if the program hangs, confirming unsafety . To make the program safe, suggests restructuring communication so some processes receive first, but recommends MPI_Sendrecv for simplicity. Its syntax is: int MPI_Sendrecv(void* send_buf_p, int send_buf_size, MPI_Datatype send_buf_type, int dest, int send_tag, void* recv_buf_p, int recv_buf_size, MPI_Datatype recv_buf_type, int source, int recv_tag, MPI_Comm communicator, MPI_Status* status_p). In the sort, a single call like MPI_Sendrecv(my_keys, n/comm_sz, MPI_INT, partner, 0, recv_keys, n/comm_sz, MPI_INT, partner, 0, comm, MPI_STATUS_IGNORE) (Section 3.7.4) handles both send and receive, with the MPI implementation scheduling them to avoid deadlocks. This replaces complex manual scheduling (e.g., even-ranked processes send first, odd-ranked receive first,), ensuring safety regardless of message size or comm_sz (even or odd). MPI_Sendrecv simplifies the code, enhances reliability, and maintains performance, making it ideal for the sort's partner exchanges.

2. The trapezoidal rule in MPI,

The Trapezoidal Rule in MPI is a numerical integration technique used to approximate the definite integral of a function. In a parallel computing context using MPI, the trapezoidal rule can be parallelized by:

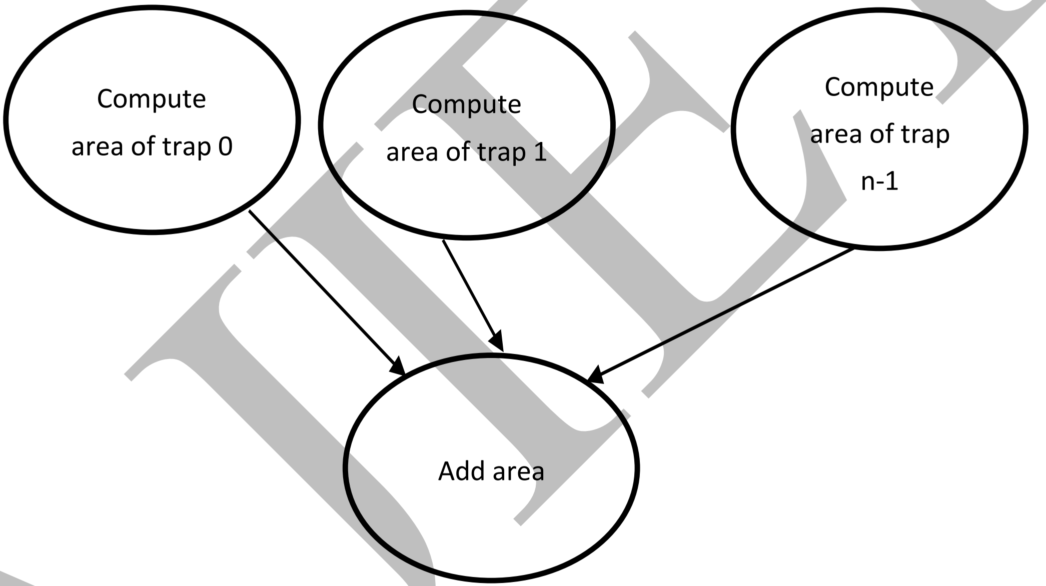
1. Dividing the interval: Splitting the integration interval into smaller sub-intervals.
2. Assigning sub-intervals: Distributing sub-intervals among MPI processes.
3. Calculating partial sums: Each process calculates the integral approximation for its subinterval.

4. Combining results: Gathering and summing partial sums from all processes.

This parallel approach enables efficient computation of definite integrals, leveraging the power of multiple processes in MPI.

1. **Explain the process of parallelizing the trapezoidal rule using Foster's methodology.**

Answer: Foster's methodology is applied to parallelize the trapezoidal rule in four steps: partitioning, communication, aggregation, and mapping. In the **partitioning** phase, tasks are identified as computing the area of individual trapezoids and summing these areas. Each trapezoid's area calculation is a distinct task, and summing them is another task. In the **communication** phase, channels are defined where each trapezoid area task sends its result to the summing task, as shown in Figure 3.5.



[Figure : Task and communications for the trapezoidal rule Refer: "Peter S Pacheco, Matthew Malensek – An Introduction to Parallel Programming, second edition, Morgan Kauffman]

For **aggregation**, since the number of trapezoids (n) is typically much larger than the number of processes (comm_sz), the trapezoid tasks are grouped by dividing the interval [a, b] into comm_sz subintervals, each containing n / comm_sz trapezoids (assuming comm_sz divides n). Each process computes the trapezoidal rule for its subinterval. In the **mapping** phase, these composite tasks are assigned to processes: each of the comm_sz processes computes the integral over its subinterval, and process 0 aggregates the results by summing local integrals. This is implemented in Above Program, where each process calculates local_int using the Trap function, non-zero processes send their local_int to process 0 via MPI_Send, and process 0 receives these using MPI_Recv and sums them into total_int, ensuring efficient parallel computation.

2. **Describe the structure and functionality of Above Program, focusing on how it implements the parallel trapezoidal rule.**

Answer: In Above Program is an MPI program that implements the parallel trapezoidal rule to approximate the integral of a function over [a, b] using n trapezoids across comm_sz processes. It begins with initialization: MPI_Init sets up the MPI environment, MPI_Comm_rank assigns each process a rank (my_rank), and MPI_Comm_size determines the number of processes (comm_sz). The program hardwires input values (a = 0.0, b = 3.0, n = 1024). Each process computes the trapezoid width h = (b - a) / n and its local number of trapezoids local_n = n / comm_sz. The local subinterval for each process is defined by local_a = a + my_rank * local_n * h and local_b = local_a + local_n * h. Each process calls the Trap function (Program) to compute its local integral local_int over [local_a, local_b] using local_n trapezoids.

```
double   Trap(            double
left_endpt,   /* in */     double
right_endpt,   /*  in  */      int
trap_count,     /* in */   double
base_len      /* in */
) {    double estimate,
x;
    int i;
     estimate = (f(left_endpt) + f(right_endpt)) / 2.0;      for (i = 1; i <= trap_count - 1;
i++) {         x = left_endpt + i * base_len;        estimate += f(x);
    }
    estimate = estimate * base_len;
     return estimate;
} /* Trap */
```

Non-zero processes (rank ≠ 0) send their local_int to process 0 using MPI_Send with one MPI_DOUBLE, destination rank 0, and tag 0. Process 0 initializes total_int with its local_int, then uses a loop to receive local_int from each process (1 to comm_sz-1) via MPI_Recv, adding each to total_int. Finally, process 0 prints the result, formatting it to show the integral estimate. MPI_Finalize cleans up the MPI environment. The program assumes comm_sz divides n, ensuring equal work distribution, and uses SPMD (Single Program, Multiple Data) by branching on my_rank.

3.Discuss how local and global variables are used in the MPI trapezoidal rule program and their significance in distributed memory programming.

Answer: In Above Program , local and global variables are distinguished to manage data in a distributed memory system where each process has its own memory. **Local variables** are specific to each process and include local_a, local_b, local_n, and local_int. local_a and local_b define the endpoints of a process's subinterval, calculated as local_a = a + my_rank * local_n * h and local_b = local_a + local_n * h, unique to each process based on its rank (my_rank). local_n = n / comm_sz is the number of trapezoids per process, and local_int is the integral computed over the subinterval, both varying by process. These variables are critical as they allow each process to work independently on its portion of the problem without accessing other processes' memory. **Global variables** have the same value across all processes and include a, b, n, and h. a and b are the interval endpoints (0.0 and 3.0), n is the total number of trapezoids (1024), and h = (b - a) / n is the trapezoid width. These ensure consistent problem parameters across processes, facilitating coordinated computation. In distributed memory programming, this distinction is vital: global variables define the shared problem context, while local variables enable parallel execution by partitioning data, avoiding the need for shared memory access and reducing communication overhead, as seen in the program's design.

4.Explain the communication structure in Above Program and how it ensures the correct computation of the total integral.

Answer: The communication structure in Above Program is designed to collect local integral contributions from all processes to compute the total integral, using point-to-point MPI functions. After each process computes its local_int using the Trap function, processes with my_rank ≠ 0 (1 to comm_sz-1) send their local_int to process 0 using MPI_Send(&local_int, 1, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD). This call sends one MPI_DOUBLE value to destination rank 0 with tag 0 in MPI_COMM_WORLD. Process 0, meanwhile, starts with total_int = local_int (its own contribution), then enters a for loop to receive contributions from each process source = 1 to comm_sz-1 using MPI_Recv(&local_int, 1, MPI_DOUBLE, source, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE). This receives one

MPI_DOUBLE from the specified source with tag 0, storing it in local_int, which is then added to total_int. The loop ensures all contributions are collected in rank order, guaranteeing each process's local_int is included exactly once. Message matching rules ensure correctness: the communicator (MPI_COMM_WORLD), tag (0), destination (0), and source ranks align, and the data type (MPI_DOUBLE) and size (1) are compatible. MPI_Recv blocks until a matching send occurs, preventing data loss, and the sequential receive order avoids deadlocks. This structure ensures total_int is the sum of all local_int values, accurately approximating the integral over [a, b], with process 0 handling output.

3. Dealing with I/O,

Dealing with I/O in MPI requires careful consideration due to parallel execution. Key challenges and approaches:

Challenges:

1. Concurrent access: Multiple processes accessing shared files.
2. Data consistency: Ensuring data integrity and consistency.

Approaches:

1. Independent I/O: Each process performs I/O operations independently.
2. Collective I/O: Coordinated I/O operations among multiple processes.
3. Parallel file systems: Utilizing file systems optimized for parallel I/O.

Best practices:

1. Minimize I/O overhead: Reduce I/O operations.
2. Use collective I/O: Optimize performance with coordinated I/O.
3. Ensure data consistency: Implement synchronization mechanisms.

By addressing I/O challenges, MPI applications can achieve better performance and scalability.

1. **What function is used in Program to read input in the trapezoidal rule program?**

Answer: The Get_input function is used to read input for a, b, and n.

```
void  Get_input(           int
my_rank,   /* in   */      int
comm_sz,       /*   in      */
double*  a_p,       /*  out  */
double*  b_p,       /*  out  */
int*     n_p      /* out */
) {
   int   dest;             if   (my_rank    ==   0)   {
printf("Enter a, b, and n\n");        scanf("%lf
%lf %d", a_p, b_p, n_p);          for (dest = 1;
dest < comm_sz; dest++) {
         MPI_Send(a_p, 1, MPI_DOUBLE, dest, 0, MPI_COMM_WORLD);
         MPI_Send(b_p, 1, MPI_DOUBLE, dest, 0, MPI_COMM_WORLD);
         MPI_Send(n_p, 1, MPI_INT, dest, 0, MPI_COMM_WORLD);
      }
   } else {  /* my_rank != 0 */
```

```
            MPI_Recv(a_p, 1, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD,
        MPI_STATUS_IGNORE);
            MPI_Recv(b_p, 1, MPI_DOUBLE, 0, 0,
              MPI_COMM_WORLD, MPI_STATUS_IGNORE);
            MPI_Recv(n_p, 1, MPI_INT, 0, 0, MPI_COMM_WORLD,
        MPI_STATUS_IGNORE);
        }
      } /* Get_input */
```

2. **Explain the issue of nondeterminism in MPI program output and how it can be addressed,**

Answer: Nondeterminism in MPI program output arises because multiple processes can write to stdout simultaneously, and most MPI implementations do not schedule access to this shared output device. This competition results in unpredictable output order across different runs, as illustrated in Program Above, where a simple program with each process printing a message (e.g., "Does anyone have a toothpick?") produces varying output sequences when run with six processes. For example, one run might show process 5's output before process 3's, while another run shows a different order. This variability occurs because the processes compete for stdout, and the operating system or MPI implementation does not enforce a specific order, leading to nondeterministic behavior. To address this, the document suggests modifying the program to ensure orderly output. A common approach, as used in the "greetings" program, is to have each process other than 0 send its output to process 0, which then prints all messages in process rank order. This ensures a deterministic output sequence, as process 0 controls the printing, eliminating the competition for stdout and producing consistent results across runs.

3. **Describe the structure and functionality of the Get_input function in Above Program, and explain how it handles input distribution in the parallel trapezoidal rule program.**

Answer: The Get_input function in Above Program is designed to read input parameters (a, b, and n) for the parallel trapezoidal rule program and distribute them to all processes in MPI_COMM_WORLD. It takes five parameters: my_rank (the process rank), comm_sz (the number of processes), and pointers a_p, b_p, and n_p to store the input values. The function uses an SPMD approach, branching based on my_rank. If my_rank == 0, process 0 prompts the user with printf("Enter a, b, and n\n") and reads the input using scanf("%lf %lf %d", a_p, b_p, n_p). It then iterates over processes 1 to comm_sz-1 using a for loop, sending each input value to each process with three separate MPI_Send calls: one for a_p (type MPI_DOUBLE), one for b_p (type MPI_DOUBLE), and one for n_p (type MPI_INT), all with destination rank dest, tag 0, and communicator MPI_COMM_WORLD. For processes with my_rank != 0, the function executes three MPI_Recv calls to receive a_p, b_p, and n_p from process 0 (source rank 0, tag 0, communicator MPI_COMM_WORLD), using MPI_STATUS_IGNORE for the status argument. This structure ensures that only process 0 accesses stdin, avoiding conflicts, and all processes receive identical copies of the input values, enabling them to compute their local subintervals for the trapezoidal rule. The communication pattern mirrors the "greetings" program but reverses the direction, with process 0 sending data instead of receiving it.

4. **Discuss why most MPI implementations restrict stdin access to process 0 and how this impacts the design of input handling in MPI programs.**

   Answer: Most MPI implementations restrict stdin access to process 0 in MPI_COMM_WORLD to avoid ambiguity and potential errors when multiple processes attempt to read from the same input stream. If multiple processes could access stdin, it would be unclear how the input data should be divided among them. For instance, questions arise such as whether process 0 should read the first line and process 1 the second, or whether process 0 should read the first character, leading to chaotic and unpredictable behavior. This restriction simplifies input handling by designating a single process as the input reader, ensuring consistent and controlled access to stdin. This design choice impacts MPI program design by requiring programmers to structure input handling around process 0. As shown in Above Program 's Get_input function, process 0 reads the input (e.g., a, b, and n for the trapezoidal rule) and then distributes it to other processes using MPI_Send. Non-zero ranked processes receive this data via MPI_Recv, ensuring all processes have the necessary input to perform their computations. This approach necessitates an SPMD structure with rank-based branching, where process 0 handles stdin access and communication, while others wait to receive data. This design avoids conflicts, maintains determinism in input handling, and aligns with the distributed memory model, where each process operates independently and communicates explicitly via message passing.

5. **Compare the communication structures used for handling output in the "greetings" program and input in the Get_input function, highlighting their similarities and differences.**

   Answer: The "greetings" program and the Get_input function in Above Program both employ communication structures involving process 0 and other processes in MPI_COMM_WORLD, but they differ in direction and purpose. In the "greetings" program, processes 1 to comm_sz1 generate messages (e.g., "Greetings from process %d of %d!") and send them to process 0 using MPI_Send. Process 0 receives these messages in rank order (from process 1 to comm_sz1) using MPI_Recv in a for loop, printing its own message first, followed by each received message. This structure ensures orderly output to stdout, avoiding nondeterministic output from multiple processes writing directly. Conversely, in the Get_input function , process 0 reads input (a, b, n) from stdin using scanf and sends these values to processes 1 to comm_sz1 using MPI_Send in a for loop, with each process receiving one MPI_DOUBLE for a, one for b, and one MPI_INT for n. Non-zero ranked processes use

MPI_Recv to receive these values from process 0. **Similarities** include the use of process 0 as the central coordinator, point-topoint communication (MPI_Send and MPI_Recv), and a loop to handle communication with each non-zero process in rank order, ensuring structured data exchange. **Differences** lie in the direction of communication (non-zero to 0 for output in "greetings" vs. 0 to non-zero for input in Get_input), the data type (MPI_CHAR strings vs. MPI_DOUBLE and MPI_INT), and the purpose (collecting output for printing vs. distributing input for computation). Both approaches leverage process 0's role to manage I/O, reflecting MPI's design for controlled access to shared resources like stdout and stdin.

4. Collective communication,

Collective communication in MPI involves coordinated communication among multiple processes in a group. Key features:

1. Broadcast: One process sends data to all others.
2. Scatter: One process sends different data to each process.
3. Gather: Multiple processes send data to one process.
4. Reduce: Multiple processes contribute data, and one process receives the result of a reduction operation.

Collective communication:

1. Simplifies code: Reduces need for explicit point-to-point communication.
2. Improves performance: Optimized for efficiency and scalability.

Common collective communication functions in MPI include:

- MPI_Bcast
- MPI_Scatter
- MPI_Gather
- MPI_Reduce
- MPI_Allreduce
- MPI_Allgather

By leveraging collective communication, MPI applications can achieve better performance, scalability, and code readability.

1. **Explain the inefficiency in the global sum operation of the trapezoidal rule program and how a tree-structured communication can improve it,.**

Answer: In the trapezoidal rule program, the global sum operation is inefficient because each process with rank greater than 0 sends its local integral to process 0, which then performs all additions, while other processes do minimal work. This is analogous to seven workers instructing one worker to perform all additions, leading to process 0 executing comm_sz - 1 receives and additions. proposes a tree-structured communication, illustrated with a binary tree for eight processes , to address this. Initially, processes 1, 3, 5, and 7 send their values to processes 0, 2, 4, and 6, respectively, which add these to their own values. In the second phase, processes 2 and 6 send their sums to processes 0 and 4, which add them. Finally, process 4 sends its sum to process 0, which computes the final sum. This reduces process 0's work to three receives and three additions, compared to seven in the original scheme. Additionally, many operations (e.g., receives by processes 0, 2, 4, 6 in the first phase) occur concurrently, reducing the total time by over 50%. For larger comm_sz, such as 1024, the tree structure requires only 10 receives and additions for process 0, improving performance by a factor of over 100 compared to 1023 in the original scheme. This approach distributes work more equitably and leverages concurrency, significantly enhancing efficiency.

2. **Describe the functionality and syntax of the MPI_Reduce function, including how it generalizes operations like global sum.**

Answer: The MPI_Reduce function, , is a collective communication function that performs a global reduction operation on data distributed across all processes in a communicator, storing the result on a designated process. It generalizes operations like global sum by allowing various reduction operators, such as maximum, minimum, or product, through its operator argument.

The syntax is:

```
int MPI_Reduce(
   void*  input_data_p,        /*  in  */
void* output_data_p,   /* out */     int
count,            /* in */
   MPI_Datatype datatype, /* in */
MPI_Op operator,       /* in */     int
dest_process,      /* in */
   MPI_Comm comm         /* in */
);
```

The input_data_p points to the local data each process contributes, while output_data_p points to where the result is stored on the dest_process. The count specifies the number of elements, and datatype (e.g., MPI_DOUBLE, MPI_INT) defines their type. The operator argument, of type MPI_Op, selects the reduction operation, with predefined values like MPI_SUM for summation, MPI_MAX for maximum, or MPI_MIN for minimum. For example, to compute a global sum in the trapezoidal rule program, MPI_SUM is used. The dest_process specifies the rank of the process receiving the result, and comm is the communicator (e.g., MPI_COMM_WORLD). This generalization allows MPI_Reduce to handle diverse operations, and its implementation is optimized by MPI developers, leveraging their knowledge of hardware and system software to ensure efficient performance across different systems, relieving application developers from coding complex tree-structured communications.

3 Compare and contrast the MPI_Scatter and MPI_Gather functions, including their roles in data distribution and collection.

Answer: The MPI_Scatter and MPI_Gather functions, are collective communication functions that handle data distribution and collection, respectively, and are described as inverses of each other. **MPI_Scatter** distributes elements of an array from one process (typically rank 0) to all processes in a communicator. If the array on the sending process has n elements and there are p processes, each process receives n/p elements. The first n/p elements go to process 0, the next n/p to process 1, and so on. This is useful for partitioning data, such as dividing a vector for parallel processing. For example, in a program where process 0 reads a vector, MPI_Scatter can distribute its components across all processes for computation. Conversely, **MPI_Gather** collects elements from all processes into a single array on a designated process (typically rank 0). If each process has m elements, MPI_Gather assembles them into an array of m*p elements on the designated process, with elements from process 0 first, followed by process 1, and so on. This is useful for aggregating results, such as collecting local computation outputs into a global result. **Similarities** include their collective nature, involving all processes in the communicator, and their use in managing distributed data. **Differences** lie in their direction (MPI_Scatter disperses data, MPI_Gather collects it), their primary use cases (distribution vs. aggregation), and the process roles (one sender in MPI_Scatter vs. one receiver in MPI_Gather). Together, they enable efficient data management in parallel programs, with MPI_Scatter initiating parallel work and MPI_Gather consolidating results.

3.Discuss the role and implementation of MPI_Barrier in MPI programs, and explain how it is used for synchronization.

Answer: The MPI_Barrier function, is a collective communication function used to synchronize processes within a communicator, ensuring that no process proceeds until all processes in the communicator have reached the barrier. Its primary role is to coordinate processes, particularly for timing or ensuring consistent program states before proceeding with further computations or communications. The syntax is simple:

```
int MPI_Barrier(MPI_Comm comm /* in */);
```

The sole argument, comm, specifies the communicator (e.g., MPI_COMM_WORLD). No process returns from MPI_Barrier until every process in the communicator has called it, effectively creating a synchronization point. An example application is timing a block of MPI code, as shown in the document. To measure elapsed time, processes call MPI_Barrier before starting the timer (MPI_Wtime), execute the code, record the finish time, and compute local elapsed time. A subsequent MPI_Reduce with MPI_MAX finds the maximum elapsed time across processes, reported by process 0. This ensures all processes start the timed section simultaneously, providing a consistent measurement. MPI_Barrier is critical in scenarios requiring synchronized execution, such as ensuring all processes have completed a phase of computation before proceeding to a collective operation like MPI_Reduce or MPI_Allgather. Its implementation by the MPI library ensures efficient synchronization, leveraging systemspecific optimizations, and it avoids the need for programmers to manually coordinate process execution, simplifying the development of robust parallel programs.

5. MPI-derived datatypes,

MPI-derived datatypes enable the creation of custom data types for complex data structures, facilitating efficient communication in parallel applications. Key features:

1. Non-contiguous data: Handle non-contiguous memory layouts.
2. Complex data structures: Support for structures, arrays, and nested types.
3. Type construction: Functions like MPI_Type_contiguous, MPI_Type_vector, and MPI_Type_struct create custom types.

Benefits:

1. Efficient data transfer: Reduced overhead and improved performance.
2. Flexibility: Handle complex data structures and non-contiguous memory.

Common use cases:

1. Array sections
2. Structures
3. Nested data structures

By using MPI-derived datatypes, developers can optimize data transfer and improve the performance of parallel applications.

1. **Explain how the MPI_Type_create_struct function is used to create a derived datatype for a C struct, including the steps and arguments .**

Answer: the MPI_Type_create_struct function is described as a method to build a derived datatype that represents a C struct, enabling efficient communication of complex data structures in MPI programs. The function allows programmers to specify the layout of a struct by defining the number of elements, their displacements in memory, and their datatypes. The syntax is:

```
int MPI_Type_create_struct(
    int count,          /* in */    int
block_lengths[],        /* in */
```

```
                    MPI_Aint displacements[],     /* in */
                    MPI_Datatype datatypes[],     /* in */
                    MPI_Datatype* new_type_p      /* out */
                  );
```

The count argument specifies the number of elements in the struct. The block_lengths array indicates how many elements of each datatype are included (typically 1 for each struct member). The displacements array provides the byte offsets of each element relative to the start of the struct, using the MPI_Aint type for address arithmetic. The datatypes array lists the MPI datatypes (e.g., MPI_INT, MPI_DOUBLE) for each element. The new_type_p argument returns the newly created derived datatype.

To create the datatype, the programmer first defines arrays for block_lengths, displacements, and datatypes. For example, for a struct with an int, a double, and another int, count would be 3, block_lengths might be {1, 1, 1}, and datatypes could be {MPI_INT, MPI_DOUBLE, MPI_INT}. The displacements are computed using MPI_Get_address to get the memory
addresses of each struct member and calculate their offsets from the struct's base address. After calling MPI_Type_create_struct, the new datatype must be committed using MPI_Type_commit to make it usable for communication functions like MPI_Send or MPI_Recv. Finally, when the datatype is no longer needed, MPI_Type_free is called to deallocate its resources. This process ensures that complex data structures can be sent as a single message, improving communication efficiency by reducing the number of messages.

2. **Discuss the application of a derived datatype in the matrix-vector multiplication program, and explain how it improves communication efficiency.**

Answer: the matrix-vector multiplication program uses a derived datatype to optimize the communication, specifically for sending a row of the matrix from one process to another. In the program, the matrix is distributed among processes, and each process needs to access rows of the matrix stored on other processes to complete the multiplication. Initially, the program could send each element of a matrix row individually, but this approach is inefficient due to the high cost of sending multiple messages, as sending a given volume of data in fewer messages is generally less expensive.

To address this, the program uses MPI_Type_create_struct to define a derived datatype representing a row of the matrix. For a matrix with n columns, a row consists of n elements of type double. However, the document simplifies this by noting that a matrix row is stored contiguously in memory, so a derived datatype like MPI_Type_contiguous could be used, but for illustrative purposes, MPI_Type_create_struct is employed. The derived datatype is constructed by specifying count as 1 (since it's one block of data), block_lengths as {n}, displacements as {0} (since the row starts at the base address), and datatypes as {MPI_DOUBLE}. After creating the datatype, MPI_Type_commit is called to make it usable.

This derived datatype allows a process to send an entire matrix row as a single message using MPI_Send, rather than sending n separate messages for each element. For example, in the program, process 0 might send a row to process 1 using the derived datatype, reducing communication overhead. The receiving process uses MPI_Recv with the same derived datatype to correctly interpret the incoming data. This approach significantly improves communication efficiency by minimizing the number of messages, which is critical in distributed-memory systems where communication is expensive. The document emphasizes that this optimization leverages the MPI implementation's ability to handle complex data layouts, relieving programmers from manually packing data into buffers.

3. **Compare and contrast the use of MPI-derived datatypes with MPI_Pack and MPI_Unpack for combining data into a single message.**

Answer: The two methods for combining multiple data items into a single message in MPI programs: using MPI-derived datatypes and using MPI_Pack and MPI_Unpack. Both approaches aim to reduce communication overhead by sending data in fewer messages, but they differ in their mechanisms, flexibility, and use cases.

MPI-Derived Datatypes: Derived datatypes, created using functions like MPI_Type_create_struct, describe arbitrary collections of data by specifying their types and relative positions in memory. For example, in the matrix-vector multiplication program, a derived datatype represents a matrix row, allowing it to be sent as a single message. The process involves defining the datatype with count, block_lengths, displacements, and datatypes, committing it with MPI_Type_commit, and using it in communication functions like MPI_Send or MPI_Recv. The key advantage is that once created, the datatype can be reused across multiple communications, providing a structured and efficient way to handle complex data layouts. Derived datatypes are particularly useful for repetitive communications of the same data structure, as they encapsulate the data's memory layout, reducing programmer effort in managing buffers. However, creating and committing a derived datatype incurs initial overhead, and it requires careful calculation of displacements, which can be complex for non-contiguous or heterogeneous data.

MPI_Pack and MPI_Unpack: These functions provide an alternative by manually packing data into a contiguous buffer before sending and unpacking it after receiving. MPI_Pack copies data items into a user-provided buffer, specifying their types and counts, while MPI_Unpack extracts them in the same order. This approach is more flexible for one-time or irregular communications, as it doesn't require defining a reusable datatype. For example, a process could pack an int, a double, and an array into a buffer, send it, and the receiver could unpack them sequentially. However, this method requires explicit buffer management, increasing the risk of errors (e.g., buffer overflows or incorrect unpacking order). It also incurs runtime overhead for packing and unpacking each time data is sent or received, making it less efficient for repetitive communications compared to derived datatypes.

Comparison: Derived datatypes are more efficient for repeated communications of fixed data structures, as they define the layout once and reuse it, while MPI_Pack and MPI_Unpack are better suited for ad-hoc or variable data structures due to their flexibility. Derived datatypes reduce programmer effort in managing data layouts but require upfront setup, whereas MPI_Pack and MPI_Unpack are simpler to implement for one-off communications but demand careful buffer handling. Both methods achieve the goal of reducing message counts, but derived datatypes offer better performance for structured, repetitive tasks, as seen in the matrix-vector multiplication example, while MPI_Pack and MPI_Unpack provide a quick solution for less regular data.

4. **Describe the process of creating and using a derived datatype in an MPI program, including error handling and resource management.**

Answer: the process of creating and using an MPI-derived datatype to facilitate efficient communication of complex data structures, such as a matrix row in the matrix-vector multiplication program. The process involves several steps, including creation, commitment, usage, and cleanup, with considerations for error handling and resource management. Below is a detailed description:

Step 1: Define the Data Structure: Identify the data to be communicated, such as a C struct or a matrix row. For example, a matrix row of n doubles is contiguous in memory, but a struct might have multiple elements (e.g., an int, a double, an int) with specific offsets.

Step 2: Create the Derived Datatype: Use MPI_Type_create_struct to define the datatype. The function requires:

- count: The number of elements (e.g., 3 for a struct with three members). ○ block_lengths: An array specifying the number of elements per datatype (e.g., {1, 1, 1}).
- displacements: An array of MPI_Aint values, calculated using MPI_Get_address to determine the byte offsets of each element relative to the struct's base address.
- datatypes: An array of MPI datatypes (e.g., {MPI_INT, MPI_DOUBLE, MPI_INT}).
- new_type_p: A pointer to the new derived datatype.

For a matrix row, a simpler approach might use MPI_Type_contiguous, but MPI_Type_create_struct is used for generality. The call might look like:

```
MPI_Datatype row_type; int block_lengths[1] = {n}; MPI_Aint
displacements[1] = {0};
```

MPI_Datatype datatypes[1] = {MPI_DOUBLE};

MPI_Type_create_struct(1, block_lengths, displacements, datatypes, &row_type); **Step 3: Commit the Datatype**: Call MPI_Type_commit to register the datatype with the MPI implementation, making it usable for communication: MPI_Type_commit(&row_type);

Step 4: Use the Datatype: Use the committed datatype in communication functions like MPI_Send or MPI_Recv. For example, to send a matrix row:

MPI_Send(matrix_row, 1, row_type, dest, tag, MPI_COMM_WORLD); The

receiving process uses the same datatype:

MPI_Recv(matrix_row, 1, row_type, source, tag,
MPI_COMM_WORLD, MPI_STATUS_IGNORE);

Step 5: Error Handling: Although the document does not emphasize error handling for derived datatypes, it notes that MPI functions return error codes. Programmers should check the return values of MPI_Type_create_struct, MPI_Type_commit, and communication functions to detect errors (e.g., invalid datatypes or memory issues). For simplicity, the example programs ignore error codes, but in production code, one might use:

```
int err = MPI_Type_create_struct(...);
if (err != MPI_SUCCESS) {
   /* Handle error */
}
```

Step 6: Free the Datatype: After communication is complete, deallocate the datatype using MPI_Type_free to release resources:

MPI_Type_free(&row_type);

Resource Management: The document stresses that derived datatypes must be committed before use and freed afterward to avoid resource leaks. Failing to call MPI_Type_free could lead to memory leaks in the MPI implementation. Additionally, MPI_Get_address ensures portable displacement calculations, avoiding alignment issues across systems.

This process, as illustrated in the matrix-vector multiplication example, enables efficient communication by sending complex data as a single message, reducing overhead. It requires careful setup but simplifies repetitive communications, with proper resource management ensuring robust program execution.

6. Performance evaluation of MPI programs,

Performance evaluation of MPI programs assesses the efficiency and scalability of parallel applications. Key aspects:

1. Scalability: How performance changes with increasing number of processes.
2. Efficiency: Ratio of actual to ideal performance.

3. Overhead: Communication, synchronization, and other overheads.

Evaluation metrics:

1. Speedup: Ratio of parallel to serial execution time.
2. Efficiency: Speedup divided by number of processes.
3. Scalability: Ability to handle increased workload.

Tools and techniques:

1. Profiling tools: Measure execution time, communication overhead.
2. Tracing tools: Visualize program execution and communication patterns.

By evaluating performance, developers can:

1. Identify bottlenecks: Optimize critical sections.
2. Improve scalability: Enhance parallel efficiency.
3. Optimize resource utilization: Reduce waste and improve performance.

1. **Describe the process of timing an MPI program to evaluate its performance, including the use of specific MPI functions.**

Answer: a systematic approach to timing MPI programs to evaluate their performance, focusing on measuring elapsed or "wall clock" time. The process involves synchronizing processes, recording start and end times, and determining the maximum elapsed time across all processes to account for the slowest process. The key steps and MPI functions used are as follows: First, all processes in the communicator are synchronized using MPI_Barrier, which ensures that no process proceeds until all have reached the barrier. This is critical for accurate timing, as it aligns the start of the timed code across processes. The syntax is: int MPI_Barrier(MPI_Comm comm);

Next, the start time is recorded using MPI_Wtime, a function that returns the wall clock time as a double. The code to be timed is then executed, followed by another call to MPI_Wtime to record the finish time. The local elapsed time for each process is computed as the difference between finish and start times:

```
double local_start, local_finish, local_elapsed;
MPI_Barrier(comm);        local_start        =
MPI_Wtime();  /*  Code  to  be  timed  */
local_finish = MPI_Wtime(); local_elapsed =
local_finish - local_start;
```

To obtain a single elapsed time representing the program’s performance, the maximum local elapsed time across all processes is computed using MPI_Reduce with the MPI_MAX operator. This ensures the reported time reflects the slowest process, which determines the overall program completion time. The reduction stores the result on process 0, which can then print it: double elapsed;

```
MPI_Reduce(&local_elapsed, &elapsed, 1, MPI_DOUBLE, MPI_MAX, 0, comm);
if (my_rank == 0) {     printf("Elapsed time = %e seconds\n", elapsed); }
```

The variability in timings due to unpredictable system interactions (e.g., operating system scheduling) requires multiple runs, with the minimum run-time typically reported as it best represents performance on a quiet system. Additionally, on hybrid systems with multicore nodes, running one MPI process per node may reduce contention and variability, improving timing accuracy. This structured approach ensures reliable performance measurements for MPI programs.

2. **Analyze the run-time results of the matrix-vector multiplication program, and explain how they reflect the relationship between serial and parallel run-times.**

   Answer: run-time results for the matrix-vector multiplication program in Table 3.5, with times in milliseconds for square matrices of orders 1024, 2048, 4096, 8192, and 16,384, run with 1, 2, 4, 8, and 16 processes. The analysis reveals key insights into the relationship between serial (Tserial (n)) and parallel (Tparallel (n,p)) run-times, as described by the formula:

   Tparallel (n,p)=Tserial (n)/p+Toverhead

   The results show that for a fixed number of processes (p), increasing the matrix order (n) increases run-times. For small p (e.g., p=1,2), doubling n roughly quadruples the run-time, consistent with the serial program's complexity (Tserial (n)≈an2), since matrix-vector multiplication involves 2n2 floating-point operations. For example, with p=1, the run-time increases from 4.1 ms (n=1024) to 16.0 ms (n=2048), a factor of approximately 4. However, for larger p, this quadratic relationship weakens due to the influence of parallel overhead.

   When fixing n and increasing p , run-times generally decrease, especially for large n. For instance, with n=16,384, the run-time drops from 1100 ms (p=1) to 560 ms (p=2), 280 ms (p=4), 140 ms (p=8), and 71 ms (p=16), showing that doubling p roughly halves the run-time, indicating near-linear speedup for large n n n. This suggests that for large n and small p, the term Tserial (n)/p dominates, as the work is evenly divided among processes. However, for small n (e.g., n=1024), increasing p from 8 to 16 yields no improvement (1.7 ms for both), indicating that the overhead term, primarily from MPI_Allgather, becomes significant. The parallel program divides the serial work (n2 operations) by p, performing n2/p operations per process, but incurs overhead from MPI_Allgather, which is negligible for large n and small p but dominant for small n and large p. Specific examples, such as Tserial (8192)≈1.9×Tparallel (8192,2) and Tparallel (8192,2)≈2.0×Tparallel (8192,4), confirm that the parallel run-times align with the serial run-time divided by p when overhead is minimal. For small n and large p, such as Tparallel (1024,8)=Tparallel (1024,16), the overhead limits performance gains, illustrating the trade-off between computation and communication in MPI programs.

3. **Explain the concepts of speedup and efficiency in the context of the matrix-vector multiplication program, and discuss how the results demonstrate these metrics.**

Answer: speedup and efficiency as key metrics for evaluating MPI program performance, using the matrix-vector multiplication program as a case study. Speedup (S(n,p)) is the ratio of the serial run-time (Tserial (n) T_{\text {serial }}(n) Tserial (n)) to the parallel run-time (Tparallel (n,p)):

S(n,p)=Tserial (n)Tparallel (n,p)

The ideal speedup is p , the number of processes, termed linear speedup, where the parallel program runs p times faster than the serial program. Efficiency (E(n,p)) is the speedup per process:

E(n,p)=S(n,p)p=Tserial (n)p×Tparallel (n,p)

E(n,p)=pS(n,p)=p×Tparallel (n,p)Tserial (n)

The ideal efficiency is 1.0, indicating perfect utilization of all processes.

Table: Speedups of parallel matrix–vector multiplication:

comm_sz	**1024**	**2048**	**4096**	**8192**	**16,384**
1	1.0	1.0	1.0	1.0	1.0
2	1.8	1.9	1.9	1.9	2.0
4	2.1	3.1	3.6	3.9	3.9
8	2.4	4.8	6.5	7.5	7.9
16	2.4	6.2	10.8	14.2	15.5

The speedup results in Table show that for small p and large n, the program achieves nearlinear speedup. For example, with n=16,384, speedups are 2.0 (p=2), 3.9 (p=4), 7.9 (p=8), and 15.5 (=16), approaching the ideal values of 2, 4, 8, and 16, respectively. This indicates efficient parallelization when the problem size is large, as the computational work (n2/p) dominates over communication overhead. However, for small n and large p, speedups are far from linear; for n=1024 and p=16, the speedup is only 2.4, well below the ideal 16, due to significant communication overhead from MPI_Allgather.

The efficiency results in Table below mirror this trend. For n=16,384, efficiencies are 0.98 (p=2), 0.98 (p=4), 0.98 (p=8), and 0.97 (p=16 p = 16 p=16), close to the ideal 1.0, showing that each process is effectively utilized. For n=1024, efficiencies drop significantly, e.g., 0.15 (p=16), indicating poor per-process performance due to dominant communication costs.

Table: Efficiencies of parallel matrix–vector multiplication

comm_sz	**1024**	**2048**	**4096**	**8192**	**16,384**
1	1.00	1.00	1.00	1.00	1.00
2	0.89	0.94	0.97	0.96	0.98
4	0.51	0.78	0.89	0.96	0.98
8	0.30	0.61	0.82	0.94	0.98
16	0.15	0.39	0.68	0.89	0.97

The document explains that low efficiencies for small n and large p result from parallel overhead, which increases with p and diminishes the benefits of additional processes when the computational work per process is small.

These results demonstrate that speedup and efficiency are highly dependent on the balance between computation and communication. The matrix-vector multiplication program achieves high speedup and efficiency for large problem sizes and moderate process counts, but performance degrades when communication overhead dominates, highlighting the importance of optimizing MPI programs for specific problem sizes and system configurations.

4. **Discuss the concept of scalability in MPI programs, using the matrix-vector multiplication program as an example.**

Answer: defines scalability as the ability of a parallel program to maintain efficiency as the number of processes (p) increases, provided the problem size (n) can be increased at an appropriate rate. Two types of scalability are introduced: strong scalability, where efficiency remains constant without increasing n, and weak scalability, where efficiency is maintained by increasing n at the same rate as p. The matrix-vector multiplication program is analyzed to assess its scalability based on the efficiency results in Above Table.

Scalability is critical because it indicates whether a program can effectively utilize additional computational resources. The document notes that the matrix-vector multiplication program does not exhibit strong scalability, as efficiency generally decreases when p increases for a fixed n. For example, with n=1024 , efficiency drops from 0.89 (p=2) to 0.15 (p=16), showing that adding processes reduces per-process efficiency due to increased communication overhead, primarily from MPI_Allgather.

However, the program demonstrates weak scalability. When both p and n are doubled, efficiency often increases, especially for larger p . For instance, for p=4 and n=4096, efficiency is 0.89; doubling to p=8 and n=8192, efficiency rises to 0.94. Similarly, from p=8, n=8192 (efficiency 0.94) to p=16, n=16,384 (efficiency 0.97), efficiency improves. This suggests that scaling the problem size with the number of processes maintains or enhances efficiency, as the increased computational work (n2/p) offsets the communication overhead.

The document compares the program to two hypothetical programs: Program A, which maintains constant efficiency (0.75) regardless of n, is strongly scalable, while Program B, which maintains efficiency by increasing n proportional to p , is weakly scalable. The matrixvector multiplication program aligns with Program B, as its efficiency improves when n scales with p , particularly for p≥4. The exceptions occur at smaller p (e.g., p=2 to 4), but the document emphasizes that scalability discussions typically focus on large p, where the program’s weak scalability is evident.

This analysis underscores that the matrix-vector multiplication program is weakly scalable, suitable for large-scale systems where problem sizes can grow with available processes. However, its lack of strong scalability highlights the challenge of minimizing communication overhead in MPI programs to achieve efficient performance across a wide range of configurations.

7. A parallel sorting algorithm.

The parallel odd-even transposition sort is a sorting algorithm that leverages parallel processing to sort large datasets efficiently. Here's a summary of how it works in an MPI program:

Key Steps:

1. Data Distribution: The dataset is divided among MPI processes.
2. Odd-Even Phases: Processes compare and swap adjacent elements in odd and even phases.
3. Communication: Processes exchange data with neighbors during each phase.
4. Iteration: The algorithm iterates through multiple phases until the data is sorted.

Parallelization Benefits:

1. Improved Performance: Parallel processing reduces sorting time for large datasets.
2. Scalability: The algorithm can handle large datasets by adding more processes.

MPI Implementation:

1. Process Communication: MPI functions like MPI_Send and MPI_Recv facilitate data exchange between processes.
2. Synchronization: MPI's synchronization mechanisms ensure proper data exchange and comparison.

The parallel odd-even transposition sort is a efficient sorting algorithm for large datasets in distributed-memory architectures.

1. **Explain the structure and operation of the serial odd-even transposition sort algorithm, and discuss why it is more suitable for parallelization than bubble sort.**

Answer: The serial odd-even transposition sort, is a variant of bubble sort that organizes compare-swap operations into alternating even and odd phases to sort a list of n keys.

```
void Odd_even_sort(
   int  a[] ,   /*  in/out  */
int n      /* in */
) {     int phase, i, temp;
for (phase = 0; phase < n; phase++) {
 if (phase % 2 == 0) {   /* Even phase */
for (i = 1; i < n; i += 2) {
if (a[i - 1] > a[i]) {
```

```
temp = a[i];
a[i] = a[i - 1];
a[i - 1] = temp;
            }
        }
    } else {   /* Odd phase */
for (i = 1; i < n - 1; i += 2) {
if (a[i] > a[i + 1]) {
temp = a[i];
a[i] = a[i + 1];
 a[i + 1] = temp;
            }
        }
    }
  }
} /* Odd_even_sort */
```

The algorithm, shown in Above Program, operates as follows: In even phases, it performs compare-swaps on pairs of elements at indices (a[0],a[1]),(a[2],a[3]),(a[4],a[5]),… (a[0], a[1]),

(a[2], a[3]), (a[4], a[5]),…, and in odd phases, it performs compare-swaps on pairs (a[1],a[2]),(a[3],a[4]),(a[5],a[6]),… (a[1], a[2]), (a[3], a[4]), (a[5], a[6]), …. If a pair is out of order (i.e., a[i]>a[i+1]), the elements are swapped. This process continues for up to n n n phases, as guaranteed by the theorem that after n phases, the list is sorted.

An example with the list [5,9,4,3] illustrates the process:

1. **Even phase**: Compare-swap (5,9) (no swap) and (4,3) (swap), yielding [5,9,3,4]
2. **Odd phase**: Compare-swap (9,3) (swap), yielding [5,3,9,4]
3. **Even phase**: Compare-swap (5,3) (swap) and (9,4) (swap), yielding [3,5,4,9]
4. **Odd phase**: Compare-swap (5,4) (swap), yielding [3,4,5,9] which is sorted. The key feature of odd-even transposition sort is that all compare-swaps within a phase are independent and can occur simultaneously, unlike bubble sort, where comparisons must follow a strict sequential order. In bubble sort (Program 3.14), each pass moves the largest unsorted element to its final position, but the sequential dependency (e.g., comparing a[i−1] and a[i] before a[i] and a[i+1]) prevents parallel execution. For instance, with [9,5,7], bubble sort requires swapping 9 and 5, then 9 and 7, in order, to get [5,7,9]; out-of-order comparisons yield incorrect results like [5,9,7].

The decoupled compare-swaps in odd-even transposition sort make it more suitable for parallelization, as multiple pairs can be processed concurrently by different processes or threads, enabling efficient distribution of work in a parallel environment, as exploited in the parallel version described in Section 3.7.2

2. **Describe the parallel odd-even transposition sort algorithm for the case where n=p, and explain how it is extended to handle n>p,.**

   Answer: The parallel odd-even transposition sort algorithm, is designed for a distributed memory system with p processes, each initially holding n/p keys. The algorithm ensures that, upon termination, each process's keys are sorted, and keys on process q are less than or equal to those on process r for q<r. The document describes its operation for two cases: when n=p (one key per process) and when n>p (multiple keys per process).

   Case: n=p

   When n=p, each process holds one key, and the algorithm mimics the serial odd-even transposition sort across processes. In each phase, processes exchange keys with a partner process and keep either the smaller or larger key based on their rank. For even phases, oddranked processes (e.g., rank 1) exchange with rank myrank−1 (e.g., 0), and even-ranked processes (e.g., rank 0) exchange with myrank+1(e.g., 1). In odd phases, the pairings reverse: odd-ranked processes exchange with myrank+1, and even-ranked processes with myrank−1. After exchanging keys, the lower-ranked process keeps the smaller key, and the higher-ranked keeps the larger. Processes at boundaries (e.g., rank 0 in even phases or rank p−1 in odd phases) may be idle, with their partner set to MPI_PROC_NULL, indicating no communication. The algorithm requires at most p p p phases to sort the keys, as per the theorem in Section 3.7.2.

   Case: n>p

   When each process holds n/p>1 keys, the algorithm extends the n=p case by adapting the exchange and selection process. Initially, each process sorts its local n/p keys using a fast serial algorithm like qsort. Then, for up to p phases, processes exchange all their keys with a partner, determined similarly to the n=p case. For example, in an even phase, processes 0 and 1 exchange all n/p keys, as do processes 2 and 3. After receiving the partner's keys, each process has 2n/p keys (its own and the partner's). The lower-ranked process (e.g., 0) keeps the smallest n/p n/p n/p keys, and the higher-ranked process (e.g., 1) keeps the largest n/p keys. This process is illustrated in Table 3.8 for p=4, n=16: after local sorting, phase 0 exchanges keys between 0-1 and 2-3, phase 1 exchanges between 1-2, and so on, resulting in a globally sorted list after 4 phases.

   The extension to n>p leverages the fact that local keys are pre-sorted, reducing the complexity of selecting the smallest or largest n/p keys. Instead of sorting 2n/p keys, processes merge the two sorted lists of n/p keys, stopping once the required n/p keys. The partner computation and phase structure remain identical to the n=p case, with idle processes handled similarly. The algorithm's worst-case performance requires p phases, ensuring scalability for larger problem sizes by distributing the workload across processes.

3. **Discuss the safety issues in MPI communication for the parallel odd-even transposition sort, and explain how the algorithm addresses them using MPI_Sendrecv**

Answer: safety issues in MPI communication for the parallel odd-even transposition sort, particularly when using MPI_Send and MPI_Recv, and explains how MPI_Sendrecv resolves these issues to ensure a safe program.

Safety Issues:

The parallel odd-even transposition sort requires processes to exchange keys with their partners in each phase. A naive implementation using MPI_Send followed by MPI_Recv (e.g., MPI_Send(my_keys, n/comm_sz, MPI_INT, partner, 0, comm); MPI_Recv(temp_keys, n/comm_sz, MPI_INT, partner, 0, comm, MPI_STATUS_IGNORE);) can lead to an unsafe program. The issue arises because MPI_Send may either buffer the message or block until the matching MPI_Recv starts, depending on the message size and MPI implementation. If both partner processes execute MPI_Send first and the sends block (common for large messages), neither can proceed to MPI_Recv, causing a deadlock where each process waits indefinitely for the other's receive. Such a program is unsafe because its correct execution depends on MPI buffering, which is unreliable for larger messages or different implementations.

To check for safety, the document suggests replacing MPI_Send with MPI_Ssend, which always blocks until the matching receive starts. If the program runs correctly with MPI_Ssend, it is safe; otherwise, it may hang, confirming the unsafe nature of the original communication structure.

Solution Using MPI_Sendrecv:

To address this, the algorithm restructures communication to avoid simultaneous sends. The document proposes using MPI_Sendrecv, which performs a blocking send and receive in a single call, with the MPI implementation scheduling them to prevent deadlock. The syntax is: MPI_Sendrecv(my_keys, n/comm_sz, MPI_INT, partner, 0,   recv_keys, n/comm_sz, MPI_INT, partner, 0, comm, MPI_STATUS_IGNORE);

This function ensures that the send and receive operations are coordinated safely, regardless of whether the partner is the same or different process. For example, in an even phase, processes 0 and 1 exchange keys using MPI_Sendrecv, and the MPI implementation guarantees that one process's send completes with the other's receive without requiring explicit scheduling by the programmer. This eliminates the need for complex code to alternate sends and receives based on process rank (e.g., even-ranked processes send first, odd-ranked receive first), as shown in the alternative approach .

The use of MPI_Sendrecv simplifies the parallel odd-even sort implementation (Section 3.7.4) and ensures safety across all inputs and communicator sizes. The document notes that an alternative, MPI_Sendrecv_replace, could be used if the send and receive buffers are the same, but the algorithm uses separate buffers (my_keys and recv_keys) for clarity. By adopting MPI_Sendrecv, the algorithm avoids deadlocks, making it robust and reliable for parallel execution on distributed memory systems.

4. **Analyze the performance of the parallel odd-even transposition sort based on the run-time results in Table , and discuss the factors affecting its efficiency**

Answer: run-time results for the parallel odd-even transposition sort in Table 3.9, with times in milliseconds for sorting 200, 400, 800, 1600, and 3200 thousand keys using 1, 2, 4, 8, and 16 processes. The analysis of these results, combined with the algorithm's design, reveals its performance characteristics and factors affecting efficiency.

Run-Time Analysis:

Table below shows that for a fixed number of processes (p), increasing the number of keys (n) roughly doubles the run-time when n doubles, consistent with the serial quicksort used for local sorting.

Run-times of parallel odd-even sort (times are in milliseconds):

Processes	**200**	**400**	**800**	**1600**	**3200**
1	88	190	390	830	1800
2	43	91	190	410	860
4	22	46	96	200	430
8	12	24	51	110	220
16	7.5	14	29	60	130

For example, with p=1, run-times increase from 88 ms (n=200K) to 190 ms (n=400K), 390 ms (n=800K), 830 ms (n=1600K n =), and 1800 ms (n=3200K), approximately doubling each step. For p>1, the trend is similar but less pronounced due to parallelization. For instance, with p=4, times are 22 ms (n=200K), 46 ms (n=400K), 96 ms (n=800K), 200 ms (n=1600K), and 430 ms (n=3200K).

When fixing n and increasing p, run-times decrease significantly, indicating parallel speedup. For n=3200K, run-times drop from 1800 ms (p=1) to 860 ms (p=2), 430 ms (p=4), 220 ms (p=8), and 130 ms (p=16), showing that doubling p roughly halves the time for smaller p. However, the speedup diminishes as p increases; for p=16, the time (130 ms) is only 1.7 times faster than for p=8 (220 ms), suggesting communication overhead impacts performance at higher process counts.

Factors Affecting Efficiency:

The efficiency of the parallel odd-even transposition sort is influenced by several factors:

1. **Local Sorting**: Each process sorts n/p keys using qsort, which dominates for p=1 and remains significant for small p. The document notes that the single-process case uses quicksort, not serial odd-even sort, explaining the relatively fast serial times.

2. **Communication Overhead**: The algorithm requires up to p phases of key exchanges using MPI_Sendrecv. Communication is costly compared to local computation, especially for large p, where multiple phases increase overhead. For n=200K, the run-time for p=16 (7.5 ms) is closer to p=8 (12 ms), indicating that communication dominates for small n and large p.
3. **Merge Operation**: After exchanging keys, processes merge two sorted lists of n/p keys to select the smallest or largest n/p keys (Program). The optimized merge reduces computational cost compared to sorting 2n/p keys, but it still adds to the run-time, particularly for large n.

```
void Merge_low(
   int my_keys[],      /* in/out */
int recv_keys[],   /* in */      int
temp_keys[],  /* scratch */     int
local_n       /* n/p, in */
) {     int m_i, r_i, t_i;      m_i = r_i = t_i = 0;
while (t_i < local_n) {       if (my_keys[m_i]
<= recv_keys[r_i]) {            temp_keys[t_i]
= my_keys[m_i];          t_i++;           m_i++;
}  else  {                        temp_keys[t_i]  =
recv_keys[r_i];             t_i++;             r_i++;
       }
   }
   for    (m_i    =    0;    m_i    <    local_n;    m_i++)
my_keys[m_i] = temp_keys[m_i];
} /* Merge_low */
```

4. **Scalability**: The algorithm is weakly scalable, as efficiency improves when n increases with p. For larger n (e.g., 3200K), the computational work (n/p) outweighs communication overhead, leading to better speedup (e.g., 1800 ms to 130 ms for p=16). For small n , communication limits efficiency, as seen with n=200K .

The communication costs make the algorithm impractical for n=p, as message exchanges outweigh computational benefits. The “final improvement” (avoiding array copying by swapping pointers) reduces overhead, contributing to the reported times. Overall, the algorithm performs well for large n and moderate p , but efficiency decreases with large p due to communication, aligning with typical MPI program behavior where overhead dominates for small problem sizes and many processes.

Shared-memory programming with OpenMP – openmp pragmas and directives, The trapezoidal rule, Scope of variables, The reduction clause, loop carried dependency, scheduling, producers and consumers, Caches, cache coherence and false sharing in openmp, tasking, tasking, thread safety.

4 Module

Module -4 Lecturer Notes: Shared-memory programming with OpenMP

This Lecture Notes provides a comprehensive overview of **Shared-memory programming with OpenMP**, based on Chapter 5 of "Peter S Pacheco, Matthew Malensek – An Introduction to Parallel Programming, second edition, Morgan Kauffman , VTU syllabus based BCS702 Parallel Computing" It is designed to serve as a detailed lecture presentation for a classroom setting, covering the openmp pragmas and directives, Shared-memory programming with OpenMP programming basics, and practical examples, while ensuring clarity for students new to parallel computing.

MODULE-4: Shared-memory programming with OpenMP –

1. openmp pragmas and directives,
2. The trapezoidal rule,
3. Scope of variables,
4. The reduction clause,
5. loop carried dependency,
6. scheduling,
7. producers and consumers,
8. Caches,
9. cache coherence and false sharing in openmp,
10. tasking,
11. thread safety.

1. openmp pragmas and directives,

OpenMP (Open Multi-Processing) is an API for parallel programming in C, C++, and Fortran. Key pragmas and directives include:

1. #pragma omp parallel: Creates a parallel region, dividing work among threads.
2. #pragma omp for: Distributes loop iterations across threads.
3. #pragma omp section: Divides work into sections, executed by different threads.
4. #pragma omp critical: Ensures exclusive access to shared data.
5. #pragma omp barrier: Synchronizes threads at a specific point.

These directives enable parallelization of code, leveraging multi-core processors for improved performance.

1. **Explain the functionality and usage of the parallel directive in OpenMP, including how it interacts with threads and the role of the num_threads clause, as illustrated in the "hello, world" program (Program).**

   Answer: The parallel directive in OpenMP, as shown in Program , is used to fork a team of threads to execute a structured block of code in parallel. #include <stdio.h>

```
#include <stdlib.h> #include <omp.h>
void Hello(void); /* Thread function */
int main(int argc, char* argv[]) {     int
thread_count;

   /* Get number of threads from command line */
thread_count = strtol(argv[1], NULL, 10);

# pragma omp parallel num_threads(thread_count)
   Hello();

   return 0;
}
```

A structured block has a single entry and exit point (allowing exit calls). In the "hello, world" program, the directive #pragma omp parallel num_threads(thread_count) instructs the run-time system to start thread_count - 1 additional child threads, with the original (parent) thread continuing as part of the team. Each thread in the team executes the Hello function, which prints a message using its rank (omp_get_thread_num()) and the total number of threads (omp_get_num_threads()). The num_threads(thread_count) clause specifies the number of threads, determined from the command-line input via strtol. Without this clause, the system defaults to one thread per available core. After the block completes, an implicit barrier ensures all threads finish (e.g., return from Hello) before the parent thread proceeds to the return statement, terminating child threads. This directive simplifies thread management compared to Pthreads, as it abstracts thread creation and joining, allowing high-level parallelization with minimal code.

2. **Discuss the parallel for directive and its restrictions for parallelizing loops, using the trapezoidal rule example to illustrate its application and the importance of the reduction clause.**

Answer: The parallel for directive in OpenMP combines the parallel directive's thread forking with automatic division of a for loop's iterations among the threads, as shown in the trapezoidal rule example.

In the example, the serial code computes the integral using h = (b-a)/n; approx = (f(a)+f(b))/2.0; for (i=1; i<=n-1; i++) approx += f(a+i*h); approx = h*approx;. By adding #pragma omp parallel for num_threads(thread_count) reduction(+:approx), the loop is parallelized, with iterations split among threads (typically in blocks, e.g., first m/thread_count iterations to thread 0). The loop variable i is private by default, ensuring each thread has its own copy, avoiding race conditions on i++. The reduction(+:approx) clause is critical because approx is updated in each iteration (approx += f(a+i*h)), creating a potential race condition if shared. The clause creates private copies of approx for each thread, initialized to 0, and adds them to the shared approx at the loop's end, ensuring correct summation.

for (index = start; index < end; index++) for (index = start; index < end; ++index) for (index = start; index < end; index--) for (index = start; index < end; --index) for (index = start; index < end; index += incr) for (index = start; index < end; index -= incr) for (index = start; index < end; index = index + incr) for (index = start; index < end; index = incr + index) for (index = start; index < end; index = index - incr) for (index = start; index < end; index = index + 1) for (index = start; index > end; index--) for (index = start; index >= end; index--)

Restrictions include that the loop must be in canonical form , with determinable iteration counts before execution, integer/pointer index, and unchanging start/end/increment expressions. Noncanonical loops (e.g., with break or infinite loops) cannot be parallelized, ensuring the runtime system can partition iterations safely.

3.Describe how the critical and reduction directives address race conditions in the trapezoidal rule program and compare their approaches to managing shared variable updates.

Answer: In the trapezoidal rule program, a race condition arises when multiple threads update the shared global_result variable via global_result_p += my_result. The critical directive (#pragma omp critical) in Program ensures mutual exclusion by allowing only one thread at a time to execute global_result_p += my_result, preventing concurrent updates that could overwrite values (e.g., as shown in the timetable on page 230). Each thread computes its local trapezoid areas in Trap, storing them in my_result, then adds to global_result within the critical section, ensuring correct summation but serializing the update step. In contrast, the reduction clause, offers a cleaner solution: #pragma omp parallel num_threads(thread_count) reduction(+:global_result) allows each thread to call Local_trap and add its result to a private copy of global_result, initialized to 0. At the parallel block's end, OpenMP combines these private copies into the shared global_result using addition, automatically handling synchronization. The reduction approach avoids explicit critical sections, allowing parallel execution of Local_trap, unlike an incorrect critical section around the function call that serializes execution. The critical directive is more general but requires explicit placement, while reduction is specialized for associative operations like addition, automating private variable management and combination, improving performance and readability.

```
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>
 void Trap(double a, double b, int n, double* global_result_p);
 int main(int argc, char* argv[]) {      double global_result = 0.0;      /* Store our result in
global_result */      double a, b;                    /* Left and right endpoints */      int n;
/* Total number of trapezoids */     int thread_count;
     thread_count = strtol(argv[1], NULL, 10);     printf("Enter a, b, and n\n");     scanf("%lf
%lf %d", &a, &b, &n);
 #   pragma omp parallel num_threads(thread_count)
    Trap(a, b, n, &global_result);
     printf("With n = %d trapezoids, our estimate\n", n);     printf("of the integral from %f to
%f = %.14e\n",           a, b, global_result);
     return 0; }  /* main */ void Trap(double a, double b, int n, double* global_result_p) {
double h, x, my_result;     double local_a, local_b;     int i, local_n;
    int my_rank = omp_get_thread_num();     int
thread_count = omp_get_num_threads();      h
= (b - a) / n;       local_n = n / thread_count;
local_a = a + my_rank * local_n * h;     local_b
= local_a + local_n * h;         my_result =
(f(local_a) + f(local_b)) / 2.0;     for (i = 1; i <=
local_n - 1; i++) {           x = local_a + i * h;
my_result += f(x);
    }
    my_result = my_result * h;
 #   pragma omp critical
    *global_result_p += my_result;
}  /* Trap */
```

4. **Analyze the role of the task and taskwait directives in parallelizing recursive algorithms, using the Fibonacci number calculation (Program) as an example, and explain how data scoping affects correctness.**

Answer: The task and taskwait directives, introduced in OpenMP 3.0, enable parallelization of dynamic problems like recursive algorithms, as shown in the Fibonacci program (Program).

```
int fib(int n) {
int i = 0;     int
j = 0;
    if (n <= 1) {        fibs[n] = n;        return n;     }
#   pragma omp task shared(i)     i = fib(n - 1);
#   pragma omp task shared(j)     j = fib(n - 2);
#   pragma omp taskwait
   fibs[n]    =    i    +    j;
return fibs[n];
}
```

The serial fib function recursively computes Fibonacci numbers, storing them in a global array fibs. To parallelize, a parallel directive creates a thread team, and a single directive ensures one thread generates tasks. In fib, each recursive call (fib(n-1) and fib(n-2)) is preceded by #pragma omp task shared(i) and #pragma omp task shared(j), creating tasks that compute these values in parallel. Without proper scoping, the default private scope for i and j causes their values to be lost post-task, resulting in fibs[n] = 0 + 0. Declaring i and j as shared ensures the task results are stored in the parent thread's variables. However, task execution order is nondeterministic, so the parent thread might update fibs[n] before tasks complete. The #pragma omp taskwait directive acts as a barrier, ensuring fib(n-1) and fib(n-2) finish before fibs[n] = i + j, guaranteeing correct results. The if(n>20) clause further optimizes by limiting task creation for small n, reducing overhead. This demonstrates how task enables parallel recursive calls, taskwait ensures synchronization, and explicit scoping (shared) ensures data integrity, making OpenMP suitable for dynamic computations like recursion.

2. The trapezoidal rule,

The trapezoidal rule is a numerical integration technique used to approximate the definite integral of a function. It works by:

1. Dividing the area under the curve into trapezoids
2. Calculating the area of each trapezoid
3. Summing the areas to estimate the total integral

The rule approximates the curve by connecting function values at regular intervals with straight lines, forming trapezoids. The formula combines the function values at the endpoints and intermediate points, weighted by the interval width. While simple, the trapezoidal rule provides a reasonable estimate of the integral, especially for smooth functions or small intervals.

1. **Explain the serial implementation of the trapezoidal rule including the mathematical formula and the corresponding C code.**

   Answer: The trapezoidal rule estimates the area under a curve y=f(x) from x=a to x=b by dividing the interval [a,b] into n subintervals of equal length h=(b−a)/n. Each subinterval is approximated by a trapezoid, and the total area is the sum of these trapezoid areas. The mathematical formula is h[f(x0)/2+f(x1)+f(x2)+…+f(xn−1)+f(xn)/2], where xi=a+i for i=0,1,…,n. The serial C code implementing this is: h = (b-a) / n;

```
approx = (f(a) + f(b)) / 2.0;
for (i = 1; i <= n-1; i++) {
x_i = a + i * h;    approx +=
f(x_i);
```

} approx = h * approx; Here, h is the width of each subinterval, approx is initialized with the average of the function values at the endpoints f(a) and f(b), and the loop adds the function values at interior points xi. The final multiplication by h scales the sum to approximate the integral. This code assumes inputs a, b, and n, and a function f(x), providing a straightforward sequential computation.

2. **Describe how the trapezoidal rule is parallelized in Above Program, including the role of the Trap function and the handling of the shared variable global_result.**

   Answer: Above Program parallelizes the trapezoidal rule using OpenMP by distributing the computation of trapezoid areas across multiple threads. The main function initializes a shared variable global_result = 0.0, reads inputs a, b, and n , and gets the number of threads (thread_count) from the command line. The parallel directive #pragma omp parallel num_threads(thread_count) forks a team of threads, each executing the Trap function with arguments a, b, n, and a pointer to global_result. In Trap, each thread:

   1. Gets its rank (my_rank) and the total number of threads using omp_get_thread_num() and omp_get_num_threads().
   2. Calculates local_n = n/thread_count, the number of trapezoids per thread.
   3. Determines its subinterval's left endpoint as local_a = a + my_rank*local_n*h and right endpoint as local_b = local_a + local_n*h, where h = (b-a)/n.
   4. Computes its contribution to the integral by applying the trapezoidal rule to its subinterval, storing the result in my_result.

5. Adds my_result to global_result within a critical section (#pragma omp critical) to prevent race conditions, as multiple threads updating global_result concurrently could lead to errors (e.g., overwriting values, as shown in the timetable on page 230). The critical directive ensures exclusive access, serializing updates but ensuring correctness. After the parallel block, the parent thread prints global_result. The program assumes n is divisible by thread_count to avoid using fewer trapezoids than requested, ensuring each thread processes a contiguous block of trapezoids.

3. **Analyze the potential race condition in the parallel trapezoidal rule implementation (Program) and how the critical directive resolves it, including the implications for program performance.**

Answer: In Program , a race condition occurs when multiple threads attempt to update the shared variable global_result via global_result_p += my_result in the Trap function.

```
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>
void Trap(double a, double b, int n, double* global_result_p);

int main(int argc, char* argv[]) {      /* We'll store
our result in global_result */    double global_result
= 0.0;       double a, b;                  /* Left and right
endpoints */     int n;                    /* Total number of
trapezoids */     int thread_count;

    thread_count = strtol(argv[1], NULL, 10);
printf("Enter a, b, and n\n");      scanf("%lf
%lf %d", &a, &b, &n);

#   pragma omp parallel num_threads(thread_count)
    Trap(a, b, n, &global_result);

    printf("With n = %d trapezoids, our estimate\n", n);
printf("of the integral from %f to %f = %.14e\n",
a, b, global_result);

    return 0;
}  /* main */
```

```
void Trap(double a, double b, int n, double* global_result_p) {
double h, x, my_result;      double local_a, local_b;      int i,
local_n;
   int my_rank = omp_get_thread_num();    int
thread_count = omp_get_num_threads();

   h = (b - a) / n;    local_n = n / thread_count;
local_a = a + my_rank * local_n * h;
local_b = local_a + local_n * h;    my_result
= (f(local_a) + f(local_b)) / 2.0;    for (i = 1;
i <= local_n - 1; i++) {        x = local_a + i *
h;        my_result += f(x);    }

   my_result = my_result * h;

#  pragma omp critical
   *global_result_p += my_result;
}  /* Trap */
```

A race condition arises because multiple threads access a shared resource (global_result), at least one access is an update, and concurrent updates can produce incorrect results. For example, if thread 0 has my_result = 1 and thread 1 has my_result = 2, simultaneous execution of global_result += my_result could result in global_result = 1 or 2 instead of 3, as one thread's update may overwrite another's. The #pragma omp critical directive resolves this by ensuring mutual exclusion: only one thread at a time can execute global_result_p += my_result, while others wait, guaranteeing that each thread's contribution is added correctly. This critical section, however, serializes the updates, potentially impacting performance, as threads must wait for access, reducing parallelism. For large thread_count, this bottleneck can significantly slow execution, especially if the computation of my_result is quick compared to the synchronization overhead. The document suggests an alternative (using a reduction clause), which would allow parallel computation without serializing updates, but within, the critical directive is the chosen solution, prioritizing correctness over optimal performance.

4. **Discuss the design considerations for parallelizing the trapezoidal rule, including task partitioning, communication, and error handling for uneven divisions.**

Answer: Foster's parallel program design methodology to parallelize the trapezoidal rule, considering task partitioning, communication, and aggregation. Two types of jobs are identified: (a) computing individual trapezoid areas, which are independent, and (b) summing these areas, which requires communication. To parallelize, the methodology assumes many more trapezoids than cores, so tasks are aggregated by assigning contiguous blocks of trapezoids to each thread, effectively partitioning [a,b][a, b][a,b] into subintervals (illustrated in Figure 5.4).

Each thread applies the serial trapezoidal rule to its subinterval, computing a local sum (my_result). Communication occurs when threads add their my_result to the shared global_result, necessitating a critical section (#pragma omp critical) to avoid race conditions. For aggregation, the critical section ensures all local sums are correctly added to global_result. A key error-handling consideration is ensuring n is evenly divisible by thread_count. If not, local_n = n/thread_count results in fewer trapezoids (e.g., for n=14, thread_count=4, each thread uses 3 trapezoids, totaling
12 instead of 14). The document suggests checking this with code like if (n %
thread_count != 0)
{ fprintf(stderr, "n must be evenly divisible by thread_count\n"); exit(0); }.

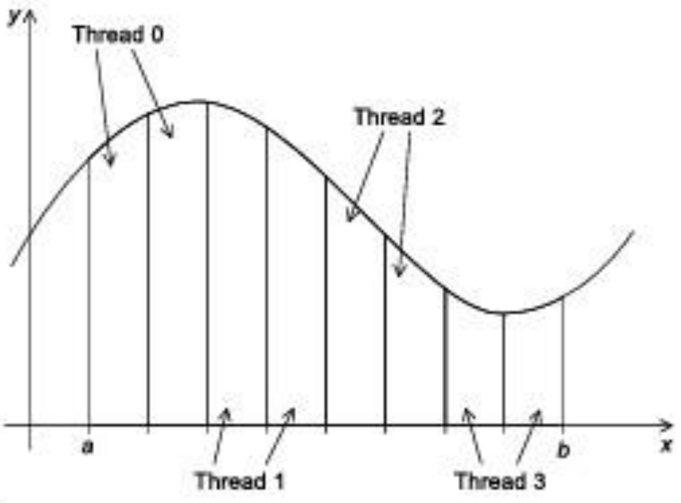


FIGURE 5.4

Assignment of trapezoids to threads

[Refer: "Peter S Pacheco, Matthew Malensek – An Introduction to Parallel Programming, second edition, Morgan Kauffman]

This ensures all n trapezoids are used, maintaining accuracy. The design balances parallelism by distributing independent computations while managing shared resource updates and addressing potential errors in task division.

3. Scope of variables,

In OpenMP, variable scope determines how variables are shared or privatized among threads. Key aspects include:

1. Shared variables: Accessible by all threads, shared variables require synchronization to avoid data races.
2. Private variables: Each thread has its own copy, private variables are not shared.
3. Default sharing: Variables are shared or private based on predefined rules.

4. Clause specifications: Data-sharing attributes can be explicitly specified using clauses like shared, private, firstprivate, and lastprivate.

Understanding variable scope is crucial for writing correct and efficient OpenMP parallel code, ensuring data consistency and avoiding synchronization issues.

1. **Explain the concept of variable scoping in OpenMP, including the differences between shared and private scopes and their impact on parallel execution.**

   Answer: In OpenMP, variable scoping determines how variables are accessed by threads in a parallel block, with two primary scopes: **shared** and **private**. A **shared** variable has a single memory location accessible by all threads, meaning any thread can read or modify it, which is the default scope in a parallel block. This is useful for variables that need consistent values across threads, like thread_count in the example, but can lead to race conditions if multiple threads update it concurrently. A **private** variable, conversely, gives each thread its own uninitialized copy, ensuring no interference between threads. For example, my_rank is declared private in the directive #pragma omp parallel private(my_rank) shared(thread_count), so each thread has its own my_rank to store its unique thread ID from omp_get_thread_num(). The document emphasizes that private variables are undefined at the start and end of the parallel block unless explicitly managed (e.g., via firstprivate or lastprivate clauses, not detailed in this section). The default(none) clause can be used to force explicit scoping of all variables, reducing errors by ensuring programmers consciously assign scopes. This scoping mechanism is critical for correct parallel execution, preventing unintended data sharing (e.g., race conditions) while allowing controlled access to shared resources, as seen in the example where my_rank is private to avoid conflicts and thread_count is shared for consistent access.

2. **Describe how the example code uses variable scoping to ensure correct execution of the parallel "hello, world" program, and discuss the role of the default(none) clause.**

   Answer: The example code, a parallel "hello, world" program, uses OpenMP's variable scoping to ensure correct execution by explicitly defining the scopes of key variables. The program includes a parallel directive #pragma omp parallel private(my_rank) shared(thread_count), which forks a team of threads to execute the Hello function. The variable my_rank, which stores the thread's ID from omp_get_thread_num(), is declared **private**, ensuring each thread has its own uninitialized copy. This prevents race conditions, as each thread independently assigns and uses its my_rank to print a unique greeting (e.g., "Hello from thread 2 of 4"). The variable thread_count, obtained from omp_get_num_threads(), is declared **shared**, allowing all threads to access the same value, which is necessary for printing the total number of threads consistently. Without explicit scoping, the default shared scope could cause issues if my_rank were shared, leading to unpredictable values due to concurrent assignments. The document suggests using #pragma omp parallel default(none) private(my_rank) shared(thread_count) to force explicit scoping of all variables, preventing errors from unconsidered variables assuming the default shared scope. In this case, default(none) ensures that only my_rank and thread_count are scoped as intended, enhancing code reliability by catching potential scoping oversights during compilation, thus ensuring correct parallel execution.

3. **Analyze the implications of the default shared scope in OpenMP parallel blocks, and how explicit scoping with private and shared clauses mitigates potential issues in parallel programming.**

   Answer: the default scope for variables in an OpenMP parallel block is **shared**, meaning all threads access the same memory location for a variable. This can simplify programming for variables that need consistent values across threads, such as thread_count in the example, which all threads read to report the total number of threads. However, the default shared scope poses risks, particularly race conditions, if multiple threads update a variable concurrently, leading to unpredictable results (e.g., as seen in later sections with shared variables like global_result). To mitigate this, OpenMP provides private and shared clauses to explicitly control scoping. In the example, #pragma omp parallel private(my_rank) shared(thread_count) makes my_rank private, giving each thread its own copy to store its unique thread ID from omp_get_thread_num(), preventing conflicts during assignment. The shared(thread_count) clause ensures all threads access the same thread_count, maintaining consistency without modification risks, as it's read-only in the Hello function. The default(none) clause further enhances safety by requiring explicit scoping for all variables, forcing programmers to consider each variable's role and preventing accidental shared access that could cause errors. For instance, if my_rank were accidentally left as shared, concurrent assignments could corrupt its value. Explicit scoping thus ensures thread safety and correctness, while default(none) reduces errors by enforcing deliberate scope decisions, critical for robust parallel programming in OpenMP.

4. The reduction clause,

The reduction clause in OpenMP enables parallelization of reduction operations, such as sum, product, or minimum/maximum calculations. It allows threads to accumulate partial results, which are then combined to produce the final result. The reduction clause:

1. Specifies the operator: Defines the reduction operation (e.g., +, *, min, max).
2. Creates private copies: Each thread has a private copy of the reduction variable.
3. Combines partial results: Partial results are combined using the specified operator.

The reduction clause simplifies parallelizing reduction operations, improving performance and scalability in OpenMP parallel programs.

1. **Explain how the reduction clause in OpenMP improves the parallel trapezoidal rule implementation in Program, focusing on its mechanism for handling the shared variable global_result.**
   Answer: In Program , the original parallel trapezoidal rule implementation uses a critical section (#pragma omp critical) to protect updates to the shared variable global_result, where each thread adds its local sum (my_result) from the Local_trap function. This approach causes a race condition without protection, as multiple threads updating global_result concurrently can lead to incorrect results . The reduction clause, introduced, improves this by modifying the parallel directive to #pragma omp parallel num_threads(thread_count) reduction(+:global_result). This clause creates a private copy of global_result for each thread, initialized to 0 (the identity for addition). Each thread executes Local_trap and adds its result to its private copy without interference. At the end of the parallel block, OpenMP combines these private copies using the + operator and updates the shared global_result with the sum. This eliminates the need for a critical section, allowing Local_trap calls to run fully in parallel without serialization. The document notes that placing a critical section around the Local_trap call itself would serialize execution, defeating parallelism, whereas the reduction clause ensures correct summation while maintaining parallel efficiency.
2. **Describe the process by which the reduction clause manages variable updates in the context of Program , and compare its performance implications to using a critical section.** **Answer**: In Program , the reduction clause manages updates to the shared variable global_result by automating the process of combining thread-local results. With the directive #pragma omp parallel num_threads(thread_count) reduction(+:global_result), OpenMP creates a private copy of global_result for each thread, initialized to 0 (the identity for the + operator). Each thread executes the Local_trap function, which computes the trapezoidal rule approximation for its assigned subinterval, and adds its result to its private copy of global_result. At the end of the parallel block, OpenMP combines these private copies by adding them together and updates the shared global_result with the final sum. This process avoids race conditions without requiring explicit synchronization. In contrast, the original Program uses #pragma omp critical to ensure only one thread at a time updates global_result with global_result_p += my_result, serializing these updates and creating a performance bottleneck, especially with many threads. The document highlights that the reduction clause allows Local_trap calls to execute in parallel, improving performance by eliminating the wait times associated with the critical section. However, the reduction clause is limited to associative operations like addition, whereas critical sections are more general but less efficient due to serialization, making reduction preferable for summation tasks like the trapezoidal rule.

3. **Analyze the role of the reduction clause in preventing race conditions in the parallel trapezoidal rule program, and discuss why it is a more elegant solution than the critical section approach in Program .**

   Answer: The reduction clause in Program prevents race conditions when updating the shared variable global_result by providing a structured mechanism to handle concurrent updates. Without protection, multiple threads executing global_result_p += my_result in Local_trap could overwrite each other's updates, leading to incorrect results (e.g., as illustrated in the timetable on page 230). The reduction(+:global_result) clause, used in #pragma omp parallel num_threads(thread_count) reduction(+:global_result), creates a private copy of global_result for each thread, initialized to 0. Each thread adds its Local_trap result to its private copy without interference. At the parallel block's end, OpenMP sums these private copies and updates the shared global_result, ensuring correctness without explicit synchronization. Compared to the critical section approach (#pragma omp critical), which serializes updates by allowing only one thread to execute global_result_p += my_result at a time, the reduction clause is more elegant because it allows full parallelism in Local_trap execution, avoiding the performance bottleneck of threads waiting for critical section access. The document notes that an incorrect critical section around Local_trap would serialize the entire computation, whereas reduction automates the combination process, reducing code complexity and improving scalability for large thread_count. However, reduction is limited to operations like addition, making it less general but highly effective for the trapezoidal rule's summation task.

5. loop carried dependency,

In OpenMP, a loop-carried dependency occurs when iterations of a loop depend on results from previous iterations, making parallelization challenging. This dependency can lead to:

1. Incorrect results: Parallel execution may produce different results due to unpredictable order of operations.
2. Serialization: The loop may need to be executed sequentially to ensure correct results.

To address loop-carried dependencies, OpenMP provides techniques like:

1. Private variables: Creating private copies of variables to eliminate dependencies.
2. Reduction operations: Using reduction clauses to accumulate results.
3. Synchronization: Implementing synchronization mechanisms to control access to shared variables.

Identifying and managing loop-carried dependencies is crucial for effective parallelization and ensuring correctness in OpenMP parallel programs.

1. **Explain the concept of a loop-carried dependency, using the provided example to illustrate why it prevents parallelization with OpenMP.**

   Answer: A loop-carried dependency, occurs when the execution of one iteration of a loop depends on the result of a previous iteration, making it impossible to execute iterations in parallel without risking incorrect results. The document provides the example for (i = 1; i < n; i++) a[i] = a[i-1] + b[i];. In this loop, each iteration computes a[i] using a[i-1], the value of a from the previous iteration. For instance, to compute a[1], the loop uses a[0]; to compute a[2], it uses a[1], and so on. This dependency means that iteration i i i cannot start until iteration i−1 i-1 i−1 has completed, enforcing a sequential execution order. The #pragma omp parallel for directive in OpenMP assumes that loop iterations are independent, allowing them to be distributed across threads arbitrarily. In this case, parallelizing the loop would lead to race conditions or incorrect values, as threads might access a[i-1] before it is computed. For example, if thread 1 computes a[2] while thread 0 is still computing a[1], the value of a[1] may be undefined or outdated, leading to errors. The document emphasizes that such loops cannot be parallelized with #pragma omp parallel for and suggests finding an alternative algorithm with independent iterations to enable parallelization.

2. **Discuss the implications of loop-carried dependencies for parallel programming in OpenMP, including the challenges they pose and the approach to address them.**

   Answer: Loop-carried dependencies, pose significant challenges for parallel programming in OpenMP because they violate the requirement of the #pragma omp parallel for directive that loop iterations be independent. A loop-carried dependency occurs when an iteration depends on the result of a previous iteration, as in the example for (i = 1; i < n; i++) a[i] = a[i-1] + b[i];, where a[i] requires a[i-1]. This dependency enforces a sequential order, as each iteration must wait for the prior one to compute its input, preventing threads from executing iterations concurrently without risking race conditions or incorrect results. For instance, if this loop were parallelized, a thread computing a[2] might access an uncomputed or incorrect a[1], leading to unpredictable outcomes. The document highlights that OpenMP's parallel for construct cannot handle such loops because it assumes iterations can be executed in any order. The suggested approach is to find an alternative algorithm where iterations are independent. For example, if the computation can be reformulated to eliminate the dependency on previous iterations (e.g., by precomputing necessary values or restructuring the algorithm), parallelization becomes feasible. This requires careful algorithm design, as not all problems with loop-carried dependencies have obvious parallel equivalents, posing a challenge for developers aiming to leverage OpenMP's parallelism for performance gains.

3. **Analyze why the loop for (i = 1; i < n; i++) a[i] = a[i-1] + b[i]; cannot be parallelized with OpenMP's #pragma omp parallel for, and evaluate the potential consequences of ignoring this limitation.**

   Answer: The loop for (i = 1; i < n; i++) a[i] = a[i-1] + b[i];, cannot be parallelized with OpenMP's #pragma omp parallel for directive due to a loop-carried dependency. This dependency arises because each iteration i i i requires the value of a[i-1], which is computed in the previous iteration. For example, computing a[1] depends on a[0], a[2] depends on a[1], and so forth, creating a chain of dependencies that enforces sequential execution. The #pragma omp parallel for directive assumes that loop iterations are independent, allowing OpenMP to distribute iterations across threads in any order. If this loop were incorrectly parallelized, threads executing different iterations concurrently might access a[i-1] before it is computed, leading to race conditions or undefined behavior. For instance, if thread 0 computes a[1] while thread 1 computes a[2], thread 1 may use an incorrect or uninitialized a[1], resulting in wrong values in a. The document warns that such incorrect parallelization leads to unpredictable results, undermining program correctness. The only way to parallelize this computation would be to reformulate the algorithm to remove the dependency, such as by computing values in a way that iterations do not rely on previous results. Ignoring this limitation risks producing incorrect outputs, emphasizing the need for careful analysis of loop dependencies before applying OpenMP parallelization.

6. scheduling,

The scheduling clause in OpenMP determines how loop iterations are distributed among threads. It controls the assignment of iterations to threads, allowing for:

1. Static scheduling: Iterations are divided into fixed-size chunks and assigned to threads.
2. Dynamic scheduling: Iterations are assigned to threads dynamically, as they become available.
3. Guided scheduling: Iterations are divided into chunks, with chunk size decreasing as the loop progresses.

The scheduling clause helps optimize parallel performance by:

1. Balancing workload: Distributing iterations to achieve better load balancing.
2. Reducing overhead: Minimizing scheduling overhead.
3. Improving scalability: Adapting to varying workloads.

Choosing the right scheduling strategy depends on the specific use case and characteristics of the loop.

1. **Explain the role of the schedule clause in OpenMP and describe how the static scheduling type works, including its advantages and potential drawbacks.**

   Answer: The schedule clause in OpenMP, controls how iterations of a parallel for loop (invoked with #pragma omp parallel for) are distributed among threads, specifying a scheduling type and an optional chunk size. The static scheduling type divides the loop's iterations into chunks of approximately equal size, typically n/thread_count n/\text{thread_count} n/thread_count if no chunk_size is specified, and assigns them to threads before the loop begins. For example, with #pragma omp parallel for schedule(static, 4), iterations are grouped into chunks of 4 and assigned to threads in a round-robin fashion.

   This ensures each thread gets a fixed set of iterations, as shown in the document's example where 16 iterations with chunk_size=4 and 4 threads result in thread 0 getting iterations 0–3, thread 1 getting 4–7, and so on. The advantage of static scheduling is its simplicity and low overhead, as assignments are determined upfront, making it efficient for loops where iterations take roughly the same time (e.g., array operations). However, a drawback is potential load imbalance if iteration times vary significantly, as threads with "heavy" iterations may take longer, leaving others idle. The document notes that for such cases, static scheduling can lead to poor performance, whereas dynamic or guided scheduling may better balance workloads by assigning iterations dynamically.

2. **Compare and contrast the dynamic and guided scheduling types in OpenMP, and discuss their suitability for loops with varying iteration execution times.**

   Answer: dynamic and guided as two dynamic scheduling types in OpenMP for distributing parallel for loop iterations. In dynamic scheduling (#pragma omp parallel for schedule(dynamic, chunk_size)), iterations are divided into chunks of a specified size (defaulting to 1 if unspecified), and each chunk is assigned to a thread as it becomes available. This allows threads that finish early to take on new chunks, reducing idle time. In guided scheduling (#pragma omp parallel for schedule(guided, chunk_size)), chunks are also assigned dynamically, but their size decreases as the loop progresses, starting larger and shrinking to a minimum (specified or implementation-defined). For example, with chunk_size=4, early chunks in guided scheduling are larger, becoming smaller as fewer iterations remain. Both types are suitable for loops with varying iteration times, as they prevent load imbalance by redistributing work dynamically. Dynamic scheduling ensures consistent chunk sizes, which can be simpler to predict but may incur more scheduling overhead due to frequent assignments. Guided scheduling reduces overhead by assigning larger chunks initially, but the variable chunk size may complicate performance tuning. The document highlights that both are superior to static scheduling for imbalanced loops, as they allow threads to finish at roughly the same time, improving efficiency, though guided may be more efficient for loops with gradually decreasing work due to its adaptive chunk sizing.

3. **Analyze the impact of scheduling choices on the performance of a parallel for loop in OpenMP, and explain why dynamic scheduling might be preferred for certain types of loops.**

   Answer: The scheduling choice in OpenMP, significantly impacts the performance of a parallel for loop by determining how iterations are assigned to threads, affecting load balance and overhead. The static scheduling type (schedule(static, chunk_size)) assigns fixed chunks of iterations (approximately n/thread_count or a specified size) to threads before execution, minimizing scheduling overhead but risking load imbalance if iterations vary in execution time. For example, if some iterations are computationally intensive, threads assigned those iterations may take longer, leaving others idle. The dynamic scheduling type (schedule(dynamic, chunk_size)) addresses this by assigning chunks (default size 1) to threads as they become available, ensuring that threads finishing early can take on more work. This is particularly effective for loops where iteration times differ significantly, as it balances workloads, reducing idle time and improving overall execution time. The guided scheduling type (schedule(guided, chunk_size)) similarly assigns chunks dynamically but with decreasing sizes, starting larger and shrinking, which can reduce scheduling overhead compared to dynamic while still addressing imbalance. The document emphasizes that dynamic scheduling is preferred for loops with unpredictable or varying iteration times (e.g., loops involving complex computations or data-dependent operations) because it ensures threads finish at roughly the same time, maximizing parallelism. However, dynamic scheduling incurs higher overhead due to runtime assignment, so it's less ideal for uniform loops where static would suffice. Choosing dynamic thus optimizes performance for imbalanced loops at the cost of increased scheduling complexity.

7. producers and consumers,

The Producers-Consumers problem is a classic synchronization challenge in parallel programming, including OpenMP. It involves:

1. **Producers:** Threads generating data and adding it to a shared buffer.
2. **Consumers:** Threads removing data from the buffer for processing.

The problem requires synchronization to ensure:

1. **Buffer management:** Coordinating access to the shared buffer.
2. **Data consistency**: Ensuring data is not overwritten or read simultaneously.

OpenMP provides synchronization mechanisms, such as:

1. **Locks:** Protecting critical sections.
2. **Atomic operations**: Ensuring thread-safe updates.

By using these mechanisms, developers can implement efficient and correct producer-consumer patterns in OpenMP, enabling concurrent data production and consumption.

1. **What is the main characteristic of a producer-consumer problem?**

Answer: A producer-consumer problem involves one or more threads generating data (producers) and one or more threads processing that data (consumers), requiring careful synchronization.

```
for (phase = 0; phase < n; phase++) {
if (phase % 2 == 0) {
            #pragma omp parallel for num_threads(thread_count) \
        default(none) shared(a, n) private(i, tmp)
         for (i = 1; i < n; i += 2) {
        if (a[i - 1] > a[i]) {
         tmp = a[i];
         a[i] = a[i - 1];
           a[i - 1] = tmp;
                  }
              }
            } else {
               #pragma omp parallel for num_threads(thread_count) \
         default(none) shared(a, n) private(i, tmp)
         for (i = 1; i < n - 1; i += 2) {
          if (a[i] > a[i + 1]) {
         tmp = a[i + 1];
                  a[i  +  1]  =  a[i];
         a[i] = tmp;
                  }
               }
            }
         }
```

2. **Explain the structure and operation of the producer-consumer program ,focusing on how threads are organized and synchronized to manage the shared buffer.**

   Answer: Program implements a producer-consumer system using OpenMP, where half the threads are producers generating data and half are consumers processing that data, sharing a fixed-size array buffer of size BUFF_SIZE. The program begins by setting the total number of threads to 2 * thread_count using omp_set_num_threads(2 * thread_count), where thread_count is a command-line input, ensuring equal numbers of producer and consumer threads (e.g., 2 producers and 2 consumers for thread_count = 2). Each thread executes the Prod_cons function, with its role determined by its rank (my_rank from omp_get_thread_num()): threads with my_rank < thread_count are producers, and those with my_rank >= thread_count are consumers. Producers generate random floating-point numbers and write them to buffer[i] when flag[i] == 0 (indicating an empty slot), setting flag[i] = 1 to mark the data as valid. Consumers read from buffer[i] when flag[i] == 1, process the data (e.g., by printing), and set flag[i] = 0 to mark the slot as empty. Both operations occur within a critical section (#pragma omp critical) to ensure exclusive access to buffer[i] and flag[i], preventing race conditions. The threads iterate ITERATIONS times over the buffer, with the critical section ensuring synchronized access to each position. This setup allows concurrent production and consumption while maintaining data integrity through mutual exclusion.

3. **Discuss the synchronization mechanism used in Program to handle the producerconsumer problem, including the role of the flag array and the critical directive in preventing race conditions.**

   Answer: In Program , the producer-consumer problem is synchronized using the #pragma omp critical directive and a flag array to manage access to the shared buffer array. The buffer array of size BUFF_SIZE holds floating-point numbers, and the flag array, also of size BUFF_SIZE, tracks whether each position contains valid data (flag[i] = 1) or is empty (flag[i] = 0). Producers (threads with rank < thread_count) generate random numbers and attempt to write to buffer[i] when flag[i] == 0, setting flag[i] = 1 after writing. Consumers (threads with rank >= thread_count) read from buffer[i] when flag[i] == 1, process the data, and set flag[i] = 0. Both operations are enclosed in a critical section (#pragma omp critical), ensuring that only one thread (producer or consumer) accesses a given buffer[i] and flag[i] at a time. This prevents race conditions, such as a producer overwriting a position before a consumer reads it or multiple threads accessing the same position simultaneously, which could corrupt data. The document notes that without the critical section, concurrent updates to buffer or flag could lead to incorrect results (similar to race conditions in Section 5.2). The flag array ensures proper coordination by signaling when a position is ready for writing or reading, while the critical section enforces mutual exclusion, maintaining data integrity but potentially serializing access, which may limit parallelism if contention is high.

4. **Analyze the design of the producer-consumer program in terms of thread roles, synchronization challenges, and the implications of using a critical section for performance.**

Answer: Program designs a producer-consumer system where threads are evenly split into producers and consumers, with synchronization managed via a critical section and a flag array. The program sets 2 * thread_count threads using omp_set_num_threads(2 * thread_count), assigning threads with rank < thread_count as producers and those with rank >= thread_count as consumers. Producers generate random floating-point numbers and write to buffer[i] when flag[i] == 0, setting flag[i] = 1, while consumers read from buffer[i] when flag[i] = 1, resetting flag[i] = 0, iterating ITERATIONS times over BUFF_SIZE positions. The flag array coordinates access by indicating whether a buffer position is empty or contains valid data. Synchronization is achieved using #pragma omp critical, which ensures exclusive access to buffer[i] and flag[i], preventing race conditions (e.g., a producer overwriting unread data or multiple threads accessing the same position). However, the critical section serializes access to the buffer, creating a potential performance bottleneck, especially with many threads or a small BUFF_SIZE, as threads may wait for access, reducing parallelism. The document implies that this serialization could be mitigated with alternative synchronization mechanisms (e.g., locks or atomic operations,), but the critical section ensures simplicity and correctness. The design balances thread role clarity and data integrity but may scale poorly if contention for the critical section is high, suggesting that performance could be improved by increasing BUFF_SIZE or using finer-grained synchronization for larger systems.

8. Caches,

In shared-memory systems, caches can cause performance issues in parallel programs due to:

1. False Sharing: Multiple threads accessing different variables within the same cache line, causing unnecessary cache coherence traffic.

2. Cache coherence overhead: Ensuring data consistency across multiple caches can lead to performance degradation.

False sharing occurs when:

- Multiple threads access nearby data
- Cache lines are invalidated and reloaded unnecessarily

This can significantly impact parallel program performance. Techniques like:

- Data alignment and padding
- Cache-aware data structures
- Minimizing shared variables can help mitigate false sharing and improve performance.

1. **Explain the cache coherence problem in shared-memory systems, including how it arises and its impact on parallel programs.**

   Answer: The cache coherence problem in shared-memory systems, arises because each core in a multicore processor has its own cache, which stores copies of data from main memory to reduce access latency. When multiple cores cache the same memory location, a write by one core updates its local cache but not the caches of other cores or main memory immediately. As

   a result, threads running on different cores may see different values for the same variable, leading to inconsistent program behavior. For example, if thread 0 on core 0 writes to a shared variable, core 1's cache may retain an outdated copy, causing thread 1 to read incorrect data. This inconsistency can cause errors in parallel programs, especially those relying on shared variables for coordination or computation. The document notes that hardware implements cache coherence protocols (not detailed) to ensure consistency, but these protocols add overhead. In shared-memory programming with OpenMP, the cache coherence problem complicates the use of shared variables, requiring careful synchronization (e.g., critical sections or locks) to ensure updates are propagated correctly, impacting performance due to the time needed to synchronize caches across cores.

2. **Describe the false sharing problem in the context of Program, and explain how it affects performance and how it can be mitigated.**

   Answer: In Program , a parallel trapezoidal rule implementation, false sharing occurs when multiple threads update elements of the shared array sums, where each thread t writes to sums[t]. False sharing arises because cache lines (typically 64 bytes) may contain multiple array elements (e.g., 16 floats at 4 bytes each).

```
#pragma omp parallel num_threads(thread_count) \
default(none) shared(a, n) private(i, tmp, phase) for
(phase = 0; phase < n; phase++) {
if (phase % 2 == 0) {
#pragma omp for
for (i = 1; i < n; i += 2) {
if (a[i - 1] > a[i]) {
tmp = a[i - 1];
a[i - 1] = a[i];
a[i] = tmp;
        }
      }
   } else {
      #pragma omp for
for (i = 1; i < n - 1; i += 2) {
if (a[i] > a[i + 1]) {
tmp = a[i + 1];
a[i + 1] = a[i];
            a[i] = tmp;
        }
      }
   }
}
```

When threads on different cores update their respective sums[t] elements, which are likely in the same cache line, a write by one thread invalidates the entire cache line in other cores' caches. This forces those cores to reload the cache line from main memory, even though the threads are accessing different variables (sums[0], sums[1], etc.). The document explains that this repeated invalidation and reloading significantly degrades performance due to increased memory access latency. For example, if thread 0 updates sums[0] and thread 1 updates sums[1] in the same cache line, each update invalidates the other's cache, causing frequent memory accesses. To mitigate false sharing, the document suggests using thread-private variables instead of a shared array. For instance, each thread could compute its partial sum in a private variable and combine results using a reduction clause or critical section, avoiding cache line contention. This approach ensures threads operate on independent memory locations, eliminating false sharing and improving performance by reducing cache coherence overhead.

3. **Analyze the impact of caches on the performance of parallel programs in OpenMP, , focusing on both the cache coherence problem and false sharing, and evaluate potential solutions.**

Answer: caches, while designed to reduce memory access latency by storing recently used data, introduce significant challenges in OpenMP parallel programs due to the cache coherence problem and false sharing. The **cache coherence problem** occurs because each core has its own cache, and a write to a shared memory location by one core updates only its cache, leaving other cores' caches with potentially outdated copies. This can cause threads to see inconsistent values for shared variables, leading to incorrect results unless managed by hardware coherence protocols, which add overhead. For example, in a parallel program, a shared variable update requires synchronizing all caches, slowing execution. **False sharing**, as seen in Program , exacerbates performance issues when threads access different variables within the same cache line (e.g., sums[t] in a shared array). Updates to one element invalidate the entire cache line across cores, causing frequent reloads from main memory, significantly increasing latency. In Program 5.5, threads updating sums[t] cause false sharing because multiple elements reside in one cache line, leading to performance degradation. To mitigate these issues, the document suggests using **thread-private variables** to avoid false sharing, as seen in replacing sums[t] with private variables combined via a reduction clause. For cache coherence, careful use of synchronization mechanisms like critical sections or atomic operations (not detailed in 5.9) can ensure consistent updates, though at the cost of serialization. These solutions improve performance by minimizing cache conflicts but require programmers to carefully design data access patterns, balancing parallelism with synchronization overhead in OpenMP programs.

9. cache coherence and false sharing in openmp,

In OpenMP, cache coherence and false sharing are crucial considerations for performance optimization. Cache Coherence:

Ensures that changes to shared data are visible to all threads, maintaining data consistency across multiple cores.

False Sharing:

Occurs when multiple threads access different variables within the same cache line, causing unnecessary cache coherence traffic and performance degradation.

To mitigate false sharing in OpenMP:

1. Use data alignment and padding to avoid shared variables in the same cache line.
2. Minimize shared variables and use thread-private data when possible.
3. Optimize data structures and access patterns to reduce cache coherence overhead.

By understanding and addressing cache coherence and false sharing, developers can write more efficient and scalable OpenMP parallel programs.

1. **Explain the cache coherence problem in shared-memory systems, including its causes, effects on OpenMP programs, and the role of hardware in addressing it.**

Answer: The cache coherence problem, arises in shared-memory systems because each core has its own cache, which stores copies of main memory data to reduce access latency. When multiple cores cache the same memory location, a write by one core updates its local cache but not other cores' caches or main memory immediately. This results in threads on different cores seeing different values for the same variable, leading to potential inconsistencies in OpenMP programs. For example, if thread 0 on core 0 updates a shared variable, thread 1 on core 1 may read an outdated value from its cache, causing incorrect computations or coordination errors. This is particularly problematic in OpenMP, where shared variables (e.g., global_result in earlier sections) are common. The document notes that hardware implements cache coherence protocols to maintain consistency, such as invalidating or updating other caches when a write occurs, but these protocols introduce overhead, slowing program execution. In OpenMP programs, developers must use synchronization mechanisms like critical sections or locks to ensure consistent updates, further impacting performance due to the time needed to synchronize caches. The cache coherence problem thus complicates parallel programming, requiring careful management of shared data to ensure correctness and efficiency.

2. **Describe the false sharing problem in Program ,including its mechanism, performance impact, and the proposed solution to mitigate it.**

   Answer: In Program , a parallel trapezoidal rule implementation, false sharing occurs when threads update elements of the shared array sums, where each thread t writes its partial sum to sums[t]. As, false sharing arises because a cache line, typically 64 bytes, may contain multiple array elements (e.g., 16 floats at 4 bytes each). When threads on different cores update their respective sums[t] elements within the same cache line, a write by one thread (e.g., thread 0 updating sums[0]) invalidates the entire cache line in other cores' caches, forcing them to reload it from main memory. This happens even though threads access different variables, as the cache operates at the granularity of cache lines. For instance, thread 1 updating sums[1] in the same cache line as sums[0] triggers invalidation, causing performance degradation due to repeated memory accesses. The document highlights that this frequent invalidation and reloading significantly increases latency, slowing the program. To mitigate false sharing, Section 5.9 suggests using thread-private variables instead of a shared array. For example, each thread could compute its partial sum in a private variable and combine results using a reduction clause or critical section, ensuring threads operate on distinct memory locations not sharing cache lines. This solution eliminates cache line contention, improving performance by reducing unnecessary memory operations.

3. **Analyze the interplay between cache coherence and false sharing in OpenMP programs, and evaluate the effectiveness of using thread-private variables as a solution for false sharing in the context of Program .**

   Answer: cache coherence and false sharing are critical challenges in OpenMP programs due to the use of caches in shared-memory systems. The **cache coherence problem** occurs when multiple cores cache the same memory location, and a write by one core does not immediately propagate to other caches, causing threads to see inconsistent values. This requires hardware coherence protocols to synchronize caches, adding overhead that impacts performance, especially in OpenMP programs with shared variables. **False sharing**, as seen in Program , exacerbates this by occurring when threads access different variables within the same cache line, such as sums[t] in a shared array. Updates to one element (e.g., sums[0]) invalidate the entire cache line across cores, forcing reloads even for unrelated variables (e.g., sums[1]), increasing latency. In Program , false sharing in sums causes significant performance degradation due to frequent invalidations. The document proposes using **thread-private variables** to mitigate false sharing, such as computing partial sums in private variables and combining them via a reduction clause. This is highly effective for Program , as it ensures threads operate on separate memory locations, eliminating cache line conflicts and reducing latency. However, this solution does not address cache coherence for inherently shared variables, which still require synchronization. While thread-private variables solve false sharing, developers must also manage cache coherence through synchronization mechanisms, balancing parallelism with overhead. For Program , private variables are an elegant solution, significantly improving performance by avoiding false sharing, but their applicability depends on the program's data access patterns.

10. tasking,

The task directive in OpenMP enables parallelization of irregular and dynamic workloads by:

1. Creating tasks: Dividing work into smaller, independent tasks that can be executed by threads.
2. Task parallelism: Allowing threads to execute tasks concurrently, improving parallelism and flexibility.

Key aspects:

1. Task creation: The task directive creates a new task.
2. Task execution: Tasks are executed by threads in the team.
3. Task synchronization: OpenMP provides synchronization mechanisms, such as taskwait and taskgroup, to manage task dependencies.

The task directive is useful for parallelizing:

1. Recursive algorithms
2. Dynamic data structures 3. Irregular workloads

By leveraging tasks, developers can write more efficient and scalable parallel programs in OpenMP.

1. **What is the role of the single directive in the context of tasking in Program?**

Answer: The single directive ensures that only one thread executes the block that generates tasks, preventing multiple threads from creating redundant tasks.

```
int fib(int n) {
int i = 0;
int j = 0;
    if (n <= 1) {
fibs[n] = n;
return n;
    }
    #pragma  omp  task  shared(i)
i = fib(n - 1);
     #pragma omp task shared(j)     j = fib(n - 2);
     #pragma omp taskwait
    fibs[n] = i + j;
     return fibs[n];
}
```

2. **Explain how the task and taskwait directives are used in Program to parallelize the Fibonacci number computation, including the role of variable scoping.**

Answer: In Program 5.6, the task and taskwait directives are used to parallelize the recursive computation of Fibonacci numbers, addressing the challenge of parallelizing a recursive algorithm. The serial fib function computes the n-th Fibonacci number by recursively calling fib(n-1) and fib(n-2), storing results in a global array fibs. To parallelize this, the program uses #pragma omp parallel to create a team of threads and #pragma omp single to ensure only one thread executes the block that generates tasks, preventing redundant task creation. Within the fib function, each recursive call is wrapped in a task directive: #pragma omp task shared(i) for fib(n-1) and #pragma omp task shared(j) for fib(n-2). These directives create tasks for computing i = fib(n-1) and j = fib(n-2), which the OpenMP run-time schedules across available threads. The shared(i) and shared(j) clauses are critical because, without them, i and j would default to private scope, losing their values after task completion, resulting in fibs[n] = 0 + 0. The taskwait directive (#pragma omp taskwait) ensures that the parent thread waits for both tasks to complete before executing fibs[n] = i + j, guaranteeing correct results. The if(n>20) clause limits task creation for small n to reduce overhead. This approach enables parallel execution of recursive calls while ensuring data integrity through proper scoping and synchronization.

3. **Describe the synchronization mechanism provided by the taskwait directive in Program , and analyze its importance in ensuring correct results for the Fibonacci computation.** **Answer**: In Program , the taskwait directive is crucial for synchronizing the parallel computation of Fibonacci numbers. The program uses the task directive to generate tasks for computing fib(n1) and fib(n-2) within the fib function, allowing these recursive calls to execute in parallel across available threads. However, because task execution order is nondeterministic, the parent thread might execute fibs[n] = i + j before the tasks computing i and j complete, leading to incorrect results (e.g., using uninitialized or outdated values). The #pragma omp taskwait directive addresses this by acting as a barrier, forcing the parent thread to wait until all its subtasks (i.e., the tasks for fib(n-1) and fib(n-2)) are completed before proceeding. This ensures that i

and j contain the correct Fibonacci values before they are summed and stored in fibs[n]. The document emphasizes that without taskwait, the nondeterministic scheduling of tasks could cause race conditions, as the parent thread might access i and j prematurely. By enforcing synchronization, taskwait guarantees the correctness of the Fibonacci computation, making it essential for the recursive parallel algorithm. However, this synchronization introduces some overhead, as the parent thread may idle while waiting, but it is necessary to maintain the integrity of the recursive dependency structure.

4. **Analyze the role of tasking in OpenMP for parallelizing recursive algorithms, as illustrated in Program , and evaluate the impact of the if(n>20) clause on performance.**

Answer: OpenMP's tasking, introduced in OpenMP 3.0, is ideal for parallelizing recursive algorithms like the Fibonacci computation in Program, where traditional loop-based parallelization is infeasible due to the dynamic nature of recursion. The task directive allows the fib function to create tasks for fib(n-1) and fib(n-2), enabling parallel execution of these recursive calls across a thread team created by #pragma omp parallel. The single directive ensures only one thread generates tasks, avoiding redundant task creation, while the shared(i) and shared(j) clauses ensure that the computed values are accessible to the parent thread for summing in fibs[n] = i + j. The taskwait directive synchronizes the parent thread, ensuring tasks complete before the sum is computed. The if(n>20) clause in the task directives (#pragma omp task shared(i) if(n>20)) optimizes performance by preventing task creation for small values of n, where the overhead of task creation and scheduling outweighs the benefits of parallelism. For n≤20, the computation is executed sequentially, reducing run-time system overhead. The document notes that this clause is critical because recursive task creation for small n n n generates excessive tasks (e.g., thousands for Fibonacci), leading to significant overhead. By limiting task creation, the if(n>20) clause improves performance, making the program more efficient while maintaining correctness, though the optimal threshold (20) may depend on the system and workload, requiring tuning for different environments.

5. **thread safety.**

Thread safety in OpenMP ensures that parallel programs produce correct results when multiple threads access shared data. Key considerations:

a Data sharing: Understanding how data is shared among threads (shared, private, etc.).

b Synchronization: Using mechanisms like locks, critical sections, and atomic operations to protect shared data.

c Avoiding data races: Ensuring that concurrent access to shared data does not lead to incorrect results.

Best practices:

1. Use synchronization mechanisms judiciously.
2. Minimize shared data.
3. Use thread-private data when possible.

By ensuring thread safety, developers can write correct and efficient parallel programs in OpenMP.

1. **How is the thread safety issue in the Odd_even function resolved in Program ?**

   Answer: The issue is resolved by declaring factor as threadprivate using #pragma omp threadprivate(factor).

```
void Tokenize(
   char* lines[],    /* in/out */
int line_count,   /* in */
 int thread_count  /* in */
) {      int my_rank,
i, j;
char *my_token;
   #pragma    omp    parallel    num_threads(thread_count)    \
default(none) private(my_rank, i, j, my_token) \
 shared(lines, line_count)
   {
my_rank = omp_get_thread_num();

      #pragma omp for schedule(static, 1)
for (i = 0; i < line_count; i++) {
printf("Thread %d > line %d = %s\n", my_rank, i, lines[i]);
         j = 0;
my_token   =   strtok(lines[i],   "   \t\n");
while    (my_token    !=    NULL)    {
printf("Thread %d > token %d = %s\n",
my_rank, j, my_token);
            my_token  =  strtok(NULL,  "  \t\n");
j++;
         }
      } /* for i */
   } /* omp parallel */
} /* Tokenize */
```

2. **Explain the concept of thread safety in OpenMP, using the Odd_even function in Program to illustrate the problem and its solution.**

Answer: Thread safety, refers to a function's ability to be called simultaneously by multiple threads without causing race conditions or errors. A race condition occurs when multiple threads access a shared resource (e.g., a variable) concurrently, with at least one access being a write, leading to unpredictable results. In Program , the Odd_even function, which sorts an array using odd-even transposition sort, has a thread safety issue due to its use of a static variable factor. The function updates factor to alternate between -1 and 1 to determine comparison directions in each phase of the sort. In a parallel region with #pragma omp parallel for, multiple threads call Odd_even simultaneously, and concurrent updates to the shared factor cause a race condition, potentially corrupting its value and producing incorrect comparisons. The document resolves this by declaring factor as threadprivate with #pragma omp threadprivate(factor). This directive gives each thread its own copy of factor, initialized separately and maintained across parallel regions, ensuring that each thread's updates to factor do not interfere with others. This eliminates the race condition, making Odd_even thread-safe, as each thread operates on its own instance of factor, preserving the correctness of the sorting algorithm in a parallel context.

3.Describe how the threadprivate directive works in OpenMP to address thread safety issues, as illustrated in Program, and discuss its impact on the behavior of static variables in parallel regions.

Answer: The threadprivate directive in OpenMP, addresses thread safety by making a static variable private to each thread, ensuring that each thread has its own copy of the variable, thus preventing race conditions. In Program, the Odd_even function uses a static variable factor to alternate between -1 and 1 for comparison directions in an odd-even transposition sort. Without threadprivate, factor is shared across all threads in a #pragma omp parallel for region, and concurrent updates (e.g., factor = -factor) by multiple threads cause a race condition, leading to unpredictable values and incorrect sorting. By adding #pragma omp threadprivate(factor), each thread gets its own copy of factor, initialized independently (typically to an undefined value unless set explicitly). These copies persist across parallel regions, maintaining their values for each thread, unlike private variables in a parallel directive, which are uninitialized and discarded after the region. In Program, this ensures each thread can update its factor without affecting others, eliminating the race condition. The document notes that this makes Odd_even thread-safe, as each thread's operations on factor are isolated, ensuring correct sorting. The impact is significant: threadprivate enables safe use of static variables in parallel contexts but requires careful initialization and management, as the persistent nature of threadprivate variables can lead to unexpected behavior if not reset properly between parallel regions.

4.Analyze the thread safety issue in the Odd_even function of Program and evaluate the effectiveness of using the threadprivate directive as a solution, including its advantages and potential limitations.

Answer: In Program , the Odd_even function, used in a parallel odd-even transposition sort, is not thread-safe due to its use of a static variable factor, which alternates between -1 and 1 to control comparison directions. When called in a #pragma omp parallel for region, multiple threads access and update factor concurrently, causing a race condition. For example, if thread 0 and thread 1 both execute factor = -factor, the resulting value of factor may be incorrect due to interleaved operations, leading to wrong comparisons and an invalid sort. Section 5.11 resolves this by declaring factor as threadprivate with #pragma omp threadprivate(factor), giving each thread its own copy of factor. This ensures that each thread's updates are isolated, eliminating the race condition and making the function thread-safe. The effectiveness of this solution is high, as it allows parallel execution without altering the algorithm's logic, preserving correctness while leveraging OpenMP's parallelism. Advantages include simplicity (requiring only a single directive) and the preservation of factor's value across parallel regions, which suits the iterative nature of the sort. However, limitations include the need for careful initialization of threadprivate variables, as they are undefined initially, and their persistence, which may cause issues if not reset properly between parallel regions. Additionally, threadprivate is specific to static or global variables, limiting its applicability. Despite these constraints, the document demonstrates that threadprivate is an effective solution for Program 5.7, ensuring thread safety with minimal code changes, though developers must manage variable initialization to avoid subtle bugs.

GPU programming with CUDA - GPUs and GPGPU, GPU architectures, Heterogeneous computing, Threads, blocks, and grids Nvidia compute capabilities and device architectures, Vector addition, Returning results from CUDA kernels, CUDA trapezoidal rule I, CUDA trapezoidal rule II: improving performance, CUDA trapezoidal rule III: blocks with more than one warp.

5 Module

Module -5 Lecturer Notes: GPU Programming with CUDA

This Lecture Notes provides a comprehensive overview of GPU programming with CUDA, based on Chapter 6 of "Peter S Pacheco, Matthew Malensek – An Introduction to Parallel Programming, second edition, Morgan Kauffman , VTU syllabus based BCS702 Parallel Computing" It is designed to serve as a detailed lecture presentation for a classroom setting, covering the evolution of GPUs, their architectures, CUDA programming basics, and practical examples, while ensuring clarity for students new to parallel computing.

MODULE-5: GPU programming with CUDA

1. GPUs and GPGPU,
2. GPU architectures,
3. Heterogeneous computing,
4. Threads, blocks, and grids
5. Nvidia compute capabilities and device architectures,
6. Vector addition,
7. Returning results from CUDA kernels,
8. CUDA trapezoidal rule I,
9. CUDA trapezoidal rule II: improving performance,
10. CUDA trapezoidal rule III: blocks with more than one warp.

1. GPUs and GPGPU,

Introduction to GPUs and GPGPU

Graphics Processing Units (GPUs) were initially developed in the late 1990s and early 2000s to meet the computational demands of realistic video games and animations. Their ability to handle parallel tasks led to their adaptation for general-purpose computing (GPGPU), such as searching and sorting. Early GPGPU programming was challenging, requiring the use of graphics APIs like Direct3D and OpenGL, which complicated algorithm implementation. To address this, Nvidia introduced CUDA, a platform that simplifies GPGPU programming for Nvidia GPUs, while OpenCL offers broader portability across various processors, including FPGAs and DSPs. CUDA is preferred in this context for its simplicity and tight integration with Nvidia hardware.

1. **Explain the evolution of GPU usage from graphics rendering to general-purpose computing, including the challenges faced by early developers.**

Answer: In the late 1990s and early 2000s, GPUs were developed to meet the demand for highly realistic computer video games and animations, focusing on rendering detailed images efficiently. Their computational power attracted programmers outside of graphics, leading to General Purpose computing on GPUs (GPGPU). Early GPGPU developers faced significant challenges because GPUs could only be programmed using graphics APIs like Direct3D and OpenGL. This required reformulating general computational problems, such as searching and sorting, into graphics concepts like vertices, triangles, and pixels, which added considerable complexity to development. To address this, efforts were made to create languages and compilers that allowed programming GPUs using APIs resembling conventional high-level CPU languages. This led to the development of APIs like CUDA, designed for Nvidia GPUs with minimal setup, and OpenCL, designed for portability across various processors, including GPUs, FPGAs, and DSPs.

2. **Compare and contrast CUDA and OpenCL as APIs for general-purpose GPU programming, highlighting their design goals and implications for programmers.**

Answer: CUDA and OpenCL are the two most widely used APIs for general-purpose GPU programming, each with distinct design goals and implications. CUDA, developed by Nvidia, is tailored specifically for Nvidia GPUs, which allows for a simpler setup with less code needed to specify system compatibility or execution details. This focus makes CUDA more straightforward for programmers working exclusively with Nvidia hardware, reducing development complexity. In contrast, OpenCL is designed for high portability, enabling it to run on a variety of processors, including GPUs, FPGAs, and DSPs. To achieve this portability, OpenCL programs require additional code to provide information about compatible systems and execution instructions, increasing complexity compared to CUDA. While CUDA offers ease of use for Nvidia-specific applications, OpenCL's portability makes it suitable for developers targeting diverse hardware platforms, though at the cost of more intricate programming requirements.

3. **Discuss the significance of GPUs in the context of general-purpose computing and the role of APIs like CUDA in overcoming early programming challenges.**

Answer: GPUs, initially developed in the late 1990s and early 2000s to enhance graphics rendering for video games and animations, became significant for general-purpose computing due to their immense computational power. This power tempted programmers to apply GPUs to non-graphics tasks like searching and sorting, giving rise to GPGPU. However, early GPGPU development was hindered by the need to use graphics APIs like Direct3D and OpenGL, which forced programmers to reframe general algorithms in graphics terms, significantly complicating development. The introduction of APIs like CUDA and OpenCL addressed these challenges by providing programming interfaces that resembled conventional high-level CPU languages. CUDA, developed for Nvidia GPUs, simplified programming by requiring minimal setup, making it easier to implement general algorithms on Nvidia hardware.

OpenCL, designed for portability across various processors, allowed broader application but required more setup code. These APIs reduced the complexity of GPGPU programming, enabling programmers to leverage GPU power for diverse computational tasks more effectively.

2. GPU Architectures

GPUs differ significantly from CPUs in their architectural design. CPUs operate on a Single Instruction, Single Data (SISD) model, executing one instruction on one data item at a time. In contrast, GPUs use Single Instruction, Multiple Data (SIMD) or Single Instruction, Multiple Thread (SIMT) architectures, allowing a single instruction to process multiple data items simultaneously. A GPU consists of multiple Streaming Multiprocessors (SMs), each containing several Streaming Processors (SPs). SMs operate asynchronously, enabling efficient parallel execution. The memory hierarchy includes fast shared memory per SM and slower global memory accessible to the entire GPU, which is critical for optimizing performance.

1. **Describe the architectural differences between CPUs and GPUs, focusing on their classification in Flynn's Taxonomy and their operational mechanisms.**

Answer: The architectural differences between CPUs and GPUs, primarily through their classification in Flynn's Taxonomy and their operational mechanisms. CPUs are classified as SISD (Single Instruction stream, Single Data stream) devices, meaning they fetch a single instruction from memory and execute it on a small number of data items, processing one instruction on one data stream at a time. In contrast, GPUs are composed of SIMD (Single Instruction stream, Multiple Data stream) processors, designed to handle multiple data streams with a single instruction. A SIMD processor consists of a single control unit that fetches an instruction and broadcasts it to multiple datapaths, each of which either executes the instruction on its data or remains idle. For example, in a GPU, Nvidia's Streaming Multiprocessors (SMs) contain multiple control units and datapaths (referred to as Streaming Processors or SPs), allowing parallel execution across many data elements. Nvidia uses the term SIMT (Single Instruction Multiple Thread) instead of SIMD, reflecting that threads on an SM executing the same instruction may not be perfectly simultaneous due to memory access latency, where some threads may block while others proceed. This architecture enables GPUs to efficiently handle data-parallel tasks, unlike the sequential processing typical of CPUs.

2. **Explain how a SIMD processor operates within a GPU, using the example to illustrate its functionality.**

Answer: A SIMD (Single Instruction stream, Multiple Data stream) processor in a GPU operates by having a single control unit fetch an instruction from memory and broadcast it to multiple datapaths, which either execute the instruction on their respective data or remain idle. The document illustrates this with an example where an n-element array x is processed by n datapaths, each handling x[i]. The task is to add 1 to nonnegative elements and subtract 2 from negative elements, implemented with the code: if (x[i] >= 0) x[i] += 1; else x[i] -= 2;. In a SIMD system, all datapaths first execute the test x[i] >= 0. Then, datapaths where the condition is true execute x[i] += 1 while others are idle, followed by the reverse where datapaths with x[i] < 0 execute x[i] -= 2 while others are idle, as shown in Table.

Table — Execution of branch on a SIMD system

Time	Datapaths with x[i] ≥ 0	Datapaths with x[i] < 0
1	Test x[i] ≥ 0	Test x[i] ≥ 0
2	x[i] += 1	Idle
3	Idle	x[i] -= 2

In Nvidia GPUs, Streaming Multiprocessors (SMs) consist of multiple such SIMD processors, with each SM having control units and numerous datapaths (SPs). The SMs operate asynchronously, meaning that if threads with x[i] > 0 run on one SM and those with x[i] < 0 on another, the execution can complete in fewer stages (e.g., two stages as shown in Table 6.2), enhancing efficiency for data-parallel tasks.

Table — Execution of branch on a system with multiple SMs

Time	Datapaths with x[i] ≥ 0 (on SM A)	Datapaths with x[i] < 0 (on SM B)
1	**Test x[i] ≥ 0**	**Test x[i] ≥ 0**
2	**x[i] += 1**	**x[i] -= 2**

3 **Discuss the memory architecture of Nvidia GPUs, including the roles of shared and global memory and their implications for performance.**

Answer: The memory architecture of Nvidia GPUs, emphasizing the roles of shared and global memory and their performance implications. Each Streaming Multiprocessor (SM) in an Nvidia GPU has a small block of shared memory accessible only by its Streaming Processors (SPs), which is very fast due to its proximity to the SPs. All SMs on a GPU chip also share access to a much larger block of global memory, which is slower to access. For example, a high-end Nvidia GPU may have 82 SMs, each with 128 SPs, totaling 10,496 SPs, all sharing the global memory but with each SM having its own fast shared memory. The document notes that in earlier systems, data transfer between the CPU (host) and GPU (device) required explicit function calls due to their physical memory separation. However, in Nvidia systems with compute capability ≥ 3.0, unified memory systems reduce the need for explicit transfers, though explicit management can still optimize performance. The fast access to shared memory makes it ideal for frequently accessed data within an SM, while global memory's larger capacity suits data shared across all SPs, but its slower access can bottleneck performance if not managed carefully. This memory hierarchy significantly impacts GPU programming efficiency, requiring careful data placement to optimize speed.

3. Heterogeneous Computing

Heterogeneous computing involves using both CPUs (host) and GPUs (device) within a single program to leverage their respective strengths. GPUs excel at parallel computations, while CPUs handle sequential tasks. This approach is increasingly important due to the stagnation of CPU single-thread performance, pushing developers to utilize GPUs, FPGAs, and Digital Signal Processors (DSPs) for enhanced performance. Heterogeneous computing allows for a division of labor, where the CPU manages program flow and the GPU accelerates data-parallel tasks.

1. **Explain the concept of heterogeneous computing, including how it differs from traditional parallel computing assumptions.**

Answer: Heterogeneous computing, refers to the development of programs that run on systems utilizing processors with different architectures, specifically a conventional CPU (host) and a GPU (device). Unlike traditional parallel computing, which assumes that individual processors have identical architectures, heterogeneous computing involves writing a single program using
the SPMD (Single-Program Multiple-Data) approach, where distinct functions are written for the CPU and GPU. This effectively results in two programs within one: one for the CPU and one for the GPU. The document highlights that this approach is necessary because CPUs and GPUs have fundamentally different architectures, with CPUs typically being SISD (Single Instruction stream, Single Data stream) and GPUs being SIMD (Single Instruction stream, Multiple Data stream) or SIMT (Single Instruction Multiple Thread). This architectural diversity allows heterogeneous computing to leverage the strengths of both processor types, such as the GPU's parallel processing capabilities for data-intensive tasks and the CPU's versatility for sequential tasks.

2. **Discuss the reasons for the increased importance of heterogeneous computing in recent years, and the role of alternative processors like GPUs, FPGAs, and DSPs.**

Answer: The heterogeneous computing has gained importance due to a significant slowdown in single-thread CPU performance improvements. From 1986 to 2003, single-thread CPU performance grew by over 50% annually, but since 2003, this growth has declined, reaching less than 4% per year from 2015 to 2017. This stagnation has driven programmers to explore alternative processors to enhance performance, making heterogeneous computing a critical strategy. GPUs, with their SIMD/SIMT architecture, are highlighted as a primary focus due to their ability to handle data-parallel tasks efficiently, as seen in their original design for graphics rendering and their adaptation for general-purpose computing (GPGPU). Additionally, the document mentions Field Programmable Gate Arrays (FPGAs), which offer programmable logic blocks and interconnects configurable before execution, providing flexibility for specific tasks. Digital Signal Processors (DSPs) are also noted for their specialized circuitry designed for signal manipulation, such as compressing and filtering real-world analog signals. These alternative processors expand the scope of heterogeneous computing, allowing programmers to optimize performance by leveraging the unique strengths of each processor type in a single system.

3. **Describe how illustrates the programming approach for heterogeneous computing, including the challenges and implications for developers.**

Answer: The heterogeneous computing as a programming approach that uses the SPMD (Single-Program Multiple-Data) paradigm to create a single program containing separate functions for a CPU (host) and a GPU (device), effectively writing two distinct programs within one. This approach is necessary because CPUs and GPUs have different architectures, with CPUs designed for sequential processing (SISD) and GPUs optimized for parallel processing (SIMD/SIMT). The document notes that this dual-programming requirement presents challenges, as developers must understand and optimize for both architectures, which may involve managing data transfers between the physically separate memories of the CPU and GPU, especially in older systems. In newer Nvidia systems (compute capability $\geq$ 3.0), unified memory reduces the need for explicit data transfers, but developers may still need to manage transfers for performance optimization. The increased importance of heterogeneous computing stems from the slowdown in CPU performance improvements (from over 50% annual growth before 2003 to less than 4% from 2015 to 2017), pushing developers to leverage GPUs, FPGAs, and DSPs. This requires mastering different programming models and APIs, such as CUDA for Nvidia GPUs, to exploit the parallel processing capabilities of these processors, adding complexity but enabling significant performance gains for data-parallel applications.

CUDA Basics and "Hello, World" Program

CUDA is a platform and programming model developed by Nvidia for general-purpose computing on their GPUs. It supports languages like C and C++ and requires a specialized compiler, nvcc, to generate code for both the host (CPU) and device (GPU). A simple "Hello,

World" program demonstrates CUDA’s basic structure:

```
#include <stdio.h>  #include <cuda.h>
/* Device code: runs on GPU */ __global__ void
Hello(void) {    printf("Hello from thread %d!\n",
threadIdx.x);
}
```

```
/* Host code: Runs on CPU */ int
main(int  argc,  char*  argv[])  {
int thread_count;
   thread_count = strtol(argv[1], NULL, 10);
Hello<<<1,                thread_count>>>();
cudaDeviceSynchronize();    return 0;
}
```

In this program, the __global__ keyword denotes a kernel function that runs on the GPU. The main function on the CPU launches the kernel with a specified number of threads, and cudaDeviceSynchronize() ensures the GPU completes execution before the program terminates. This example introduces key CUDA concepts like kernels and host-device synchronization.

4. Threads, Blocks, and Grids

CUDA organizes parallel execution using threads, blocks, and grids. Threads are the smallest execution units, grouped into blocks that run on a single SM. Blocks are organized into grids, allowing for scalable parallelism. Each thread and block is identified by indices (threadIdx, blockIdx), and dimensions are defined by blockDim and gridDim. This structure enables flexible configuration of parallel tasks, ensuring efficient scheduling across the GPU's resources.

1. **What are the fields available in CUDA's built-in variables like threadIdx and gridDim? Answer**: The fields available are x, y, and z, all of which are unsigned integers.

```
#include <stdio.h>
#include <cuda.h>     /* Header file for CUDA */

/* Device code: runs on GPU */ __global__ void
Hello(void) {    printf("Hello from thread %d in
block %d\n",         threadIdx.x, blockIdx.x);
}  /* Hello */

/*  Host  code:  runs  on  CPU  */  int  main(int  argc,  char*
argv[]) {     int blk_ct;       /* Number of thread blocks */
int th_per_blk;    /* Number of threads in each block */

   blk_ct = strtol(argv[1], NULL, 10);
   /* Get number of blocks from command line */
```

```
    th_per_blk = strtol(argv[2], NULL, 10);
    /* Get number of threads per block from command line */

    Hello<<<blk_ct, th_per_blk>>>();
    /* Start blk_ct × th_per_blk threads on GPU */

    cudaDeviceSynchronize();
  /* Wait for GPU to finish */

    return 0;
  }  /* main */
```

2 **Explain the organization of threads, blocks, and grids in CUDA, and describe how this structure is utilized in the modified greetings program to manage parallel execution. Answer**: In CUDA, threads are organized into thread blocks, and thread blocks are grouped into a grid to enable parallel execution on Nvidia GPUs. A thread block is a collection of threads that execute on a single Streaming Multiprocessor (SM), with each thread running on a Streaming Processor (SP). A grid encompasses all thread blocks launched by a kernel call. The document illustrates this with the modified greetings program ,where the kernel call Hello <<<blk_ct, th_per_blk>>>(); specifies blk_ct thread blocks, each with th_per_blk threads, resulting in a total of blk_ct * th_per_blk threads. These parameters are user-defined via command-line arguments, allowing flexible configuration. When the kernel is executed, each thread block is assigned to an SM, and the threads within the block run on the SM's SPs. The program uses threadIdx.x to identify a thread's rank within its block and blockIdx.x to identify the block's rank within the grid, enabling each thread to print a unique message like "Hello from thread %d in block %d\n". This organization ensures efficient parallel execution, with thread blocks operating independently to allow flexible scheduling by the GPU.

3 **Discuss the role of CUDA's built-in variables (threadIdx, blockDim, blockIdx, gridDim) in facilitating thread management, and how they are applied in the context of the greetings program.**

Answer: CUDA's built-in variables—threadIdx, blockDim, blockIdx, and gridDim—are structs initialized in each thread's memory when a kernel starts, each with x, y, and z fields of unsigned integer type, aiding in thread management. threadIdx provides a thread's rank within its thread block, blockDim specifies the dimensions or size of the thread blocks, blockIdx indicates a block's rank within the grid, and gridDim gives the grid's dimensions. In the modified greetings program , threadIdx.x and blockIdx.x are used within the Hello kernel to generate unique thread identifiers. The kernel call Hello <<<blk_ct, th_per_blk>>>(); launches a grid with blk_ct blocks, each containing th_per_blk threads. The printf statement in the kernel uses threadIdx.x and blockIdx.x to output "Hello from thread %d in block %d\n", ensuring each thread's message reflects its position in the block and grid. These variables are particularly useful for applications like graphics or matrix operations, where x, y, and z fields can represent coordinates or indices, though the greetings program uses only the x field for simplicity

4 **Describe how the CUDA kernel call syntax controls the execution of threads and blocks, and analyze its impact on the performance and output of the greetings program. Answer**: The CUDA kernel call syntax, such as Hello <<<blk_ct, th_per_blk>>>();, defines the structure of thread execution by specifying the number of thread blocks (blk_ct) and threads per block (th_per_blk) in the grid. This determines the total number of threads (blk_ct * th_per_blk) launched on the GPU. In the modified greetings program , these values are obtained from command-line arguments, allowing users to customize the grid configuration. For instance, a call like Hello <<<2, thread_count/2>>>(); on a GPU with two SMs would create two thread blocks, each with thread_count/2 threads, potentially utilizing both SMs if thread_count is even. Each block is assigned to an SM, and its threads execute on the SM's SPs. The kernel uses threadIdx.x and blockIdx.x to print unique greetings per thread, reflecting their block and thread indices. The document notes that thread blocks must be independent, enabling the GPU to schedule them sequentially or in parallel without dependencies, enhancing performance. This structure ensures the greetings program produces correct, unique outputs for each thread while leveraging the GPU's parallel processing capabilities, though performance depends on the GPU's SM count and thread distribution.

5. Nvidia Compute Capabilities and Architectures

Nvidia GPUs are classified by compute capabilities (e.g., 1.x, 2.x, up to 8.0), which determine the maximum number of threads, blocks, and memory available. Modern CUDA programs typically require a compute capability of at least 3.0. Different Nvidia architectures, such as Tesla, Fermi, and Kepler, offer varying performance characteristics and features, impacting programming decisions.

1. **Explain the concept of compute capability in Nvidia GPUs and describe how it affects the limits on threads and blocks.**

Answer: The compute capability of an Nvidia GPU is a numerical identifier, formatted as a.b, where a represents the major revision number (e.g., 1, 2, 3, 5, 6, 7, 8) and b denotes the minor revision number (ranging from 0 to 7, depending on the major revision). It indicates the GPU's architectural features and limitations, influencing CUDA programming constraints. The document specifies that CUDA no longer supports devices with compute capability less than 3.0, reflecting the obsolescence of older architectures. For thread and block limits, GPUs with compute capability greater than 1 support a maximum of 1024 threads per block. For compute capability 2.b, a single Streaming Multiprocessor (SM) can handle up to 1536 threads, while for compute capabilities greater than 2, this limit increases to 2048 threads per SM. Additionally, dimensional limits apply: the maximum x- or y-dimension of a block is 1024, and the z-dimension is capped at 64 for compute capability greater than 1. These limits dictate how CUDA programs can structure thread blocks and grids, impacting performance optimization and resource allocation on different GPU models.

2. **Discuss the relationship between Nvidia's GPU microarchitectures and their compute capabilities, including specific examples.**

Answer: The Nvidia's GPU microarchitectures and their corresponding compute capabilities, which define the feature set and performance characteristics of the GPUs. The microarchitectures listed include Ampere (compute capability 8.0), Tesla (1.b), Fermi (2.b), Kepler (3.b), Maxwell (5.b), Pascal (6.b), Volta (7.0), and Turing (7.5), where b represents varying minor revision numbers. Each microarchitecture represents a generational advancement in GPU design, with compute capability reflecting specific hardware capabilities. For instance, Tesla GPUs have compute capabilities like 1.0 or 1.1, indicating early GPGPU designs, while

Ampere's 8.0 signifies a modern, high-performance architecture. The document notes that the term "Tesla" is also used for Nvidia's GPGPU-targeted products, which can cause confusion. The compute capability affects programming constraints, such as thread limits (e.g., 1024 threads per block for compute capability > 1) and grid dimensions (e.g., max x- or y-dimension of 1024). These microarchitecture-compute capability pairings guide developers in optimizing CUDA programs for specific Nvidia GPUs, leveraging architectural improvements for enhanced performance.

3. **Analyze the limitations imposed by Nvidia's compute capabilities on CUDA programming, particularly regarding thread and block configurations, and discuss their implications for developers.**

Answer: Nvidia's compute capabilities, impose specific limitations on CUDA programming, particularly concerning thread and block configurations, which significantly impact program design. Compute capability, expressed as a.b (with major revisions 1, 2, 3, 5, 6, 7, 8 and minor revisions 0–7), defines the GPU's feature set. CUDA no longer supports compute capabilities below 3.0, requiring developers to target newer GPUs. Key limitations include a maximum of 1024 threads per block for compute capability greater than 1, with Streaming Multiprocessors (SMs) supporting up to 1536 threads for compute capability 2.b and 2048 for higher capabilities. Dimensional constraints further restrict block configurations: the x- and y-dimensions are limited to 1024, and the z-dimension to 64 for compute capability greater than 1. These restrictions necessitate careful thread and block organization to maximize GPU utilization. For developers, this means tailoring CUDA kernels to fit within these limits, potentially requiring multiple blocks to handle large datasets or adjusting block sizes for optimal SM occupancy. The document also notes that thread blocks must be independent, ensuring flexible scheduling but requiring synchronization strategies like those introduced in CUDA 9 for multi-block coordination, which are limited to GPUs with compute capability ≥ 6.0. Developers must consult resources like the CUDA C++ Programming Guide for detailed constraints, balancing performance and compatibility across GPU models.

6. Practical Example: Vector Addition

The vector addition example demonstrates a data-parallel task where two vectors x and y are added to produce a result vector z, where z[i] = x[i] + y[i]. The CUDA kernel is:

```
__global__ void Vec_add(
   const float x[], /* in */
const float y[], /* in */
float z[], /* out */     const
int n /* in */
) {         int my_elt = blockDim.x * blockIdx.x +
threadIdx.x;   if (my_elt < n)     z[my_elt] = x[my_elt]
+ y[my_elt];
}
```

The kernel assigns each thread a unique element index using blockDim.x * blockIdx.x + threadIdx.x. The program uses CUDA managed memory (cudaMallocManaged) for simplicity, though explicit memory transfers are required for older systems. The result is verified against a serial CPU implementation using a two-norm difference, demonstrating significant speedups on GPUs compared to CPUs.

1. **Describe the structure and operation of the VecAdd kernel in the vector addition program ,including how threads are organized and how they perform the computation.** **Answer**: The VecAdd kernel in Program is designed to add two arrays of floating-point numbers, x and y, on a GPU, storing the result in a third array, z.

The kernel is executed in parallel by multiple threads organized into blocks and a grid. Each thread computes a single element of the output array z using the formula z[i] = x[i] + y[i], where i is the thread's global index, calculated as i = blockIdx.x * blockDim.x + threadIdx.x.

This formula combines the block index (blockIdx.x), the number of threads per block (blockDim.x), and the thread's index within its block (threadIdx.x) to assign each thread a unique element of the arrays. A conditional check ensures that the thread only performs the addition if i < n, where n is the array length, preventing out-of-bounds access for arrays not perfectly divisible by the block size. The kernel is launched with VecAdd <<<ceil(n/256.0), 256>>> (x, y, z, n);, where each block contains 256 threads, and the number of blocks is calculated as ceil(n/256.0) to ensure enough threads to process all n elements. This organization allows the GPU to parallelize the addition across its Streaming Multiprocessors (SMs), with each thread block assigned to an SM and threads executing on the SM's Streaming Processors (SPs).

```
__global__ void Vec_add(
   const float x[],  /* in  */
const float y[],  /* in  */
float z[],      /* out */
int n) {
    int my_elt = blockDim.x * blockIdx.x + threadIdx.x;
    /* Total threads = blk_ct * th_per_blk may be > n */
if (my_elt < n)
z[my_elt] = x[my_elt] + y[my_elt]; } /* Vec_add */
 int main(int argc, char* argv[]) {    int n, th_per_blk, blk_ct;
   char i_g;  /* Are x and y user input or random? */
float *x, *y, *z, *x_cz, *y_cz, *z_cz;      double
diff_norm;
    /* Get the command line arguments, and set up vectors */
   Get_args(argc, argv, &n, &blk_ct, &th_per_blk, &i_g);
   Allocate_vectors(&x, &y, &z, &x_cz, &y_cz, &z_cz, n);
   Init_vectors(x, y, n, i_g);
    /* Invoke kernel and wait for it to complete */
   Vec_add<<<blk_ct, th_per_blk>>>(x, y, z, n);
   cudaDeviceSynchronize();
    /* Check for correctness */    Serial_vec_add(x, y, cz, n);    diff_norm =
Two_norm(z, cz, n);
printf("Two-norm of difference between host and ");
 printf("device = %e\n", diff_norm);
    /* Free storage and quit */     Free_vectors(x, y, z, cz);     return 0;
} /* main */
```

2 **Explain the memory management process in the vector addition program , detailing the roles of cudaMalloc, cudaMemcpy, and cudaFree.**

Answer: The vector addition program manages memory between the CPU (host) and GPU (device) using CUDA-specific functions to enable efficient computation. The process begins with memory allocation on the GPU using cudaMalloc, which is called three times to allocate device memory for the input arrays x and y, and the output array z. Each call, such as cudaMalloc((void**)&x, n * sizeof(float)), reserves n * sizeof(float) bytes on the GPU for an array of n floating-point numbers, returning a pointer to the allocated device memory. Next, cudaMemcpy is used to transfer data between the host and device. The program calls cudaMemcpy twice to copy the initialized arrays h_x and h_y from the CPU to the GPU arrays x and y, respectively, using the flag cudaMemcpyHostToDevice. After the VecAdd kernel computes the sum, a third cudaMemcpy call transfers the result array z from the GPU back to the CPU array h_z using cudaMemcpyDeviceToHost. Finally, cudaFree is called three times to deallocate the GPU memory for x, y, and z, ensuring no memory leaks. This memory management process is critical for enabling the GPU to access data for parallel computation and return results to the CPU, with explicit transfers necessary due to the separate memory spaces of the CPU and GPU in systems with compute capability less than 3.0.

3 **Analyze how the vector addition program ensures correct and efficient parallel execution on the GPU, focusing on thread organization and boundary checking.**

Answer: The vector addition program ensures correct and efficient parallel execution on the GPU through careful thread organization and boundary checking in the VecAdd kernel. The kernel is launched with the call VecAdd <<<ceil(n/256.0), 256>>> (x, y, z, n);, which organizes threads into blocks of 256 threads each, with the number of blocks calculated as ceil(n/256.0) to ensure enough threads to cover all n elements of the input arrays. This configuration leverages the GPU's parallel architecture, where each thread block is assigned to a Streaming Multiprocessor (SM), and the 256 threads within a block execute on the SM's Streaming Processors (SPs). Each thread computes a single element of the output array z using the formula z[i] = x[i] + y[i], where the global index i is computed as i = blockIdx.x * blockDim.x + threadIdx.x. This ensures each thread processes a unique array element, maximizing parallelism. To prevent out-of-bounds memory access, the kernel includes a boundary check if (i < n), ensuring threads only perform addition for valid indices, which is crucial when n is not perfectly divisible by 256, as extra threads may be launched. This combination of fixed block size, dynamic block count, and boundary checking ensures that the program correctly processes all array elements in parallel while avoiding errors, making efficient use of the GPU’s resources for large-scale vector addition.

7. Returning results from CUDA kernels

CUDA kernels can't directly return values like traditional C++ functions. Instead, CUDA kernels typically:

1. Allocate memory on the GPU
2. Launch the kernel to perform computations
3. Copy results from GPU memory to host (CPU) memory
4. Use the results on the host To return results, you can:

1. Use cudaMemcpy to copy data from device to host
2. Pass pointers to kernel parameters for output
3. Utilize CUDA's built-in support for structures or classes

Proper synchronization (e.g., cudaDeviceSynchronize) ensures kernel completion before accessing results.

1. **Explain why CUDA kernels cannot return values directly and describe the process used in the vector addition program to return results to the CPU.**

   Answer: CUDA kernels, as noted in the document, are defined with a void return type, meaning they cannot directly return values to their calling function on the CPU. Instead, results are returned by writing them to memory locations in the GPU’s global memory, which are then copied back to the CPU. In the vector addition program (Program 6.3), the VecAdd kernel computes the sum of two arrays, x and y, and stores the results in a global memory array z on the GPU, where each thread writes z[i] = x[i] + y[i] for its assigned index i. After the kernel execution, the program uses cudaMemcpy with the cudaMemcpyDeviceToHost flag to copy the contents of the GPU’s z array to the CPU’s h_z array. This is necessary because the CPU and GPU have separate memory spaces, and direct access is not possible. The process involves allocating GPU memory for x, y, and z using cudaMalloc, copying input arrays from CPU to GPU, executing the kernel, copying the result back to the CPU, and finally deallocating GPU memory with cudaFree. This approach ensures that the results computed in parallel on the GPU are accessible to the CPU for further processing or output.

2. **Discuss the memory management and data transfer process in the vector addition program for returning results, including the role of each CUDA function involved.**

Answer: In the vector addition program ,memory management and data transfer are critical for returning results from the CUDA kernel due to the separate memory spaces of the CPU and GPU. The process begins with memory allocation on the GPU using cudaMalloc, which is called three times to allocate memory for the input arrays x and y, and the output array z, each with n * sizeof(float) bytes for n floating-point elements. The input arrays h_x and h_y on the CPU are then copied to the GPU arrays x and y using cudaMemcpy with the cudaMemcpyHostToDevice flag, preparing the data for the VecAdd kernel. The kernel writes the sum of x[i] and y[i] to z[i] in global memory for each thread's assigned index. After kernel execution, the results in z are copied back to the CPU array h_z using cudaMemcpy with the cudaMemcpyDeviceToHost flag, enabling the CPU to access the results. Finally, cudaFree is called to deallocate the GPU memory for x, y, and z, preventing memory leaks. The document emphasizes that this explicit memory management is necessary because CUDA kernels cannot return values directly, and the separate CPU-GPU memory spaces require such transfers, particularly for systems with compute capability less than 3.0, where unified memory is not available.

3. **Analyze the implications of the CUDA kernel's inability to return values directly, and how the vector addition program addresses this limitation to achieve correct output.** **Answer**: The CUDA kernels, such as the VecAdd kernel in Program 6.3, have a void return type, preventing them from directly returning values to the calling CPU function. This limitation stems from the GPU's parallel architecture, where kernels are executed by many threads across multiple Streaming Multiprocessors (SMs), making direct return values impractical. Instead, results must be written to GPU global memory and copied back to the CPU. In the vector addition program, this is addressed by having the VecAdd kernel store the sum of input arrays x and y in the global memory array z, where each thread computes z[i] = x[i] + y[i] for its assigned index i. The program then uses cudaMemcpy with the cudaMemcpyDeviceToHost flag to transfer the contents of z to the CPU array h_z. This process requires careful memory management: cudaMalloc allocates GPU memory for x, y, and z, cudaMemcpy handles data transfers to and from the GPU, and cudaFree deallocates GPU memory after use. The implication for developers is the need to explicitly manage memory and data transfers, adding complexity but enabling parallel computation. The program ensures correct output by structuring the kernel to write results to global memory and using cudaMemcpy to make those results accessible to the CPU, demonstrating an effective workaround for the kernel's inability to return values directly.

8. Practical Example: Trapezoidal Rule

The trapezoidal rule example approximates the integral of a function using CUDA. The optimized version uses shared memory and larger thread blocks (up to 1024 threads) to improve performance:

```
__global__ void Dev_trap(
const float a /* in */,    const
float b /* in */,    const float
h /* in */,    const int n /* in
*/,    float* trap_p /* out */)
{
    __shared__       float       thread_calcs[MAX_BLKSZ];
__shared__ float warp_sum_arr[WARPSZ];
int my_i = blockDim.x * blockIdx.x + threadIdx.x;
int my_warp = threadIdx.x / warpSize;
int my_lane = threadIdx.x % warpSize;
float* shared_vals = thread_calcs + my_warp * warpSize;
float blk_result = 0.0;
 shared_vals[my_lane] = 0.0f;
 if (0 < my_i && my_i < n) {
float my_x = a + my_i * h;
 shared_vals[my_lane] = f(my_x);
    }
    float my_result = Shared_mem_sum(shared_vals);
if (my_lane == 0)
warp_sum_arr[my_warp] = my_result;    __syncthreads();

    if (my_warp == 0) {
if   (threadIdx.x   >=   blockDim.x   /   warpSize)
warp_sum_arr[threadIdx.x] = 0.0;
blk_result = Shared_mem_sum(warp_sum_arr);
    }
    if (threadIdx.x == 0) atomicAdd(trap_p, blk_result);
}
```

This kernel uses shared memory (thread_calcs, warp_sum_arr) to store intermediate results, reducing global memory access. It employs a tree-structured sum within warps and across warps, synchronized with __syncthreads(), and uses atomicAdd to update the final result safely. Optimizations like warp shuffles and shared memory yield significant speedups on modern GPUs, such as the Nvidia Titan X.

9. CUDA trapezoidal rule I

The CUDA trapezoidal rule involves parallelizing numerical integration using NVIDIA's CUDA architecture. Key aspects:

1. Divide the integration interval into subintervals (trapezoids)
2. Assign each subinterval to a CUDA thread for parallel computation
3. Calculate the area of each trapezoid using the function values at the endpoints
4. Sum the areas using parallel reduction or atomic operations

By leveraging CUDA's parallel processing capabilities, the trapezoidal rule can be efficiently implemented to approximate definite integrals, achieving significant speedups compared to serial implementations.

1. **Explain how the Trap kernel in the CUDA trapezoidal rule program computes the integral approximation, including the role of thread organization and the trapezoidal rule formula.**

Answer: The Trap kernel in Program approximates the integral of a function f(x) over the interval [a,b] using the trapezoidal rule by dividing the interval into n trapezoids and computing their areas in parallel on the GPU.

```
void Serial_vec_add(
   const  float  x[],      /*  in    */,
const float y[],   /* in  */,
 float cz[],       /* out */,
const int n        /* in  */) {

   for  (int  i  =  0;  i  <  n;  i++)
cz[i] = x[i] + y[i];
} /* Serial_vec_add */
```

The interval is divided into n equal subintervals of width h=(b−a)/n. Each thread in the kernel is responsible for computing the area of one trapezoid, identified by its global index i=blockIdx.x∗blockDim.x+threadIdx.x

The thread calculates the left endpoint of its trapezoid as x=a+i∗h and computes the area using the formula area=h∗(f(x)+f(x+h))/2, where f(x) and f(x+h) are the function values at the trapezoid's endpoints. This area is stored in the global memory array z at index i, provided i<n to prevent out-of-bounds access. The kernel is launched with Trap <<<ceil(n/256.0), 256>>> (a, b, z, n);, creating enough blocks of 256 threads each to cover all n trapezoids. After the kernel execution, the partial areas in z are copied to the CPU array h_z using cudaMemcpy, and the Sum_array function sums these areas to compute the integral approximation, leveraging the GPU's parallel processing to efficiently handle large n.

2. **Describe the memory management and data transfer process in the CUDA trapezoidal rule program , detailing how results are computed and returned to the CPU.**

Answer: The CUDA trapezoidal rule program manages memory and data transfers between the CPU and GPU to compute the integral approximation. The process begins with allocating memory on the GPU for the array z, which stores the area of each trapezoid, using cudaMalloc((void**)&z, n * sizeof(float)) to reserve n * sizeof(float) bytes. Unlike the vector addition program, no input arrays are copied to the GPU, as the kernel directly computes function values using the parameters a , b , and n , passed to the Trap kernel. The kernel, launched as Trap <<<ceil(n/256.0), 256>>> (a, b, z, n);, has each thread compute the area of one trapezoid and store it in z[i] in global memory. After execution, the results in z are copied to the CPU array h_z using cudaMemcpy(h_z, z, n * sizeof(float), cudaMemcpyDeviceToHost), transferring the partial areas to the host. The Sum_array function then sums the elements of h_z on the CPU to compute the final integral approximation. Finally, cudaFree(z) deallocates the GPU memory to prevent leaks. This process is necessary due to the separate CPU and GPU memory spaces and the kernel's inability to return values directly, ensuring that the parallel-computed trapezoid areas are accessible to the CPU for summation.

3. **Analyze how the CUDA trapezoidal rule program ensures efficient and correct computation of the integral, focusing on thread organization, boundary checking, and result aggregation.**

Answer: The CUDA trapezoidal rule program ensures efficient and correct computation of the integral of f(x) over [a,b] by leveraging GPU parallelism, careful thread organization, boundary checking, and result aggregation. The interval [a,b] is divided into n trapezoids, and the Trap kernel is launched with Trap <<<ceil(n/256.0), 256>>> (a, b, z, n);, creating blocks of 256 threads each, with the number of blocks calculated to cover all n trapezoids. Each thread computes the area of one trapezoid using its global index $i=blockIdx.x*blockDim.x+threadIdx.x$, calculating the area as $h*(f(x)+f(x+h))/2$, where $h=(b-a)/n$ and $x=a+i*h$. A boundary check if (i < n) ensures threads only write to valid indices in the global memory array z, preventing out-of-bounds errors when n is not divisible by 256. The use of 256 threads per block optimizes GPU resource utilization, as each block runs on a Streaming Multiprocessor (SM). After kernel execution, the partial areas in z are copied to the CPU array h_z using cudaMemcpy, and the Sum_array function aggregates these areas to compute the final integral. This approach ensures correctness by assigning each trapezoid to a unique thread and efficiency by parallelizing the area calculations across the GPU, with the CPU handling the final sequential summation due to the kernel's inability to return values directly.

10 CUDA trapezoidal rule II: improving performance

1. **Explain how the Trap kernel in Program improves performance over the previous trapezoidal rule implementation, focusing on the use of shared memory and thread workload distribution.**

Answer: The Trap kernel in Program improves performance over the previous implementation (Program) by reducing global memory writes and distributing the workload more efficiently. void Serial_vec_add(

```
 const  float  x[]     /*  in  */,
const float y[]   /* in */,
 float cz[]        /* out */,
const int n       /* in */) {
     for (int i = 0; i < n; i++)
cz[i] = x[i] + y[i];
} /* Serial_vec_add */
```

In the earlier version, each thread computed the area of one trapezoid and wrote it to global memory, resulting in n n n global memory writes for n trapezoids, which is slow due to global memory's high latency. In Program , each thread computes the areas of multiple trapezoids, determined by trap_per_thread = ceil(n / (blk_ct * th_per_blk)), where blk_ct is the number of blocks and th_per_blk is the number of threads per block. Each thread accumulates a partial sum of its trapezoid areas in a local register variable sum. These partial sums are then stored in a shared memory array smem within each block, which is much faster than global memory. After computing their partial sums, threads synchronize using __syncthreads(), and the thread with threadIdx.x == 0 sums the values in smem and writes a single block partial sum to the global memory array z at index blockIdx.x. This reduces the number of global memory writes to the number of blocks (blk_ct), significantly improving performance, especially for large n n n, as shared memory access is faster and fewer global memory operations are required.

2.. **Describe the memory management and synchronization process in the CUDA trapezoidal rule program , detailing how partial sums are computed, stored, and aggregated.** **Answer**: The CUDA trapezoidal rule program (Program) employs a sophisticated memory management and synchronization process to compute the integral of a function over [a,b] using the trapezoidal rule.

```
void Get_args(

   const int   argc           /* in */,
char*     argv[]     /* in */,     int*
n_p              /*  out  */,        int*
blk_ct_p       /*  out  */,         int*
th_per_blk_p/*  out  */,       char*
i_g         /* out */) {     if (argc !=
5) {

      /* Print an error message and exit */
      ...
   }
   *n_p = strtol(argv[1], NULL, 10);

   *blk_ct_p = strtol(argv[2], NULL, 10);

   *th_per_blk_p = strtol(argv[3], NULL, 10);

   *i_g = argv[4][0];

   /* Is n > total thread count = blk_ct * th_per_blk? */

if (*n_p > (*blk_ct_p) * (*th_per_blk_p)) {

      /* Print an error message and exit */

      ...
}

} /* Get_args */
```

Memory management begins with cudaMalloc((void**)&z, blk_ct * sizeof(float)), allocating GPU global memory for the array z to store blk_ct block partial sums, where blk_ct is the number of thread blocks. In the Trap kernel, each thread computes the areas of trap_per_thread trapezoids, calculated as ceil(n / (blk_ct * th_per_blk)), where n n n is the total number of trapezoids and th_per_blk is the number of threads per block. Each thread stores its partial sum in a local variable sum, then writes it to a shared memory array smem at index threadIdx.x. Shared memory is declared as __shared__ float smem[256] (assuming 256 threads per block), providing fast access within each block. After all threads write to smem, the __syncthreads() function ensures synchronization, guaranteeing all partial sums are written before proceeding.

The thread with threadIdx.x == 0 then sums the smem array and writes the block's partial sum to z[blockIdx.x] in global memory. The global memory array z is copied to the CPU array h_z using cudaMemcpy(h_z, z, blk_ct * sizeof(float), cudaMemcpyDeviceToHost), and the Sum_array function aggregates these blk_ct partial sums on the CPU to compute the final integral. Finally, cudaFree(z) deallocates GPU memory. This process minimizes global memory writes and leverages fast shared memory, enhancing performance.

3 **Analyze how the CUDA trapezoidal rule program achieves performance improvements through thread organization, shared memory usage, and synchronization, and discuss the implications for scalability.**

Answer: Program enhances performance over the previous trapezoidal rule implementation by optimizing thread organization, using shared memory, and implementing synchronization. The Trap kernel organizes threads such that each computes the areas of multiple trapezoids, calculated as trap_per_thread = ceil(n / (blk_ct * th_per_blk)), where n is the number of trapezoids, blk_ct is the number of blocks, and th_per_blk is the number of threads per block (e.g., 256). This reduces the total number of threads needed compared to one thread per trapezoid, allowing efficient workload distribution. Each thread computes its partial sum locally and stores it in a shared memory array smem, which is significantly faster than global memory due to its proximity to the Streaming Multiprocessor (SM). The __syncthreads() function synchronizes threads within a block, ensuring all partial sums are written to smem before the thread with threadIdx.x == 0 sums them and writes a single block partial sum to the global memory array z. This reduces global memory writes from n n n (one per trapezoid) to blk_ct (one per block), mitigating the high latency of global memory access. The CPU then uses Sum_array to sum the blk_ct partial sums in h_z after copying from the GPU. For scalability, this approach is effective for large n, as the number of global memory writes scales with the number of blocks, not trapezoids, and shared memory usage minimizes latency. However, the document notes that shared memory size limits the number of threads per block, and developers must ensure blk_ct is sufficient to cover all trapezoids, impacting performance on GPUs with varying SM counts and compute capabilities.

11 CUDA trapezoidal rule III: blocks with more than one warp

In CUDA trapezoidal rule III, blocks with more than one warp are utilized to further optimize parallel computation. Key aspects:

1. Multiple warps per block: Each block consists of multiple warps (typically 32 threads), enabling better GPU utilization.
2. Shared memory: Shared memory is used to reduce global memory access and improve data locality.
3. Parallel reduction: A parallel reduction algorithm is employed to efficiently sum partial results within each block.
4. Block-level synchronization: Threads within a block synchronize to ensure accurate results.

By leveraging multiple warps per block and shared memory, this approach achieves improved performance and scalability for approximating definite integrals using the trapezoidal rule.

1. **Explain the concept of warps in the CUDA trapezoidal rule program and how they influence the design of the Trap kernel with blocks containing more than one warp.** **Answer**: In the CUDA trapezoidal rule program , a warp is defined as a group of 32 threads that execute simultaneously on a Streaming Multiprocessor (SM), as outlined in the document. With a block size of 256 threads, each block contains 8 warps (256 ÷ 32 = 8), allowing multiple warps to execute concurrently within a single SM. The Trap kernel is designed to compute the integral of a function over [a,b][a, b][a,b] using the trapezoidal rule, with each thread calculating the areas of multiple trapezoids (determined by trap_per_thread = ceil(n / (blk_ct * th_per_blk))) and storing a partial sum in a local variable. These partial sums are written to a shared memory array smem at index threadIdx.x. The use of multiple warps requires careful synchronization, achieved through the __syncthreads() function, which ensures all 256 threads in a block have written their partial sums to smem before any further processing. After synchronization, the thread with threadIdx.x == 0 sums all values in smem to produce the block's total sum and writes it to the global memory array z at index blockIdx.x. This design leverages the parallel execution of warps to compute partial sums efficiently while minimizing global memory writes (one per block), enhancing performance by utilizing fast shared memory and ensuring correct summation across multiple warps within each block.

2. **Describe the memory management and synchronization process in the Trap kernel of Program , focusing on the role of shared memory and the handling of multiple warps per block.**
   Answer: The Trap kernel in Program 6.6 employs a sophisticated memory management and synchronization process to compute the integral approximation using the trapezoidal rule, optimized for blocks with multiple warps.

```
void    Allocate_vectors(
float**  x_p    /*  out  */,
float**  y_p    /*  out  */,
float**  z_p    /*  out  */,
float**  cz_p   /*  out  */,
int    n    /* in  */) {

  /* x, y, and z are used on host and device */
cudaMallocManaged(x_p, n * sizeof(float));
cudaMallocManaged(y_p, n * sizeof(float));
cudaMallocManaged(z_p, n * sizeof(float));

/* cz is only used on host */
*cz_p = (float*) malloc(n * sizeof(float));
} /* Allocate_vectors */
```

Memory management begins with cudaMalloc((void**)&z, blk_ct * sizeof(float)) in the main function, allocating global memory for the array z to store one partial sum per block (blk_ct is the number of blocks). In the kernel, each block uses a shared memory array __shared__ float smem[256], sized for 256 threads per block, to store partial sums computed by each thread. Each thread calculates the areas of trap_per_thread trapezoids, accumulating the result in a local variable sum, which is then written to smem[threadIdx.x]. Since each block has 256 threads, organized into 8 warps of 32 threads each, synchronization is critical to ensure all threads complete their writes to smem. The __syncthreads() function is called to synchronize all threads within the block, guaranteeing that the shared memory array is fully populated before summation. The thread with threadIdx.x == 0 then iterates over smem to compute the block's total sum and writes it to z[blockIdx.x] in global memory, reducing global memory writes to one per block. After kernel execution, cudaMemcpy transfers the blk_ct partial sums from z to the CPU array h_z, which are summed by the Sum_array function to compute the final integral. Finally, cudaFree(z) deallocates GPU memory. This process leverages fast shared memory and synchronization to handle multiple warps efficiently, minimizing slow global memory accesses.

3 **Analyze how the CUDA trapezoidal rule program improves performance by using blocks with multiple warps, and discuss the implications of this approach for scalability and efficiency.**

Answer: Program improves performance over previous implementations by using blocks with 256 threads, organized into 8 warps of 32 threads each, to compute the trapezoidal rule approximation efficiently. Each thread computes the areas of multiple trapezoids (trap_per_thread = ceil(n / (blk_ct *

th_per_blk))), accumulating partial sums locally before writing to a shared memory array smem. This approach leverages the parallel execution of warps within a Streaming Multiprocessor (SM), as warps execute simultaneously, maximizing SM utilization. The use of shared memory, declared as __shared__ float smem[256], reduces global memory writes, which are slow due to high latency, to one per block: the thread with threadIdx.x == 0 sums the 256 partial sums in smem after __syncthreads() ensures all threads have written their results, then writes the block's total to z[blockIdx.x]. This results in only blk_ct global memory writes, compared to n n n in earlier versions, significantly enhancing performance for large n n n. The CPU then sums the blk_ct partial sums in h_z using Sum_array. For scalability, this approach is effective as it minimizes global memory access and leverages shared memory's low latency, but it is constrained by shared memory size (limiting threads per block) and the need for sufficient blocks to cover all trapezoids. The document notes that the fixed block size of 256 threads balances warp-level parallelism with
SM resource constraints, making the program efficient for GPUs with multiple SMs, though developers must adjust blk_ct based on n n n and GPU compute capability to optimize performance.

Advanced Topics: Bitonic Sort

Bitonic sort is a parallel sorting algorithm suitable for CUDA due to its reliance on compare-swap operations and butterfly structures. It requires frequent synchronization barriers but can outperform CPU-based sorting for large datasets. The algorithm's independence of thread operations makes it ideal for GPU parallelization, though its complexity requires careful implementation.

Performance Optimization

CUDA programs benefit from several optimization techniques:

- **Memory Hierarchy**: Utilizing registers (fastest), shared memory (block-specific), and global memory (device-wide) appropriately.
- **Warp Shuffles**: Enabling threads within a warp to exchange data efficiently, reducing global memory access (available on compute capability ≥ 3.0).
- **Shared Memory**: Faster than global memory, used for intermediate computations to minimize serialization.
- **Synchronization**: Using __syncthreads() to coordinate threads within a block, ensuring correct execution order.

These techniques are demonstrated in the trapezoidal rule example, where iterative improvements lead to substantial performance gains.

Performance Data

The following table summarizes performance data for the trapezoidal rule implementations, as referenced in the chapter:

This table illustrates how optimizations progressively reduce run-time, achieving significant speedups on modern GPUs.

Implementation	**Device**	**Run-Time (ms)**	**Speedup**
Serial (CPU)	Host	Varies	1.0
Basic CUDA	GPU	Higher	Moderate
AtomicAdd	GPU	Reduced	Improved
Tree-Structured Sum	GPU	Further Reduced	High
Warp Shuffles	GPU (≥3.0)	Lower	Very High
Shared Memory (Program)	GPU	Lowest (e.g., Titan X)	Up to 93%

Summary and Educational Value

The presentation and this report cover the essential aspects of GPU programming with CUDA, from theoretical foundations to practical implementations. The content is designed to be accessible to students new to parallel computing, with clear explanations, code examples, and performance data. The inclusion of practical examples like vector addition and the trapezoidal rule bridges theory and application, while advanced topics like bitonic sort introduce students to complex parallel algorithms. The discussion slide encourages interactive learning, prompting students to explore GPU vs. CPU differences and optimization strategies.

For further learning, students can refer to Nvidia's official CUDA documentation (CUDA Toolkit Documentation) for detailed guides and examples. The book's website, if available, may provide additional source code for the examples discussed.

This report and the accompanying presentation provide a comprehensive resource for teaching GPU programming with CUDA, ensuring students gain a solid understanding of parallel computing concepts and their practical application.

Model Question Paper

Fifth Semester MCA Degree Examination, Dec. 2016 / Jan. 2017

Parallel Computing

Time: 3 hrs **Max. Marks:** 100

Note: Answer any **FIVE full questions.**

1.

a. What is parallel computing? Why we need ever increasing performance? (08 Marks)

b. Why we need to write parallel programs? (05 Marks)

c. If we fetch a cache line from main memory, where in the cache should it be placed? Explain. (07 Marks)

2.

a. Explain the two main types of MIMD systems. (10 Marks)

b. What is cache coherence? Explain two main approaches to ensuring cache coherence. (10 Marks)

3.

a. Describe all the distributed memory interconnects. (10 Marks)

b. Explain Foster's methodology with an example. (10 Marks)

4.

a. Explain serial and parallel trapezoidal rule. (08 Marks)

b. Differentiate between collective communication and point-to-point communication. (04 Marks)

c. What is collective communication? How a global sum is calculated in collective communication? (08 Marks)

5.

a. Estimate the value of π using pthreads. (08 Marks)

b. What are Barriers? Implement barriers using busy-waiting, mutex, and semaphores. (08 Marks)

c. Explain pthreads read–write locks. (04 Marks)

6.

a. Write a 'hello, world' program that uses OpenMP. (05 Marks)

b. Explain reduction clause in OpenMP. (08 Marks)

c. Write a note on data dependence and scope of variable in OpenMP. (07 Marks)

7.

a. Explain multithreaded tokenizer in OpenMP. (10 Marks)

b. Explain different schedule types in OpenMP. (10 Marks)

8.

a. How to parallelize the n-body solver using OpenMP? (10 Marks)

b. Explain both recursive and non-recursive depth-first search to solve the travelling salesman problem. (10 Marks)

Fifth Semester MCA Degree Examination, Dec. 2017 / Jan. 2018

Parallel Computing

Time: 3 hrs **Max. Marks:** 100

Note: Answer any **FIVE full questions.**

1.

a. What is parallel computing? List any two applications that need parallel computing. (05 Marks)

b. Discuss the different types of co-ordinators necessary for a parallel program to be exempted.(05 Marks)

c. Explain Von Neumann architecture with neat diagram. Discuss the various modifications of the Von Neumann architecture with respect to caches. (10 Marks)

2.

a. Discuss Instruction level pipelining in detail. (06 Marks)

b. With a neat diagram, explain UMA and NUMA architecture. (08 Marks)

c. Explain the different types of cache coherence in detail. (06 Marks)

3.

a. What is a distributed memory system? Discuss the various issues related to distributed memory. (10 Marks)

b. Discuss the different techniques used to ensure mutual exclusion to a shared variable in shared memory system. (06 Marks)

c. What is Amdahl's law? Discuss the outcomes of Amdahl's laws. (04 Marks)

4.

a. How does MPI handle message passing between the threads? Give the syntax and discuss the issues to be handled during communication. (05 Marks)

b. Apply Foster's methodology to solve the trapezoidal rule problem using MPI. (05 Marks)

c. Explain MPI_Scatter and MPI_Gather. Write a function to read and distribute a vector using MPI scatter and gather. (10 Marks)

5.

a. Explain the working of thread program with an example. (10 Marks)

b. What is a semaphore? Discuss the various functions associated with semaphores in Pthreads. (10 Marks)

6.

a. Give the differences between Pthreads and OpenMP programming. (04 Marks)

b. Discuss `parallel for` directive in detail. (06 Marks)

c. Write a program using OpenMP to find the sum of 1 + 2 + 3 + … + N. (10 Marks)

7.

a. Explain scheduling in OpenMP. (08 Marks)

b. Explain the concept of producer-consumer problem using OpenMP. Give functions to explain the message passing using OpenMP. (12 Marks)

8.

a. What is an n-body solver problem? Explain how it can be parallelized using Foster's method.
(08 Marks)

b. Discuss how tree search can be solved using dynamic partitioning and MPI. (12 Marks)

COURSE OUTCOMES

Course Outcomes: At the end of this course, students are able to:
CO1 - Explain the need for parallel programming
CO2 - Demonstrate parallelism in MIMD system
CO3 - Apply MPI library to parallelize the code to solve the given problem.
CO4 - Apply OpenMP pragma and directives to parallelize the code to solve the given problem
CO5 - Design a CUDA program for the given problem

COs and POs Mapping of lab Component

COURSE OUTCOMES	PO1	PO2	PO3	PO4	PO5	PO6	PO7	PO8	PO9	PO10	PO11	PO12	PSO1	PSO2	PSO3
CO1	3	2	-	-	-	-	-	-	-	-	-	2	2	-	-
CO2	3	3	2	2	2	-	-	-	-	-	-	1	3	2	-
CO3	3	3	3	-	3	-	-	-	1	-	2	2	3	3	2
CO4	3	3	3	-	3	-	-	-	1	-	2	1	3	3	2
CO5	3	3	3	-	3	-	-	-	1	-		2	3	2	3

CO-PO Mapping Justifications:

- **CO1 -** Understanding the fundamentals of parallel programming equips students to analyze the limitations of sequential processing. It also promotes lifelong learning as they explore the evolution and relevance of parallel computing models.

- **CO2 -** Students gain insight into multi-core architecture by analyzing and implementing MIMD-based parallelism. This helps them apply theoretical knowledge to real-world hardware configurations and improve performance understanding

- **CO3 -** Using MPI, students learn to structure and implement distributed applications that communicate through message passing. It strengthens their ability to design scalable solutions in distributed-memory environments**.**

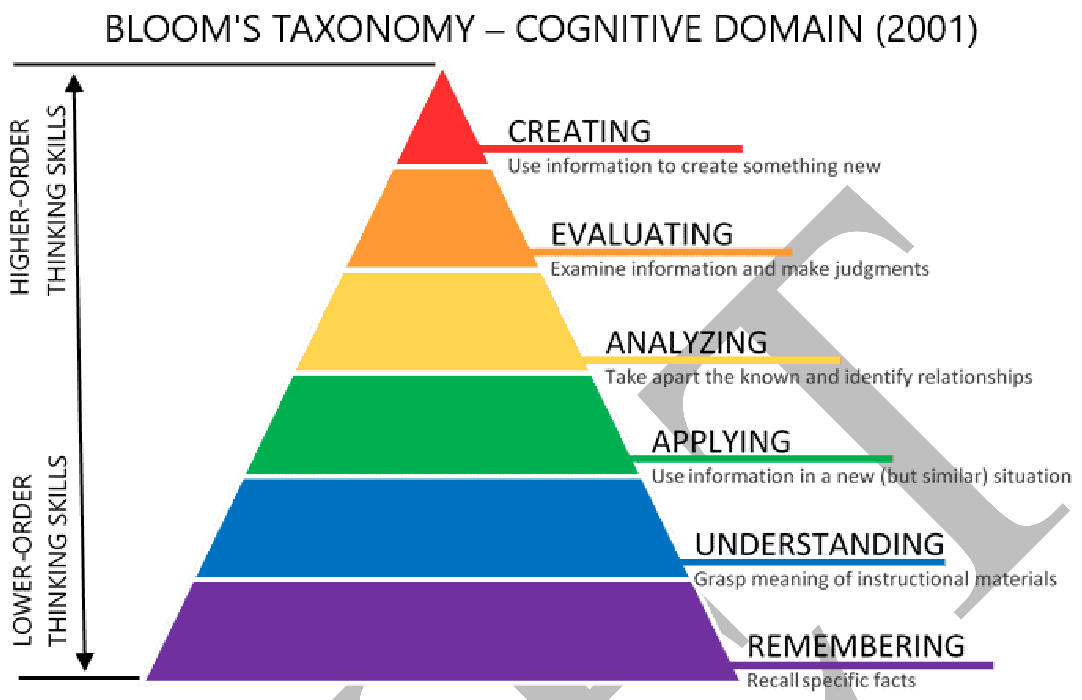
- **CO4 -** OpenMP allows students to parallelize code for shared-memory systems using compiler directives, improving execution speed. This fosters skills in identifying parallel sections and managing synchronization efficiently.

- **CO5 -** Students apply CUDA to solve data-parallel problems on GPUs, learning to optimize memory and thread usage. This enhances their understanding of heterogeneous computing and modern accelerator-based systems.

Mapping of 'Graduate Attributes' (GAs) and 'Program Outcomes' (POs)

Graduate Attributes (GAs) (As per Washington Accord Accreditation)	**Program Outcomes (POs) (As per NBA New Delhi)**
Engineering Knowledge	Apply the knowledge of mathematics, science, engineering fundamentals and an engineering specialization to the solution of complex engineering problems
Problem Analysis	Identify, formulate, review research literature and analyze complex engineering problems reaching substantiated conclusions using first principles of mathematics, natural sciences and engineering sciences.
Design/Development of solutions	Design solutions for complex engineering problems and design system components or processes that meet the specified needs with appropriate considerations for the public health and safety and the cultural, societal and environmental consideration.
Conduct Investigation of complex problems	Use research – based knowledge and research methods including design of experiments, analysis and interpretation of data and synthesis of the information to provide valid conclusions.
Modern Tool Usage	Create, select and apply appropriate techniques, resources and modern engineering and IT tools including prediction and modeling to complex engineering activities with an understanding of the limitations.
The engineer and society	Apply reasoning informed by the contextual knowledge to assess society, health, safety, legal and cultural issues and the consequential responsibilities relevant to the professional engineering practice.
Environment and sustainability	Understand the impact of the professional engineering solutions in societal and environmental context and demonstrate the knowledge of and need for sustainable development.
Ethics	Apply ethical principles and commit to professional ethics and responsibilities and norms of the engineering practice.
Individual and team work	Function effectively as an individual and as a member or leader in diverse teams and in multidisciplinary settings.
Communication	Communicate effectively on complex engineering activities with the engineering community and with society at large, such as being able to comprehend and write effective reports and design documentation, make effective presentations and give and receive clear instructions.
Project management & finance	Demonstrate knowledge and understanding of the engineering and management principles and apply these to ones won work, as a member and leader in a team, to manage projects and in multidisciplinary environments.
Life Long Learning	Recognize the need for and have the preparation and ability to engage in independent and life-long learning in the broadest context of technological change.

REVISED BLOOMS TAXONOMY (RBT)


BLOOM'S TAXONOMY – COGNITIVE DOMAIN (2001)
HIGHER-ORDER THINKING SKILLS
LOWER-ORDER THINKING SKILLS
CREATING
Use information to create something new
EVALUATING
Examine information and make judgments
ANALYZING
Take apart the known and identify relationships
APPLYING
Use information in a new (but similar) situation
UNDERSTANDING
Grasp meaning of instructional materials
REMEMBERING
Recall specific facts

LAB EVALUATION PROCESS

WEEK WISE EVALUATION OF EACH PROGRAM
PART A

SL.NO	ACTIVITY	MARKS
1	Observation Book	10
2	Record and Viva	15+5
	TOTAL	**30**

INTERNAL ASSESSMENT
PART B

SL.NO	ACTIVITY	MARKS
1	Procedure	5
2	Conduction	10
3	Viva -Voce	5
TOTAL		**20**
PART A + PART B		**50**

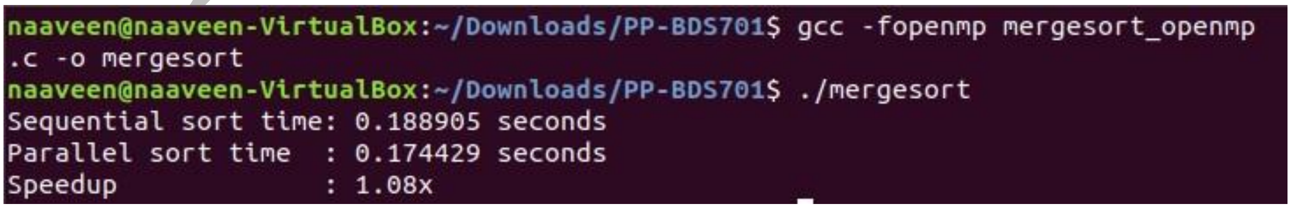
PROGRAM LIST

Sl. NO.	***Program Description***
1	Write a OpenMP program to sort an array on n elements using both sequential and parallel mergesort (using Section). Record the difference in execution time.
2	Write an OpenMP program that divides the Iterations into chunks containing 2 iterations, respectively (OMP_SCHEDULE=static,2). Its input should be the number of iterations, and its output should be which iterations of a parallelized for loop are executed by which thread. For example, if there are two threads and four iterations, the output might be the following: a. Thread 0: Iterations 0 −− 1 b. Thread 1: Iterations 2 −− 3
3	Write a OpenMP program to calculate n Fibonacci numbers using tasks.
4	Write a OpenMP program to find the prime numbers from 1 to n employing parallel for directive. Record both serial and parallel execution times.
5	Write a MPI Program to demonstration of MPI_Send and MPI_Recv.
6	Write a MPI program to demonstration of deadlock using point to point communication and avoidance of deadlock by altering the call sequence
7	Write a MPI Program to demonstration of Broadcast operation.
8	Write a MPI Program demonstration of MPI_Scatter and MPI_Gather
9	Write a MPI Program to demonstration of MPI_Reduce and MPI_Allreduce (MPI_MAX, MPI_MIN, MPI_SUM, MPI_PROD)

Program 1: **Write a OpenMP program to sort an array on n elements using both sequential and parallel mergesort (using Section). Record the difference in execution time.**

```
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>
#include <time.h>

// Function to merge two halves of the array
void merge(int* arr, int l, int m, int r) {     int i, j, k;     int n1 = m - l + 1;
    int n2 = r - m;

    int* L = (int*)malloc(n1 * sizeof(int));     int* R = (int*)malloc(n2 * sizeof(int));

    for (i = 0; i < n1; i++) L[i] = arr[l + i];     for
(j = 0; j < n2; j++) R[j] = arr[m + 1 + j];

    i = 0; j = 0; k = l;     while (i < n1 && j < n2) {
arr[k++] = (L[i] <= R[j]) ? L[i++] : R[j++];
    }
while (i < n1) arr[k++] = L[i++];
    while (j < n2) arr[k++] = R[j++];

    free(L);     free(R);
}

// Sequential MergeSort
void mergeSortSequential(int* arr, int l, int r) {     if (l < r) {         int m = (l + r) / 2;
        mergeSortSequential(arr, l, m);         mergeSortSequential(arr, m + 1, r);
        merge(arr, l, m, r);
    }
}

// Parallel MergeSort using OpenMP sections
void mergeSortParallel(int* arr, int l, int r, int depth) {     if (l < r) {
        int m = (l + r) / 2;

        if (depth <= 0) {
            // Fallback to sequential at depth limit                    mergeSortSequential(arr, l, m);
mergeSortSequential(arr, m + 1, r);         } else {
            #pragma omp parallel sections
            {
                #pragma omp section
                mergeSortParallel(arr, l, m, depth - 1);

                #pragma omp section
                mergeSortParallel(arr, m + 1, r, depth - 1);
            }
        }

        merge(arr, l, m, r);
    }
}

// Helper to check if array is sorted int isSorted(int* arr, int n) {     for (int i = 1; i < n; i++) {         if
(arr[i - 1] > arr[i]) return 0;
    }     return 1;
}

int main() {     int n = 1000000;     int* arrSeq = (int*)malloc(n * sizeof(int));
    int* arrPar = (int*)malloc(n * sizeof(int));

    // Seed for reproducibility     srand(42);     for (int i = 0; i < n; i++) {         arrSeq[i] = rand() %
100000;         arrPar[i] = arrSeq[i];
    }

    // Time sequential sort     double start = omp_get_wtime();     mergeSortSequential(arrSeq, 0, n -
1);     double end = omp_get_wtime();
    double timeSeq = end - start;

    // Time parallel sort     start = omp_get_wtime();     mergeSortParallel(arrPar, 0, n - 1, 4);  // You
can tune depth
    end = omp_get_wtime();
    double timePar = end - start;

    // Validate and output     printf("Sequential sort time: %.6f
seconds\n", timeSeq);       printf("Parallel sort time   : %.6f
seconds\n", timePar);     printf("Speedup              : %.2fx\n",
timeSeq   /   timePar);               if   (!isSorted(arrSeq,   n))
printf("Sequential sort failed!\n");     if (!isSorted(arrPar, n))
printf("Parallel sort failed!\n");

    free(arrSeq);
    free(arrPar);
return 0;
}
```

OUTPUT



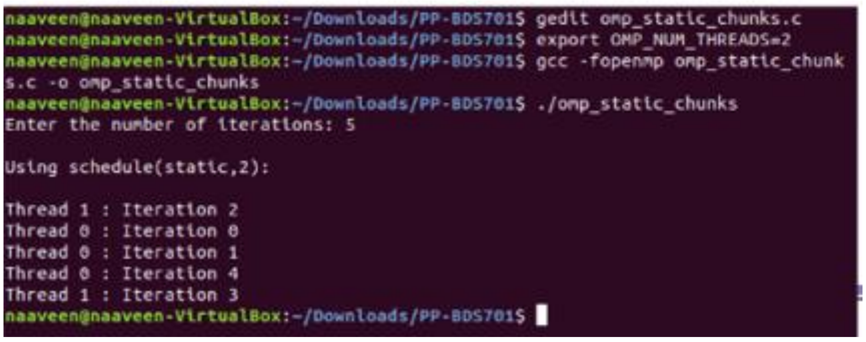
```
naaveen@naaveen-VirtualBox:~/Downloads/PP-BDS701$ gcc -fopenmp mergesort_openmp
.c -o mergesort
naaveen@naaveen-VirtualBox:~/Downloads/PP-BDS701$ ./mergesort
Sequential sort time: 0.188905 seconds
Parallel sort time  : 0.174429 seconds
Speedup             : 1.08x
```

<u>Program 2</u> Write an OpenMP program that divides the Iterations into chunks containing 2 iterations, respectively (OMP_SCHEDULE=static,2). Its input should be the number of iterations, and its output should be which iterations of a parallelized for loop are executed by which thread. For example, if there are two threads and four iterations, the output might be the following:

a. Thread 0: Iteration 0-1

b.Thread 1: Iterations 2 – 3

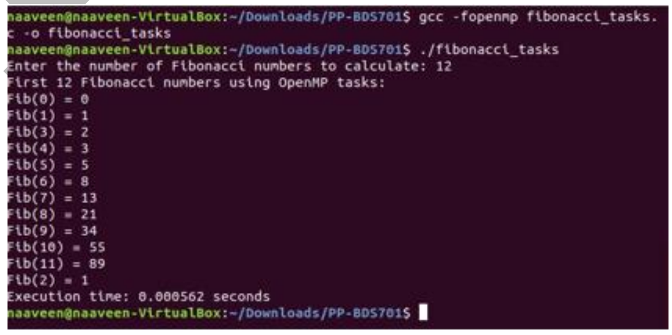
```
#include <stdio.h>

#include <omp.h>

int main() {

    int num_iterations;


    printf("Enter the number of iterations: ");

    scanf("%d", &num_iterations);

    // Optional: Set number of threads (or use OMP_NUM_THREADS)

    // omp_set_num_threads(2);


    printf("\nUsing schedule(static,2):\n\n");

    #pragma omp parallel

    {

        int tid = omp_get_thread_num();


        #pragma omp for schedule(static, 2)

        for (int i = 0; i < num_iterations; i++) {

            printf("Thread %d : Iteration %d\n", tid, i);

        }

    }


    return 0;

}
```



```
naaveen@naaveen-VirtualBox:~/Downloads/PP-BDS701$ gedit omp_static_chunks.c
naaveen@naaveen-VirtualBox:~/Downloads/PP-BDS701$ export OMP_NUM_THREADS=2
naaveen@naaveen-VirtualBox:~/Downloads/PP-BDS701$ gcc -fopenmp omp_static_chunk
s.c -o omp_static_chunks
naaveen@naaveen-VirtualBox:~/Downloads/PP-BDS701$ ./omp_static_chunks
Enter the number of iterations: 5

Using schedule(static,2):

Thread 1 : Iteration 2
Thread 0 : Iteration 0
Thread 0 : Iteration 1
Thread 0 : Iteration 4
Thread 1 : Iteration 3
naaveen@naaveen-VirtualBox:~/Downloads/PP-BDS701$ 
```

<u>Program 3:</u> Write a OpenMP program to calculate n Fibonacci numbers using tasks.

```
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>
// Recursive Fibonacci using OpenMP tasks int fib(int n) {     int x, y;     if (n <= 1) return n;
   #pragma omp task shared(x)
   x = fib(n - 1);
   #pragma omp task shared(y)     y = fib(n- 2);
   #pragma omp taskwait
   return x + y;
} int main() {      int
n;
   printf("Enter the number of Fibonacci numbers to calculate: ");
scanf("%d", &n);     if (n <= 0) {
       printf("Please enter a positive integer.\n");        return 0;
   }
   printf("First %d Fibonacci numbers using OpenMP tasks:\n", n);     double start
= omp_get_wtime();
   #pragma omp parallel
   {
       #pragma omp single
       {
           for (int i = 0; i < n; i++) {               int
result;
               #pragma omp task shared(result)
               {
                   result = fib(i);
                   #pragma                omp                critical
printf("Fib(%d) = %d\n", i, result);                }

           }
       }
}
double end =omp_get_wtime();
printf(“Execution Time:%.6f second\n”, end-start);
return 0;
}
```

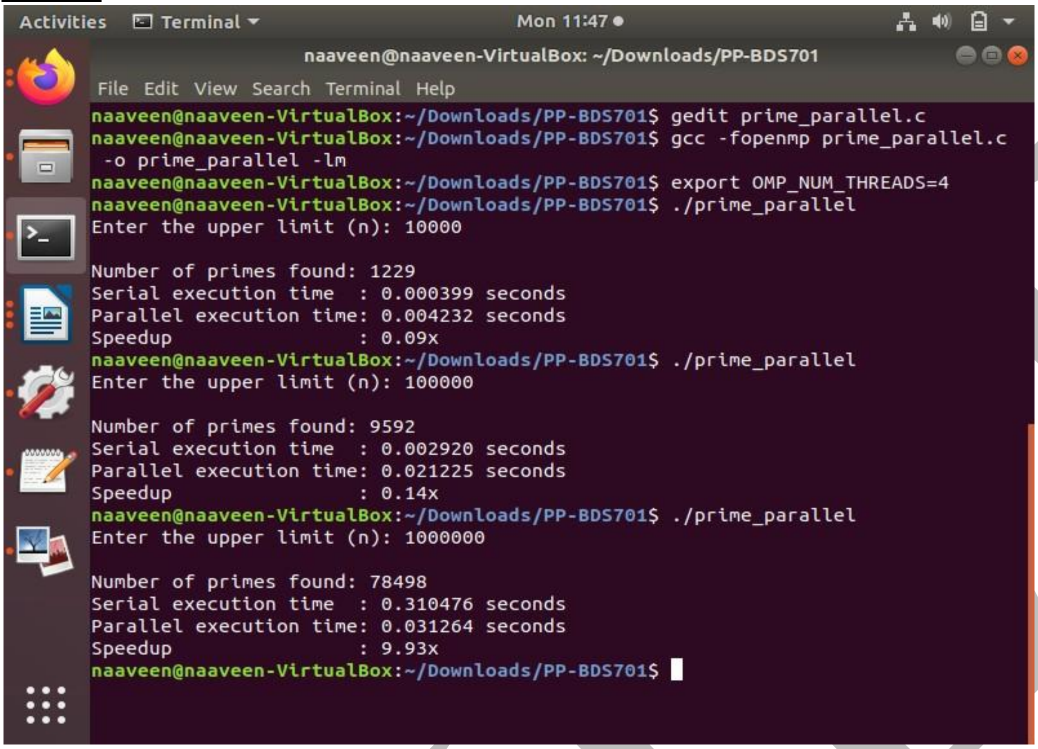


```
naaveen@naaveen-VirtualBox:~/Downloads/PP-BDS701$ gcc -fopenmp fibonacci_tasks.
c -o fibonacci_tasks
naaveen@naaveen-VirtualBox:~/Downloads/PP-BDS701$ ./fibonacci_tasks
Enter the number of Fibonacci numbers to calculate: 12
First 12 Fibonacci numbers using OpenMP tasks:
Fib(0) = 0
Fib(1) = 1
Fib(3) = 2
Fib(4) = 3
Fib(5) = 5
Fib(6) = 8
Fib(7) = 13
Fib(8) = 21
Fib(9) = 34
Fib(10) = 55
Fib(11) = 89
Fib(2) = 1
Execution time: 0.000562 seconds
naaveen@naaveen-VirtualBox:~/Downloads/PP-BDS701$
```

Program 4: **Write a OpenMP program to find the prime numbers from 1 to n employing parallel for directive. Record both serial and parallel execution times.**

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <omp.h>
// Check if a number is prime int is_prime(int num) {     if (num <= 1) return 0;     if (num == 2) return 1;
if (num % 2 == 0) return 0;    int limit = (int)sqrt(num);     for (int i = 3; i <= limit; i += 2) {         if (num %
i == 0) return 0;
    }
    return 1;
} int main() {
    int n;
    printf("Enter the upper limit (n): ");     scanf("%d", &n);     if (n
< 2) {         printf("There are no prime numbers <= %d\n", n);
        return 0;
    }
    // SERIAL VERSION     double start_serial = omp_get_wtime();
int* primes_serial = (int*)malloc((n + 1) * sizeof(int));          int
count_serial = 0;     for (int i = 2; i <= n; i++) {         if (is_prime(i)) {
printf("\nNumber of primes found: %d\n", count_serial);      printf("Serial execution
time   : %.6f seconds\n", time_serial);        printf("Parallel execution time: %.6f
seconds\n", time_parallel);      printf("Speedup                   : %.2fx\n", time_serial /
time_parallel);
    // Optional: Print the primes
    /*
    printf("\nPrimes:\n");         for (int i = 0; i <
count_parallel; i++) {                  printf("%d ",
primes_parallel[i]);
    }
    printf("\n");
    */
    free(primes_serial);
free(primes_parallel);     return 0; }
```

Output

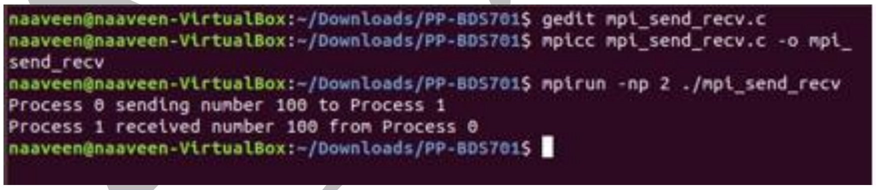

Activities
Terminal
Mon 11:47
naaveen@naaveen-VirtualBox: ~/Downloads/PP-BDS701
File Edit View Search Terminal Help

```
naaveen@naaveen-VirtualBox:~/Downloads/PP-BDS701$ gedit prime_parallel.c
naaveen@naaveen-VirtualBox:~/Downloads/PP-BDS701$ gcc -fopenmp prime_parallel.c
 -o prime_parallel -lm
naaveen@naaveen-VirtualBox:~/Downloads/PP-BDS701$ export OMP_NUM_THREADS=4
naaveen@naaveen-VirtualBox:~/Downloads/PP-BDS701$ ./prime_parallel
Enter the upper limit (n): 10000

Number of primes found: 1229
Serial execution time   : 0.000399 seconds
Parallel execution time: 0.004232 seconds
Speedup                 : 0.09x
naaveen@naaveen-VirtualBox:~/Downloads/PP-BDS701$ ./prime_parallel
Enter the upper limit (n): 100000

Number of primes found: 9592
Serial execution time   : 0.002920 seconds
Parallel execution time: 0.021225 seconds
Speedup                 : 0.14x
naaveen@naaveen-VirtualBox:~/Downloads/PP-BDS701$ ./prime_parallel
Enter the upper limit (n): 1000000

Number of primes found: 78498
Serial execution time   : 0.310476 seconds
Parallel execution time: 0.031264 seconds
Speedup                 : 9.93x
naaveen@naaveen-VirtualBox:~/Downloads/PP-BDS701$ 
```

Program 5: Write a MPI Program to demonstration of MPI_Send and MPI_Recv

```
#include <mpi.h>

#include <stdio.h>

int main(int argc, char** argv) {     int rank, size;     int number;

    MPI_Init(&argc, &argv);              // Initialize MPI environment

    MPI_Comm_rank(MPI_COMM_WORLD,        &rank);      //      Get      current      process      ID
MPI_Comm_size(MPI_COMM_WORLD, &size); // Get number of processes

    if (size < 2) {

        if (rank == 0) {            printf("This program requires at least 2 processes.\n");

        }

        MPI_Finalize();        return 0;

    }     if (rank == 0) {        number = 100; // Example message

        printf("Process 0 sending number %d to Process 1\n", number);

        MPI_Send(&number, 1, MPI_INT, 1, 0, MPI_COMM_WORLD);

    } else if (rank == 1) {

 MPI_Recv(&number, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);

        printf("Process 1 received number %d from Process 0\n", number);

    }

    MPI_Finalize(); // Clean up the MPI environment     return 0;  }
```

Output:

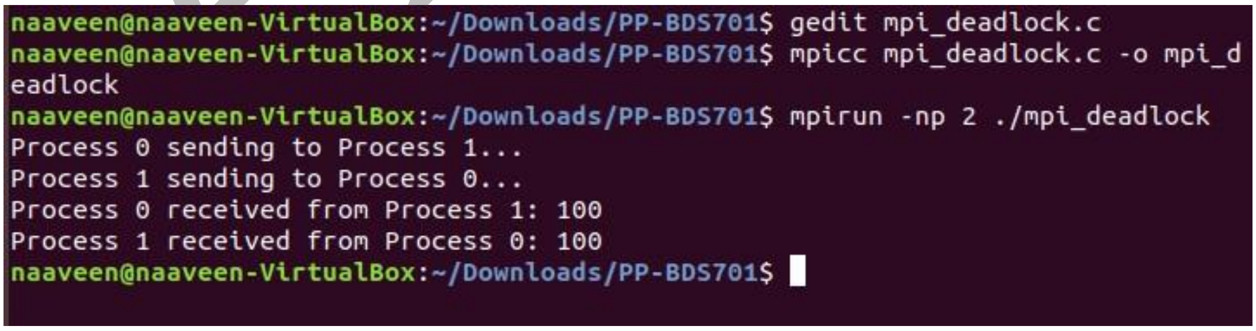


```
naaveen@naaveen-VirtualBox:~/Downloads/PP-BDS701$ gedit mpi_send_recv.c
naaveen@naaveen-VirtualBox:~/Downloads/PP-BDS701$ mpicc mpi_send_recv.c -o mpi_
send_recv
naaveen@naaveen-VirtualBox:~/Downloads/PP-BDS701$ mpirun -np 2 ./mpi_send_recv
Process 0 sending number 100 to Process 1
Process 1 received number 100 from Process 0
naaveen@naaveen-VirtualBox:~/Downloads/PP-BDS701$ 
```

Program 6: Write a MPI program to demonstration of deadlock using point to point communication and avoidance of deadlock by altering the call sequence.

```
#include <mpi.h>

#include <stdio.h> int main(int argc, char** argv) {     int rank, size;

    int msg_send = 100, msg_recv;     MPI_Init(&argc, &argv);

    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    MPI_Comm_size(MPI_COMM_WORLD, &size);     if (size < 2) {         if (rank == 0)

            printf("Run with at least 2 processes.\n");        MPI_Finalize();        return 0;

    }

    if (rank == 0) {

        printf("Process 0 sending to Process 1...\n");

        MPI_Send(&msg_send, 1, MPI_INT, 1, 0, MPI_COMM_WORLD); // blocking send
MPI_Recv(&msg_recv,1,MPI_INT,1,0,MPI_COMM_WORLD, MPI_STATUS_IGNORE);

        printf("Process 0 received from Process 1: %d\n", msg_recv);

    } else if (rank == 1) {

        printf("Process 1 sending to Process 0...\n");

        MPI_Send(&msg_send, 1, MPI_INT, 0, 0, MPI_COMM_WORLD); // blocking send
MPI_Recv(&msg_recv,1,MPI_INT,0,0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);

        printf("Process 1 received from Process 0: %d\n", msg_recv);

    }

    MPI_Finalize();     return 0; }
```

Output:

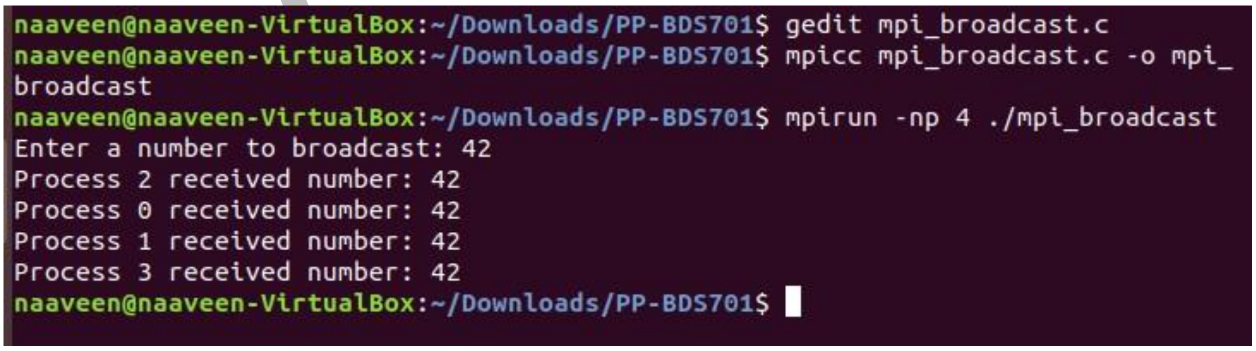


```
naaveen@naaveen-VirtualBox:~/Downloads/PP-BDS701$ gedit mpi_deadlock.c
naaveen@naaveen-VirtualBox:~/Downloads/PP-BDS701$ mpicc mpi_deadlock.c -o mpi_d
eadlock
naaveen@naaveen-VirtualBox:~/Downloads/PP-BDS701$ mpirun -np 2 ./mpi_deadlock
Process 0 sending to Process 1...
Process 1 sending to Process 0...
Process 0 received from Process 1: 100
Process 1 received from Process 0: 100
naaveen@naaveen-VirtualBox:~/Downloads/PP-BDS701$
```

Program 7: Write a MPI Program to demonstration of Broadcast operation

```
#include <mpi.h>
#include <stdio.h>
int main(int argc, char** argv) {
int rank, size;     int number;
    // Initialize the MPI environment
    MPI_Init(&argc, &argv);

    // Get the rank and number of processes
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    // Process 0 gets the input    if (rank == 0) {        printf("Enter a number to broadcast: ");        fflush(stdout);
// Ensure prompt is printed before input        scanf("%d", &number);
    }

    // Broadcast the number from process 0 to all other processes
    MPI_Bcast(&number, 1, MPI_INT, 0, MPI_COMM_WORLD);

    // Each process prints the received number     printf("Process %d received number: %d\n", rank, number);

    // Finalize the MPI environment     MPI_Finalize();     return 0;
}
```

Output:

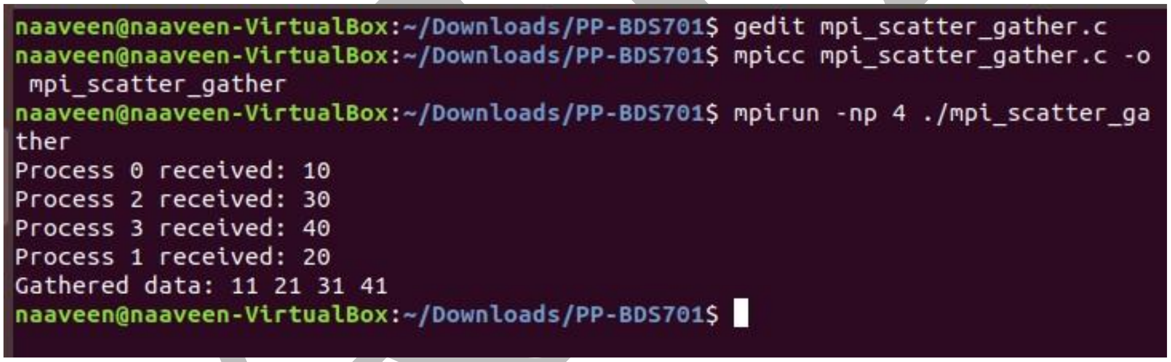


```
naaveen@naaveen-VirtualBox:~/Downloads/PP-BDS701$ gedit mpi_broadcast.c
naaveen@naaveen-VirtualBox:~/Downloads/PP-BDS701$ mpicc mpi_broadcast.c -o mpi_
broadcast
naaveen@naaveen-VirtualBox:~/Downloads/PP-BDS701$ mpirun -np 4 ./mpi_broadcast
Enter a number to broadcast: 42
Process 2 received number: 42
Process 0 received number: 42
Process 1 received number: 42
Process 3 received number: 42
naaveen@naaveen-VirtualBox:~/Downloads/PP-BDS701$
```

Program 8: Write a MPI Program demonstration of MPI_Scatter and MPI_Gather

```
#include <stdio.h>

#include <mpi.h>

int main(int argc, char** argv)

{ int rank, size, send_data[4] = {10, 20, 30, 40}, recv_data; MPI_Init(&argc, &argv);

MPI_Comm_rank(MPI_COMM_WORLD, &rank);

MPI_Comm_size(MPI_COMM_WORLD, &size);

MPI_Scatter(send_data, 1, MPI_INT, &recv_data, 1, MPI_INT, 0, MPI_COMM_WORLD); printf("Process
%d received: %d\n", rank, recv_data); recv_data += 1;

MPI_Gather(&recv_data, 1, MPI_INT, send_data, 1, MPI_INT, 0, MPI_COMM_WORLD);

if (rank == 0)

{ printf("Gathered data: "); for (int i = 0; i< size; i++) printf("%d ", send_data[i]); printf("\n");

}        MPI_Finalize();        return 0;

}
```

Output

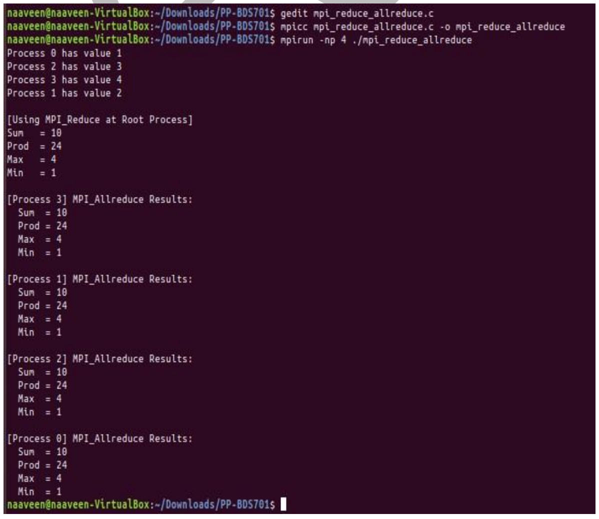


```
naaveen@naaveen-VirtualBox:~/Downloads/PP-BDS701$ gedit mpi_scatter_gather.c
naaveen@naaveen-VirtualBox:~/Downloads/PP-BDS701$ mpicc mpi_scatter_gather.c -o
 mpi_scatter_gather
naaveen@naaveen-VirtualBox:~/Downloads/PP-BDS701$ mpirun -np 4 ./mpi_scatter_ga
ther
Process 0 received: 10
Process 2 received: 30
Process 3 received: 40
Process 1 received: 20
Gathered data: 11 21 31 41
naaveen@naaveen-VirtualBox:~/Downloads/PP-BDS701$ 
```

<u>Program 9:</u> Write a MPI Program to demonstration of MPI_Reduce and MPI_Allreduce (MPI_MAX, MPI_MIN, MPI_SUM, MPI_PROD)

```
#include <mpi.h> #include <stdio.h> int main(int argc, char** argv) {    int rank, size;    int value;    int sum, prod, max, min;    int sum_all, prod_all, max_all, min_all;

    MPI_Init(&argc, &argv);

    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    MPI_Comm_size(MPI_COMM_WORLD, &size);

    // Each process sets its own value (e.g., rank + 1)    value = rank + 1;    printf("Process %d has value %d\n", rank, value);

    // ---------- MPI_Reduce ----------

    MPI_Reduce(&value, &sum,  1, MPI_INT, MPI_SUM,  0, MPI_COMM_WORLD);

    MPI_Reduce(&value, &prod, 1, MPI_INT, MPI_PROD, 0, MPI_COMM_WORLD);

    MPI_Reduce(&value, &max,  1, MPI_INT, MPI_MAX,  0, MPI_COMM_WORLD); MPI_Reduce(&value, &min,  1, MPI_INT, MPI_MIN,  0, MPI_COMM_WORLD);

    if (rank == 0) {        printf("\n[Using MPI_Reduce at Root Process]\n");        printf("Sum   = %d\n", sum); printf("Prod  = %d\n", prod);        printf("Max   = %d\n", max);        printf("Min   = %d\n", min);

    }

    // ---------- MPI_Allreduce ----------

    MPI_Allreduce(&value, &sum_all,  1, MPI_INT, MPI_SUM,  MPI_COMM_WORLD);

    MPI_Allreduce(&value, &prod_all, 1, MPI_INT, MPI_PROD, MPI_COMM_WORLD);


    MPI_Allreduce(&value, &max_all,  1, MPI_INT, MPI_MAX,  MPI_COMM_WORLD); MPI_Allreduce(&value, &min_all, 1, MPI_INT, MPI_MIN,  MPI_COMM_WORLD);    printf("\n[Process %d] MPI_Allreduce Results:\n", rank);    printf("  Sum  = %d\n", sum_all);    printf("  Prod = %d\n", prod_all); printf("  Max  = %d\n", max_all);    printf("  Min  = %d\n", min_all);    MPI_Finalize();    return 0; }
```

Output



```
naaveen@naaveen-VirtualBox:~/Downloads/PP-BDS701$ gedit mpi_reduce_allreduce.c
naaveen@naaveen-VirtualBox:~/Downloads/PP-BDS701$ mpicc mpi_reduce_allreduce.c -o mpi_reduce_allreduce
naaveen@naaveen-VirtualBox:~/Downloads/PP-BDS701$ mpirun -np 4 ./mpi_reduce_allreduce
Process 0 has value 1
Process 2 has value 3
Process 3 has value 4
Process 1 has value 2

[Using MPI_Reduce at Root Process]
Sum   = 10
Prod  = 24
Max   = 4
Min   = 1

[Process 3] MPI_Allreduce Results:
  Sum  = 10
  Prod = 24
  Max  = 4
  Min  = 1

[Process 1] MPI_Allreduce Results:
  Sum  = 10
  Prod = 24
  Max  = 4
  Min  = 1

[Process 2] MPI_Allreduce Results:
  Sum  = 10
  Prod = 24
  Max  = 4
  Min  = 1

[Process 0] MPI_Allreduce Results:
  Sum  = 10
  Prod = 24
  Max  = 4
  Min  = 1
naaveen@naaveen-VirtualBox:~/Downloads/PP-BDS701$
```