Parallel Computing - Exam Questions

Problem 1 (2 + 2 + 4 + 2 = 10 p)

1. What is the difference between task and data parallelism?

- Task parallelism
    - focuses on distributing tasks (distinct functions) across processors
    - those may work in the same or different datasets
- Data parallelism
    - involves distributing data across processors
    - the same operation is performed on different pieces of data

2. Explain MISD and MIMD.

- MISD (Multiple Instruction streams, Single Data streams)
    - a theoretical model where multiple instructions operate on the same data
    - not commonly used in practice
    - An example could be applying different filters to the same video frame simultaneously.
- MIMD (Multiple Instruction streams, Multiple Data streams)
    - allows for multiple autonomous processors to execute different instructions on different data
    - widely used in modern parallel computing environments,
    - enabling diverse computational tasks to be performed in parallel, enhancing computing efficiency and speed

3. Explain the decomposition, assignment, and orchestration steps when creating parallel programs.

- decomposition involves dividing computation into tasks
    - the objective is to identify portions of the computation that can be

executed concurrently, considering the dependencies among tasks

- assignment distributes these tasks to workers
    - involves decisions about task scheduling, prioritizing tasks based on dependencies, and balancing the load among the processors to avoid bottlenecks
- orchestration manages data access, communication, and synchronization among processes, and mapping assigns processes to processors
    - includes synchronizing tasks to ensure that data dependencies are respected, managing communication between tasks, especially in distributed systems, and handling any runtime dynamics that may affect the execution order or the assignment of tasks

4. Is uneven load a problem when we create parallel programs? Motivate and explain what we can do to avoid it.

- uneven load can be a problem, leading to inefficiencies as some processors may remain idle while others are overburdened.
- strategies to avoid uneven load include dynamic workload distribution and load balancing techniques.
    - load balancing can be achieved through various strategies, such as work-stealing, where idle processors can "steal" tasks from busier processors, or
    - by partitioning the problem domain in a way that allows for more flexible task assignment.

Problem 2 (2 + 4 + 4 = 10 p)

1. Explain message-passing and shared memory.

- Message-Passing
    - processes communicate by explicitly sending and receiving messages
    - this model is well-suited for distributed systems where processes may run on different physical machines

        - it provides a clear and structured way to handle communication but requires careful design to avoid deadlocks and ensure efficient data transfer
    - Shared Memory
        - the shared memory model allows multiple processes to access common memory spaces
        - this model is more intuitive for programmers as it resembles the traditional single-threaded programming model
        - however, it requires mechanisms to ensure consistency and prevent race conditions, such as locks, semaphores, or barriers

2. Explain how we can automatically parallelise a program.

- term automatic parallelization can be misleading as a programmer still needs to provide hints or structure the code in a way that makes parallelization more straightforward for the compiler or runtime system
- the automatic part often refers more to the dynamic assignment of tasks to available processing units rather than the complete absence of manual intervention in the parallelization process
- while automatic parallelization aims to reduce the manual effort required to parallelize code, achieving optimal parallel performance often requires a blend of compiler/runtime support and intelligent guidance from the programmer
    - this can include using specific parallel libraries, adding annotations, restructuring code to make it more amenable to parallelization, and explicitly managing some aspects of task distribution and execution

3. Why is it difficult to automatically parallelise?

Automatic parallelization faces several challenges,

- including accurately identifying independent tasks that can be executed in parallel,
- dealing with data dependencies that limit parallelism,
- and determining the most efficient way to distribute tasks and data across processors
- Additionally, optimizing communication and synchronization among parallel tasks

to minimize overhead and ensure correct execution is also a significant challenge.

Problem 3 (5 + 2 + 3 = 10 p)

1. Explain ILP. What is it, what is its relation to parallel computing, and what issues are there?

- Instruction Level Parallelism (ILP)
    - is the measure of how many of the operations in a computer program can be performed simultaneously
    - reflects the ability of a processor to execute multiple instructions at the same time, without having to complete one instruction before starting the next
    - is a key factor in the performance of a processor, exploiting parallelism that exists within a single program thread
    - Key concepts of ILP
        - Pipelining
            - is a technique where different stages of instruction execution (like fetching, decoding, executing, and writing back) are overlapped.
            - that means that a CPU can have issued a command, but doesn't need to actively manage every step of that operation's execution after it's initiated (like a read or write operation to memory, or sending data to an I/O device)
        - Superscalar Execution
            - refers to the ability of a CPU to execute more than one instruction during a single clock cycle
            - CPUs have several execution units (arithmetic logic units, floating-point units, load/store units, etc.) that can operate in parallel, assuming the instructions are independent and there are no data hazards

            - its design allows it to decode and prepare several instructions for execution simultaneously, further enhancing its ability to process instructions in parallel
        - Out-of-order Execution
            - allows the CPU to pick and choose which instructions to execute next, based on availability of data and execution units, rather than following the program order strictly
            - if the CPU encounters an instruction that it can't execute right away (perhaps because it's waiting for data to be fetched from memory), it can execute another instruction that's ready to go
            - this helps keep the CPU busy and improves efficiency
            - it is also beneficial in handling branch predictions and data dependencies more efficiently
                - if a branch prediction goes wrong, the CPU might need to discard or roll back executed instructions
    - Issues with ILP
        - **Data Hazards:** Situations where instructions that are scheduled to execute in parallel depend on each other's results, leading to delays.
        - **Control Hazards:** Caused by branch instructions (like if-else or loops) that can alter the flow of execution, making it hard to predict which instructions can be executed in parallel.
        - **Resource Conflicts:** Occur when instructions compete for the same resources (like memory or execution units), causing stalls.
        - **Diminishing Returns:** Beyond a certain point, adding more parallelism does not yield significant performance

improvements, due to increased complexity and overheads in managing it.

2. Explain the memory wall. What is the issue and why do we consider it a "wall"?

The memory wall problem arises from the growing gap between the speed of CPUs and memory. As CPUs become faster, the relative latency of memory access becomes a significant bottleneck, causing CPUs to spend a considerable amount of time waiting for data from memory. This discrepancy limits the overall system performance and is referred to as hitting the "memory wall."

3. Give two examples of how we can reduce the impact of the memory wall.

To mitigate the impact of the memory wall, several strategies can be employed:

- Cache Optimization: Designing more efficient cache hierarchies and optimizing cache algorithms to maximize cache hits and minimize cache misses.
- Prefetching: Predictively loading data into cache before it is required by the CPU, reducing wait times for memory access.
- Memory Access Optimization: Optimizing algorithms and data structures to improve locality of reference, ensuring that data accessed in close temporal proximity is also close in memory, thereby reducing the frequency of slow memory accesses.

Problem 4 (2 + 4 + 4 = 10 p)

1. GPUs are said to be SIMT. Explain.

- GPUs (Graphics Processing Units) are often described using the term SIMT, which stands for Single Instruction, Multiple Threads
- this concept is closely related to SIMD (Single Instruction, Multiple Data) from Flynn's taxonomy but with a key distinction that makes it uniquely suited to the architecture and processing style of modern GPUs

- SIMT architecture allows multiple threads to execute the same instruction but on different data
- this is similar to SIMD in that a single operation is applied across many data elements in parallel
- however, the SIMT model provides a more flexible approach that accommodates a wider variety of parallel patterns and data structures, characteristic of the diverse workloads GPUs handle, especially in graphics rendering and general-purpose computing (GPGPU tasks).
- Key Features:
    - **Thread Execution:** In SIMT, each thread can follow its own control flow. Even though all threads in a group (often called a warp in NVIDIA terminology or wavefront by AMD) execute the same instruction at any given cycle, individual threads can diverge, executing different branches based on their unique data.
    - **Efficiency and Flexibility:** This architecture strikes a balance between the efficiency of executing a single instruction across many data elements (like SIMD) and the flexibility to handle divergent paths and data-dependent control flows, making it particularly powerful for parallel algorithms.

2. Give examples of a problem that is suitable to solve on GPUs and one that is not. Motivate

Suitable Problems

- Tasks with a high degree of data parallelism, due to their ability to execute many parallel threads efficiently.
- GPUs, with their SIMT architecture and parallel processing capabilities, excel at tasks that involve processing large amounts of data in parallel, where the same operation is applied to many data elements
    - matrix multiplication
    - image processing
    - large-scale simulations
- Graphics Rendering: The original and most obvious application of GPUs. Rendering involves computing the color and intensity of millions of pixels in parallel, a task GPUs are specifically designed for.

- Scientific Simulations and Numerical Analysis: Simulations that involve large matrices and vectors, like those found in physics simulations or climate modeling, can be significantly accelerated by GPUs. Operations like matrix multiplication and Fourier transforms benefit from the parallelism offered by GPUs.
- Deep Learning and Neural Networks: Training deep learning models involves a lot of matrix and vector operations, which can be parallelized efficiently on GPUs. The ability to process many operations in parallel significantly reduces training time for complex models.
- Image and Video Processing: Tasks like filtering, convolution, and transformation operations over large sets of pixels can be efficiently executed on GPUs due to their inherent parallelism.
- Data Analysis and Mining: Algorithms that can be expressed in parallel, such as sorting, searching, and pattern matching over large datasets, can be accelerated using GPU computing.

Unsuitable Problems

- Tasks with complex data dependencies, significant branching, or those that require extensive serial processing are less suitable for GPUs, as they cannot fully leverage the parallel processing capabilities of the GPU.
- Serial Tasks with Complex Dependencies: Tasks that involve operations which must be performed in a strict sequence, or where each step depends on the results of the previous step, are not well-suited for GPUs. The overhead of managing these dependencies often negates the benefits of parallel execution.
- Small Datasets: If the dataset is too small, the overhead of transferring data between the CPU and GPU memory can outweigh the performance benefits of parallel execution.
- Branch Heavy Logic: Algorithms that involve a lot of divergent branches (if-else conditions that differ widely across data elements) can lead to inefficiencies on GPUs. While SIMT architectures can handle divergence to some extent, excessive divergence can reduce the efficiency of parallel execution.
- Tasks with High I/O Latency: If a task is bottlenecked by input/output operations, such as reading from or writing to disk, then accelerating the computation on a GPU might not result in a significant overall performance gain.

Motivation

The motivation behind selecting suitable tasks for GPU acceleration lies in maximizing computational efficiency and reducing processing times. By leveraging the parallel processing capabilities of GPUs for appropriate tasks, applications can achieve significant performance improvements. For example, in deep learning, using GPUs can reduce training time from weeks to days or even hours, enabling more rapid development and experimentation. Similarly, in scientific simulations, GPUs can handle complex calculations over large datasets more efficiently than traditional CPUs, accelerating research and discovery.

3. What is a warp? What is important to consider when we use warps?

Architecture of a GPU

- **Streaming Multiprocessors (SMs):** A GPU contains multiple SMs, each of which can execute instructions independently. SMs are the primary building block of a GPU and are designed to handle thousands of threads concurrently.
- **Streaming Processors (SPs):** Within each SM, there are numerous SPs (also referred to as CUDA cores in NVIDIA GPUs or Stream Processors in AMD GPUs). These SPs execute the actual instructions and are responsible for the heavy lifting in data processing.
- **Warp/Wavefront Execution:** The threads are grouped into warps or wavefronts, with each group executing the same instruction at the same time on different data elements. If threads within a warp follow different execution paths due to branching, the warp can temporarily diverge, handling each branch separately before reconverging, which can impact performance.
    - In GPU computing, a warp is the basic unit of thread execution. It consists of a group of threads (typically 32 in NVIDIA GPUs) that execute the same instruction on different data elements simultaneously. Warps allow GPUs to achieve high levels of parallelism. When programming for GPUs, it's important to minimize warp divergence – scenarios where threads within the same warp follow different execution paths – as it can lead to underutilization of the GPU's computing resources.
    - A warp is a group of 32 threads (unchanged between generations)

- It is the unit of thread management, scheduling, and execution
- Each thread (in a warp):
    - Start at the same program address
    - Has its own program counter and registers
    - Can branch and execute independently

i. SM receives thread block

ii. SM partitions thread block into warps
    - 0, 1, … within each warp

iii. The warp scheduler schedules warps to run

iv. The scheduled warp executes one common instruction at a time (SIMD)

- Max efficiency if all threads execute the same path
- If paths diverge across threads
    - Execute each path serially
    - Threads not on that path disabled (temporarily)
    - Threads converge when all paths complete
- Multiple warps can run at the same time, independently
    - Depends on the number of warp schedulers
    - Paths diverge only within a warp
- Execution context stays on-chip
    - PC
    - Registers
    - Shared memory
- Context switches are free, can switch each clock cycle
- At instruction issue time, the scheduler:
    - selects a warp with active threads
    - issues instruction to the warp's threads
- The number of blocks/warps that fit on an SM depends on:
    - Registers and memory required by each kernel
    - Registers and memory available on the SM

- **Memory Hierarchy:** GPUs also feature a complex memory hierarchy, including global memory, shared memory (accessible by threads within the same SM), and registers (private to each thread). Efficient use of this memory hierarchy is critical for achieving high performance on GPU architectures.

Problem 5 (3 + 2 + 3 + 2 = 10 p)

1. Why is PRAM an unfeasible model?

- The interconnection network between processors and memory would require a very large area
- The message routing would require time proportional to the network size

2. Explain EREW and CREW in PRAM

- EREW (Exclusive Read Exclusive Write)
    - No concurrent read/writes to the same memory location
- CREW (Concurrent Read Exclusive Write)
    - Multiple processors may read from the same global memory location in the same instruction step

3. Explain Common, Arbitrary, Priority, and Common for CRCW in PRAM

- COMMON, all processors that write into the same memory address must write the same value
- ARBITRARY, one processor is chosen randomly and its value is written
- PRIORITY, the processor with the highest priority's value is written
- COMBINING, some combination, e.g., max, min, etc is written
- COMMON is most often used

4. Explain Isoefficiency.

Isoefficiency is a measure of how the problem size needs to scale with the number of processors to maintain a constant efficiency. It's a key concept in evaluating the scalability of parallel algorithms. An algorithm with good isoefficiency can handle larger problem sizes efficiently as the number of processors increases, making it well-suited for parallel execution on systems with varying numbers of processors.

- How much must the problem size increase to retain the same efficiency on when the number of processors increase?

        - A way to quantify scalability
- Remember, efficiency is defined as $\frac{T_1}{pT_p}$ or $E_p = \frac{S_p}{p}$.

Problem 6 (3 + 3 + 4 = 10 p)

1. Explain the Karp-Flatt metric. How is it defined and how do we use it?

The Karp-Flatt metric is a quantitative measure used to estimate the serial fraction of a parallel program. It helps to identify the portion of a program that limits its parallel scalability. The metric is calculated using the formula $e = \frac{\frac{1}{S_p} - \frac{1}{p}}{1 - \frac{1}{p}}$, where S_p is the observed speedup with p processors, and e is the estimated serial fraction. This metric is valuable for diagnosing performance bottlenecks and understanding how effectively a program utilizes parallel

- Amdahl's and Gustafson–Barsis's laws can overestimate the speedup since they ignore the overhead from parallelisation
- The Karp-Flatt metric is considered as another kind of inherently sequential work, so Amdahl's law can be used to determine a combined serial fraction, e

2. How do we estimate the time for a superstep in BSR?

In the Bulk Synchronous Parallel (BSP) model, the computation is divided into supersteps, each consisting of computation, communication, and a barrier synchronization at the end. The time for a superstep can be estimated by considering the maximum computation time among all processors, the communication time (based on the number of messages sent and received and the network bandwidth), and the synchronization time. The BSP model allows for predictable performance analysis by making the costs of computation, communication, and synchronization explicit.

BSP

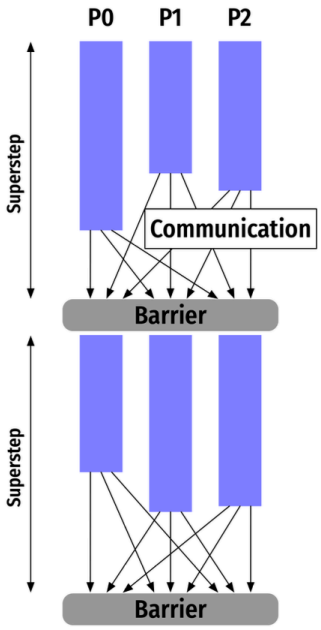
- Bulk Synchronous Parallelism
- A parallel programming model that uses SPMD style
- Supports both direct memory access and message-passing semantics

A BSP Computer

- A set of processor-memory pairs
- A communication point-to-point network
- A mechanism for efficient barrier synchronization of all processors

BSP supersteps

- A BSP computation consists of a sequence of supersteps
- In each superstep, processes execute computations using locally available data, and issue communication requests
- Processes synchronized at the end of the superstep, at which all communications issued have been completed

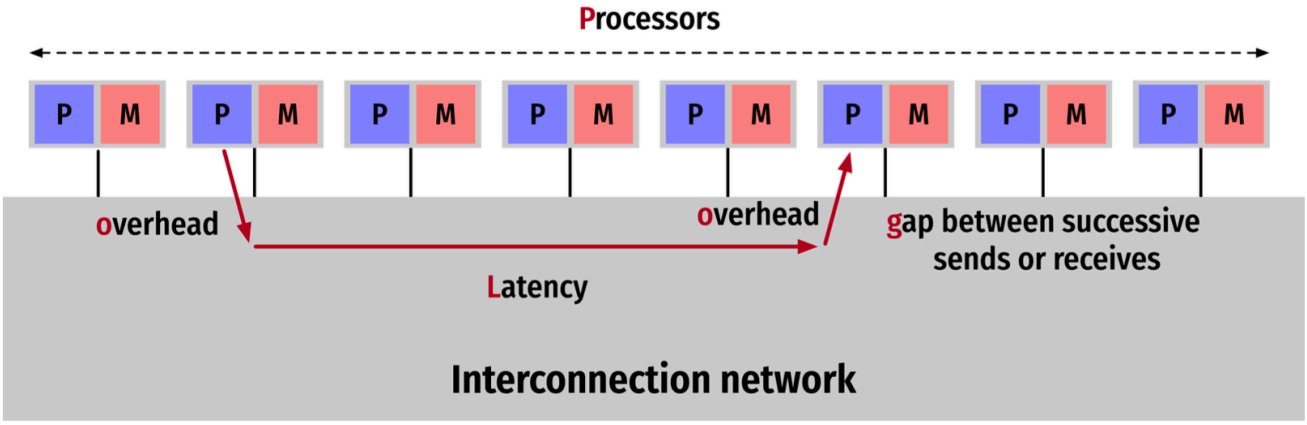


3. Explain LogP.

The LogP model is a more realistic model for parallel computation that takes into account the limitations of real-world systems, including Latency (L) of communication, Overhead (o) for sending and receiving messages, the Gap (g) between messages (inverse of bandwidth), and the number of Processors (P). The LogP model is used to design and analyze parallel algorithms by providing a framework to account for communication delays and processing overheads, making it possible to predict the performance of parallel programs on distributed-memory architectures more accurately.

LogP

- Processing
    - Powerful processor, large memory, cache, …
- Communication
    - Latency
    - Limited bandwidth
    - Overhead
- No consensus on a programming model, should notenforce one
- Latency in sending a (small) message between modules
- overhead felt by the processor on sending or receiving message
- gap between successive sends or receives (1/BW)
- Processors



LogP "Philosophy"

- Concerns
    - Mapping N words onto P processors
    - Computation within a processor
    - Communication between processors
- Characterise processor and network performance …
- … without thinking about what is happening within the network
- Values for g and l determined for machines, e.g.,
    - Cluster, g=40, l=5000-20000

Problem 1 (5 + 3 + 2 = 10 p)

1. We discussed three "walls": ILP, memory, and power. Explain these and how they relate to parallel programming.

- **ILP Wall:** Instruction-Level Parallelism (ILP) faces limitations due to complexity, diminishing returns with deeper pipelines, and hazards (structural, data, and control). ILP improvements are constrained by these challenges, pushing for parallel programming to enhance performance .
- **Memory Wall:** The gap between CPU speed and memory speed (latency and bandwidth) limits performance. Despite cache hierarchies designed to mitigate this, the increasing need for data with limited improvements in memory technology exacerbates the "memory wall" .
- **Power Wall:** Power consumption increases with higher frequencies and more transistors, leading to heat issues and limiting performance improvements. This necessitates energy-efficient parallel architectures to continue performance enhancements without excessive power use .

2. Explain how Simultaneous multithreading (SMT, also called hyper-threading) can help improve performance in a CPU.

SMT, also known as hyper-threading, allows a single physical processor to execute multiple threads concurrently. It increases ILP by utilizing idle resources within a CPU core, thereby improving performance and throughput without significantly increasing energy costs .

> Not really part of the lecture

3. Explain 2-way static issue and Very Long Instruction Word (VLIW).

- 2-way Static Issue: This refers to the ability of a processor to issue two instructions in a single clock cycle statically, i.e., as determined at compile time. This approach requires the compiler to schedule instructions carefully to avoid

hazards and maximize parallel execution .

- Very Long Instruction Word (VLIW): VLIW architectures bundle several operations into a single, wide instruction word that the processor executes in parallel. This design relies heavily on the compiler to ensure that bundled instructions are independent and can be executed without causing execution stalls.

Problem 3 (5 + 5 = 10 p)

1. Odd-even-transposition sort sorts an array by comparing values in two passes. First, i and $i + 1$ are compared, then $i + 2$ and $i + 3$ are compared. If the values are in the wrong order, they are swapped. The algorithm continues until no values are swapped. Sketch a parallel implementation using MPI.

A parallel MPI implementation of the odd-even-transposition sort involves distributing the array elements across MPI processes. Each process sorts local elements and then participates in parallel compare-exchange operations with neighboring processes to ensure global order. The processes alternately compare and exchange odd-even and even-odd indexed pairs in successive rounds until no swaps are necessary.

Initial Setup

1. **Divide the list among processors**: If you have a list of length l and p processors, each processor initially gets approximately l/p elements of the list. This division could be uneven if l is not perfectly divisible by p, so some processors might have one more element than others.
2. **Local and neighbor comparisons**: Each processor will perform local comparisons and swaps within its chunk of the list. For comparisons that involve elements on the border between two processors, the processors will send messages to each other to compare these elements and determine if a swap is needed.

Parallel Sorting Process

1. **Odd phase**: Each processor compares all odd-even pairs within its part of the list (e.g., compares elements at indices 0 and 1, 2 and 3, etc.). For the highest odd index in each processor's chunk, if it's not the last processor, it sends the element at this index to the next processor to compare with the first element of that processor's chunk. If a swap is needed, it's done across processors.
2. **Even phase**: Similar to the odd phase, but now each processor compares all even-odd pairs within its part of the list (e.g., elements at indices 1 and 2, 3 and 4, etc.). Again, for comparisons involving the last element of one processor's chunk and the first element of the next processor's chunk, messages are exchanged to compare and potentially swap these elements.
3. **Synchronization**: After each phase, use MPI barriers to synchronize all processors, ensuring that all comparisons and swaps for the phase are complete before proceeding to the next phase.
4. **Termination condition**: The algorithm continues iterating through odd and even phases until a pass occurs where no swaps are made. To detect this globally, all processors can use an MPI collective operation (like MPI_Allreduce) to combine their local "swap made" flags and determine if any swaps were made in the last pass by any processor.

Key Points

- **No need for merge-split cycles**: The list remains divided among the processors, with each processor working on sorting its chunk of the list in parallel. Processors communicate only for comparisons that involve elements at the boundaries of their chunks.
- **Efficient communication**: Use MPI messages for comparing and swapping boundary elements between neighboring processors.
- **Global synchronization**: Use MPI barriers and collective operations to synchronize and determine the global state of sorting.

2. How would you implement Odd-even-transposition sort on a GPU. It is sufficient to sketch the steps; you do not need to provide the exact CUDA calls. How would your implementation perform compared to a CPU-based shared-memory implementation? Can you modify the algorithm or your implementation to improve performance?

Implementing odd-even-transposition sort on a GPU involves using CUDA kernels to parallelize the comparison and swapping of adjacent elements. Each CUDA thread can be responsible for comparing and possibly swapping a pair of elements. The implementation can benefit from GPU's high degree of parallelism but must handle thread synchronization and data movement efficiently. Performance can often exceed CPU-based implementations due to higher parallelism but may require optimizations, such as minimizing divergence and maximizing memory throughput, to fully leverage the GPU's capabilities.

GPU Implementation Sketch

1. **Data Distribution**: Copy the array to be sorted from the host (CPU) memory to the device (GPU) memory. This step involves transferring the data over PCIe, which, while fast, is still a considerable factor in the overall performance.
2. **Kernel Configuration**: Configure the GPU kernels. A kernel is a function that runs on the GPU, and it needs to be configured with the number of threads per block and the number of blocks. These parameters heavily depend on the size of the array and the specific GPU hardware being used.
3. **Sorting in Parallel**:
    - **Odd Phase**: Launch a kernel where each thread compares and possibly swaps elements at odd-even positions (e.g., 0 and 1, 2 and 3, etc.). This step leverages the GPU's ability to perform many such comparisons in parallel.
    - **Even Phase**: Similarly, launch another kernel for comparing and swapping elements at even-odd positions (e.g., 1 and 2, 3 and 4, etc.).
4. **Synchronization**: Synchronize the threads after each phase to ensure that all comparisons and swaps for that phase are completed before starting the next

phase.

5. **Iteration**: Repeat the odd and even phases until the array is sorted. This may require keeping a flag on the GPU to indicate whether any swaps were made in the last pass, necessitating a check on this flag by the host to determine if another iteration is needed.
6. **Data Retrieval**: Once sorted, copy the array back from the GPU memory to the host memory.

Performance Comparison to CPU-Based Implementations

- **Parallelism**: The GPU implementation can perform far more comparisons and swaps in parallel than a CPU, due to the massive number of threads it supports. This parallelism can lead to significant speedups, especially for large arrays.
- **Memory Bandwidth**: GPUs have high memory bandwidth, which is beneficial for algorithms like the odd-even transposition sort that require frequent reading and writing of elements.
- **Overhead**: The overhead of copying data between the host and the GPU can be significant, especially for smaller data sets. For very large data sets, the time taken by the sorting computation itself tends to dwarf this overhead.

Performance Improvements

- **Minimize Data Transfer**: Keeping data on the GPU as much as possible and minimizing transfers between the host and the device can improve performance.
- **Optimized Memory Access**: Utilizing shared memory within blocks to reduce global memory accesses can enhance performance, as shared memory is significantly faster.
- **Coalesced Memory Accesses**: Ensuring that memory accesses are coalesced, where consecutive threads access consecutive memory locations, can improve memory access efficiency on the GPU.
- **Adaptive Approach**: For smaller arrays, it might be more efficient to perform the sorting on the CPU to avoid the overhead of data transfer to and from the GPU. An adaptive approach that chooses the sorting location based on the size of the data set could provide the best overall performance.

Problem 5 (3 + 2 + 5 = 10 p)

1. Explain the MPI programming model.

The MPI (Message Passing Interface) programming model is a standard for parallel computing that uses processes to perform computations. In MPI, data is moved from the memory of one process to that of another, enabling parallel computing across distributed systems. The model supports both point-to-point and collective communication, allowing processes to send and receive messages, synchronize, and perform collective operations like broadcast, gather, and scatter. MPI is designed to work on a wide variety of computing architectures, including single processors, shared-memory processors, and distributed-memory processors

- MPI is the only message passing library that can be considered a standard
- MPI is portable between multiple platforms
- Many different implementations available
    - Support for most programming languages, not just C and Fortran
    - Demos use MPICH on Ubuntu and mpi4py
- Rich specification, hundreds of functions
    - But still quite easy to learn, few functions needed to get started

2. Are calls such as `MPI_Send()` blocking or not? Does it matter?

`MPI_Send()` can be both blocking and non-blocking, depending on the context and how it's used. Blocking sends, like `MPI_Send()`, wait until the message has been copied out of the send buffer before returning control to the user program, which may lead to idle time if the receiving process is not ready. Non-blocking sends, like `MPI_Isend()`, return control to the user program immediately, allowing computations to overlap with communication. Whether a call is blocking or non-blocking matters because it affects the program's performance and how resources are utilized. Blocking operations can simplify program logic but may lead to inefficiencies, while non-blocking operations can improve performance but require more careful management of data dependencies and process synchronization

3. Show how `MPI_Barrier()` can be implemented using `MPI_Send()` and `MPI_Recv()`. Your solution should handle p processes. Start with $p = 2$ and show how your solution generalizes to p processes.

`MPI_Barrier()` is a synchronization operation that blocks all participating processes until all have reached the barrier. While the document doesn't explicitly describe implementing `MPI_Barrier()` using `MPI_Send()` and `MPI_Recv()`, the underlying principle involves ensuring all processes reach a point of synchronization by exchanging messages. Here is a conceptual approach:

1. Decomposition into Tasks: Each process sends a message to the next process in a circular fashion, indicating it has reached the barrier point.
2. Assignment to Workers: When a process receives a message from the previous process, it forwards a message to the next process. If it's the last process, it sends a message back to the first process.
3. Orchestration and Synchronization: The first process waits to receive the final message, ensuring all processes have reached the barrier. Then, it sends a release message back through the circle, allowing all processes to proceed.

This method leverages explicit message passing for synchronization, mimicking the barrier's functionality by ensuring all processes sequentially signal their arrival at the barrier and do not proceed until a release signal is circulated.

Problem 6 (3 + 2 + 2 + 3 = 10 p)

1. Explain Moore's, Amdahl's and Gustafson-Barsis' "laws".

- Moore's Law: Observes that the number of transistors on a microchip doubles approximately every two years, though the cost of computers is halved. This reflects the rapid pace of progress in hardware capabilities, highlighting the importance of parallel computing to leverage these advancements as individual processor speed improvements slow down.
- Amdahl's Law: Describes the speedup in latency of the execution of a task at

fixed workload that can be expected of a system whose resources are improved. Specifically, it offers a limit to the expected improvement in speedup for a fixed workload due to increased resources. Amdahl's law emphasizes that the speedup of a program using multiple processors in parallel computing is limited by the time needed for the sequential fraction of the program.

- Gustafson-Barsis's Law: Contrasts with Amdahl's law by suggesting that the speedup of a parallel system can be significantly higher if the workload increases with the number of processors, rather than fixing the problem size. This law addresses the scalability of parallel computing by arguing that larger problems can effectively utilize more processors, potentially leading to linear speedup as problem size grows.

2. Explain the actor model and how it can be used to implement parallel programs.

The actor model is a conceptual model that treats "actors" as the universal primitives of concurrent computation. In this model, actors can make local decisions, create more actors, send messages to other actors, and determine how to respond to the next message they receive. Actors can communicate with each other through message passing, allowing parallel and distributed computing without shared state. This model is beneficial for implementing parallel programs because it naturally encapsulates state within individual actors, avoiding many of the synchronization and data consistency issues found in shared-memory models. Actors can be used to structure parallel programs by decomposing tasks into smaller, independent units of work that communicate through asynchronous message passing, thus enabling scalable and flexible parallel systems.

3. Explain vexctor architectures.

Vector architectures are designed to process data formatted into vectors rather than processing each piece of data individually. Instead of executing operations on single data elements, vector processors can perform a single instruction on multiple data (SIMD) elements simultaneously. This makes vector architectures highly efficient for operations that can be applied across large datasets, such as mathematical operations on arrays or matrices. By leveraging SIMD, vector architectures can achieve high levels of parallelism within a single processor, significantly accelerating tasks that involve large amounts of data processing.

4. Explain loop fusion, fission, and hoisting.

- **Loop Fusion:** The process of combining two or more loops into a single loop that performs all the operations of the original loops. This optimization technique can reduce the overhead of loop control and improve cache utilization by minimizing the amount of data read and written multiple times.
- **Loop Fission (or Loop Distribution):** The opposite of loop fusion, where a single loop is split into multiple loops, each performing a part of the operations initially carried out by the original loop. This can help reduce the loop's complexity, making it easier to parallelize or optimize further.
- **Hoisting (or Loop Invariant Code Motion):** Involves moving code that does not change across iterations of a loop (invariant code) outside of the loop. This reduces the amount of work done within the loop, potentially decreasing the number of computations and improving performance.