Sub Code: MTCS201

M.TECH
(SEM II) THEORY EXAMINATION 2022-23
MULTICORE ARCHITECTURE AND PROGRAMMING

SECTION A

2 x 7 = 14

1. Attempt *all* questions in brief.

(a). Define Convoying.
Ans: In the context of parallel and concurrent programming, convoying refers to a situation where multiple threads are waiting for access to a shared resource, but only one of them can proceed at a time. This can result in a "traffic jam" of threads waiting for their turn to access the resource, similar to how vehicles in a convoy might wait in line to pass through a narrow road.
Convoying can lead to performance bottlenecks and reduced efficiency in a multi-threaded or multi-core environment. It occurs when threads contend for a resource that they need for execution, causing unnecessary delays and limiting the potential parallelism that could otherwise be achieved.

(b) What do you mean by Error diffusion?
Ans: Error diffusion is a technique for displaying continuous-tone digital images on devices that have limited color (tone) range. Printing an 8-bit grayscale image to a black-and-white printer is problematic. The printer, being a bi-level device, cannot print the 8-bit image natively. It must simulate multiple shades of gray by using an approximation technique.

(c) Explain barrier and Nowait in OpenMP.
Ans: Barriers: Barriers are a form of synchronization method that OpenMP employs to synchronize threads. Threads will wait at a barrier until all the threads in the parallel region have reached the same point.
Nowait : In OpenMP, a widely used framework for parallel programming, the nowait clause is used to indicate that a parallel region should execute without waiting for its parallel sections to complete before moving on to the next statement outside the parallel region. This clause is often used to avoid unnecessary synchronization and improve performance in situations where synchronization is not needed after parallel sections.

(d) Discuss the motivation for concurrency in software.
Ans: Concurrency in software is a way to manage the sharing of resources used at the same time. Concurrency in software is important for several reasons:

i) Concurrency allows for the most efficient use of system resources. Efficient resource utilization is the key to maximizing perform- ance of computing systems. Unnecessarily creating dependencies on different components in the system drastically lowers overall system performance.

ii) Many software problems lend themselves to simple concurrent implementations. Concurrency provides an abstraction for implementing software algorithms or applications that are naturally parallel.

(e) Define Semaphores.
Ans: Semaphores are integer variables that are used to solve the critical section problem by using two atomic operations, wait and signal that are used for process synchronization.
The definitions of wait and signal are as follows −
Wait: The wait operation decrements the value of its argument S, if it is positive. If S is negative or

zero, then no operation is performed.

```
wait(S)
{
   while (S<=0);

     S--;
 }
```

Signal: The signal operation increments the value of its argument S.

```
signal(S)
{
     S++;
 }
```

(f) Explain the term Live-Locks.

Ans: A live-lock is a situation in concurrent programming where multiple threads or processes are constantly reacting to each other's actions, preventing any meaningful progress and resulting in a loop of ineffective activity. Live-locks can arise due to various reasons, including overly cautious behavior, incorrect synchronization, and poor communication protocols in concurrent systems.

(g) How threads overhead can be minimized

Ans: Thread overhead can be minimized through the following approaches:

Thread Pooling: Instead of creating new threads for every task, maintain a pool of reusable threads.

Thread Reuse: Reuse threads for multiple tasks to avoid the overhead of creating and destroying threads frequently.

Fine-Grained Locking: Use fine-grained locks to allow multiple threads to access different parts of shared data simultaneously, minimizing contention.

Asynchronous I/O: Utilize asynchronous I/O operations to overlap thread waiting time with useful work, improving efficiency.

Batching: Group multiple small tasks into a single larger task to reduce the overhead of task scheduling and context switching.

Affinity Settings: Bind threads to specific processor cores to minimize cache thrashing and context switching.

Optimized Synchronization: Use lock-free or wait-free algorithms to minimize synchronization overhead.

Minimize Context Switching: Reduce unnecessary context switches by minimizing thread creation, using proper synchronization, and optimizing task scheduling.

Parallel Algorithms: Choose parallel algorithms that minimize synchronization requirements and contention, improving thread efficiency.

SECTION B

2. Attempt any *three* of the following: 7 x 3 = 21

(a) Explain Amdahl's Law. Also discuss its performance criteria and its limitations in parallel computing.

Ans: Amdahl's Law is a fundamental principle in parallel computing that provides insights into the potential speedup achievable by parallelizing a computation. It was formulated by computer architect Gene Amdahl in 1967 and is often used to analyze the benefits of parallelizing programs or processes.

Explanation of Amdahl's Law: Amdahl's Law is essentially a mathematical expression that quantifies the potential speedup of a program when a portion of it is parallelized. The law is based on the idea that not all parts of a program can be parallelized effectively, and there are sequential portions that cannot benefit from parallel processing.

The law is expressed by the following formula:

Speedup= $\frac{\mathbf{1}}{(\mathbf{1}-\boldsymbol{P})+\frac{\boldsymbol{P}}{\boldsymbol{N}}}$

Where:

- Speedup: The factor by which a program's execution time is reduced when parallelized.
- P: The proportion of the program that can be parallelized ($0 \leq P \leq 1$).
- N: The number of processing units or threads used for parallel execution.

Performance Criteria and Limitations:

1. Performance Criteria:
    - **Speedup:** Amdahl's Law helps us understand how much a program's execution time can be improved by parallelizing a portion of it. The speedup is calculated as the ratio of the original execution time to the parallel execution time. It provides a way to quantify the benefits of parallel processing.
    - **Efficiency:** Efficiency refers to how effectively the available processing resources are being utilized. It's calculated as the speedup divided by the number of processing units. Higher efficiency indicates better utilization of resources.
2. Limitations:
    - **Fixed Workload:** Amdahl's Law assumes a fixed workload, meaning the total amount of work that needs to be done remains constant. In many real-world scenarios, the workload may change dynamically, which can affect the applicability of the law.
    - **Unrealistic Assumptions:** The law assumes that the sequential portion of the program remains constant and that the parallel portion scales linearly with the number of processing units. In reality, these assumptions might not hold true for all types of applications.
    - **Neglects Overhead:** The law doesn't account for the overhead introduced by managing parallel threads, synchronization, and communication between processing units. These overheads can become significant and impact the actual speedup achieved.
    - **Strong vs. Weak Scaling:** Amdahl's Law does not distinguish between strong scaling (fixed problem size, increasing resources) and weak scaling (problem size scales with resources). Weak scaling scenarios may experience different limitations compared to strong scaling scenarios.

(b) What do you mean by decomposition? What are the implications of different types of decompositions?

Ans: Data Decomposition

Data decomposition, also known as data-level parallelism, breaks down tasks by the data they work on rather than by the nature of the task. Programs that are broken down via data decomposition generally have many threads performing the same work, just on different data items. For example, consider recalculating the values in a large spreadsheet. Rather than have one thread perform all the calculations, data decomposition would suggest having two threads, each performing half the calculations, or n threads performing 1/nth the work.

Decomposition	Design	Comments
Task	Different activities assigned to different threads	Common in GUI apps
Data	Multiple threads performing the same operation but on different blocks of data	Common in audio processing, imaging, and in scientific programming
Data Flow	One thread’s output is the input to a second thread	Special care is needed to eliminate startup and shutdown latencies

Task Decomposition

Decomposing a program by the functions that it performs is called task decomposition. It is one of the simplest ways to achieve parallel execution. Using this approach, individual tasks are catalogued. If two of them can run concurrently, they are scheduled to do so by the developer. Running tasks in parallel this way usually requires slight modifications to the individual functions to avoid conflicts and to indicate that these tasks are no longer sequential.

Data Decomposition

Data decomposition, also known as data-level parallelism, breaks down tasks by the data they work on rather than by the nature of the task. Programs that are broken down via data decomposition generally have many threads performing the same work, just on different data items. For example, consider recalculating the values in a large spreadsheet. Rather than have one thread perform all the calculations, data decomposition would suggest having two threads, each performing half the calculations, or n threads performing 1/nth the work.

Data Flow Decomposition

Many times, when decomposing a problem, the critical issue isn’t what tasks should do the work, but how the data flows between the different tasks. In these cases, data flow decomposition breaks up a problem by how data flows between tasks.

(c) Explain the term parallel programming. Also discuss about why we need parallel programming in our architecture.

Ans: Parallel programming refers to the practice of writing computer programs in a way that they can execute multiple tasks or operations simultaneously by utilizing multiple processing units or cores within a computer system. The goal of parallel programming is to improve the efficiency and performance of programs by taking advantage of the inherent parallelism present in modern computer architectures.

In traditional sequential programming, a program's instructions are executed one after another in a single thread of execution. This approach limits the program's ability to fully utilize the computational resources available in today's multi-core processors and high-performance computing systems. Parallel programming, on the other hand, involves breaking down tasks into smaller units that can be executed concurrently, allowing for better resource utilization and faster execution times.

Reasons for Using Parallel Programming:

1. **Performance Improvement:** The primary motivation for parallel programming is to achieve better performance and faster execution times. By distributing work across multiple processing units, programs can complete tasks more quickly, making them more responsive and efficient.
2. **Scalability:** As technology advances, the number of processing cores in modern CPUs continues to increase. Parallel programming allows applications to take advantage of this increasing number of cores, enabling them to scale their performance with the available hardware resources.
3. **Handling Complex Tasks**: Many real-world problems, such as simulations, data analysis, and scientific computations, involve complex calculations that can be divided into smaller subtasks. Parallel programming allows these subtasks to be processed simultaneously, reducing the overall time required to obtain results.
4. **Big Data Processing**: In the era of big data, processing and analyzing massive datasets can be time-consuming. Parallel programming techniques, especially when combined with distributed computing frameworks, enable efficient processing of large datasets by distributing the workload across multiple nodes or machines.
5. **Real-time Processing**: Certain applications, like real-time graphics, multimedia processing, and game engines, require high-speed processing to maintain smooth and responsive interactions. Parallel programming can help meet the computational demands of these applications.
6. **Scientific Simulations:** Scientific simulations and modeling often involve running simulations multiple times with varying parameters. Parallel programming allows these simulations to run concurrently, reducing the time required to analyze different scenarios.
7. **Energy Efficiency**: While it might seem counterintuitive, parallel programming can sometimes lead to energy savings. By completing tasks more quickly, processors can enter low-power modes sooner, reducing overall energy consumption.
8. **Future-Proofing**: As technology advances, parallel computing becomes increasingly prevalent. Developing skills in parallel programming ensures that programmers can leverage the capabilities of emerging hardware architectures.

In summary, parallel programming is essential to fully utilize the computational power of modern hardware architectures. It offers the potential for significant performance improvements, scalability, and efficient utilization of resources across a wide range of applications, from scientific computing to data analysis and real-time processing.

(d) Explain ABA problem in multicore programming with suitable example. Also give solution to ABA problem.

Ans: The ABA problem is a challenging issue that can arise in concurrent programming, especially in environments with multiple threads or cores. It primarily affects situations where threads are working with shared data and need to update it concurrently. The problem gets its name from the scenario where a value changes from A to B and then back to A, potentially leading to unexpected behavior if not properly managed.

Example of the ABA Problem:

Let's consider a simple example of a stack implemented using a linked list in a multicore environment:

1. Initially, the stack is empty.
2. Thread A pushes an item 'X' onto the stack, resulting in the following state:

rustCopy code

```
Top -> X -> NULL
```

3. Meanwhile, Thread B performs a pop operation and removes 'X' from the stack.
4. Thread B pushes 'Y' onto the stack:

rustCopy code
Top -> Y -> NULL

5. Now, Thread A retries its pop operation. It observes that the top element is 'X', which matches its expected value, so it continues with the pop operation.
6. Thread A successfully removes 'X' from the stack:

arduinoCopy code
Top -> NULL

In this example, Thread A did not realize that another thread (Thread B) had intervened between its observations, leading to the ABA problem. The situation that occurred can be summarized as follows:

1. Thread A read 'X' as the top element.
2. Thread B removed 'X' and added 'Y' to the stack.
3. Thread B removed 'Y' and the stack became empty.

Thread A's assumption that 'X' is still the top element, even though it had been removed and added back in the meantime, leads to incorrect behavior. This can result in unintended consequences and program errors.

Solution to the ABA Problem:

To solve the ABA problem, various techniques can be employed to ensure that threads correctly account for changes that occur while they are not actively accessing shared data. One common approach is to use atomic operations with double compare-and-swap (DCAS) or compare-and-swap with linked-list tagging.

These techniques involve associating a timestamp or a version number with each update to shared data. When performing an update, a thread will only succeed if the expected value matches the current value and the timestamp or version number matches as well. This way, if an update occurs between the initial read and the update attempt, the thread will notice the inconsistency and take appropriate action.

In the case of the stack example, the DCAS operation can be used during push and pop operations to ensure that the value being modified has not been altered since the initial observation.

By incorporating such techniques, the ABA problem can be mitigated, allowing concurrent programs to maintain correctness and consistency in shared data access.

(e) What is data race condition? Also explain how you can manage the shared and private data.

Ans: Data race condition is a common issue in concurrent programming where multiple threads or processes access shared data concurrently without proper synchronization mechanisms. It occurs when at least two threads access the same memory location concurrently, and at least one of these accesses is a write operation. Data races can lead to unpredictable and incorrect behavior of programs, making them hard to debug and causing issues such as crashes, corruption of data, and incorrect results.

Data races arise due to the interleaved execution of threads, where the order of their operations is unpredictable. Without proper synchronization mechanisms, threads can step on each other's toes, leading to unexpected behavior.

Managing Shared and Private Data:

When working with concurrent programs, it's crucial to manage shared and private data properly to avoid data races and ensure the correctness of the program's behavior. Here are some strategies to manage shared and private data effectively:

1. Synchronization Mechanisms:

- Locks and Mutexes: These are traditional synchronization mechanisms that ensure only one thread can access a shared resource at a time. Threads need to acquire a lock before accessing the resource and release it afterward.

- Semaphores: Semaphores can be used to control access to a limited number of instances of a resource. They can be used to prevent overuse of a shared resource.

2. Atomic Operations: Atomic operations are operations that are executed as a single unit of work without the possibility of interruption. They can be used to perform read-modify-write operations on shared data without the risk of data races. Examples include atomic increments and atomic compare-and-swap operations.
3. Thread-local Storage: For data that is specific to each thread and doesn't need to be shared, you can use thread-local storage. This ensures that each thread has its own copy of the data, preventing data races.
4. Immutable Data: Using immutable data (data that cannot be changed after creation) can eliminate the need for synchronization in many cases. If data cannot change, there's no risk of concurrent modifications causing issues.
5. Message Passing: In some concurrent programming models, like the actor model, threads communicate by passing messages rather than sharing data directly. This can help avoid data races by ensuring that data is accessed by one thread at a time.
6. Data Partitioning: If possible, partition data so that each thread primarily works on its own portion of the data. This reduces the need for synchronization and can improve performance.
7. Critical Sections: A critical section is a portion of code that must be executed by only one thread at a time. Surrounding critical sections with locks or other synchronization mechanisms prevents concurrent access.

Managing shared and private data effectively is essential to avoid data race conditions and ensure the correctness of concurrent programs. The choice of synchronization mechanism depends on the nature of the data, the programming language and environment, and the specific requirements of the application. Proper synchronization not only prevents data races but also contributes to the overall reliability and performance of concurrent software.

SECTION C

3. (a) Describe the concept of Multithreading on single core and on Multi-core platforms.

Ans: Multithreading is a programming technique that enables a single process to execute multiple threads concurrently. Threads are lightweight, independent units of a process that share the same memory space, allowing them to work together on different tasks simultaneously. The concept of multithreading can be applied to both single-core and multi-core platforms, but the implications and benefits differ significantly between these two scenarios. Let's explore multithreading on both types of platforms:

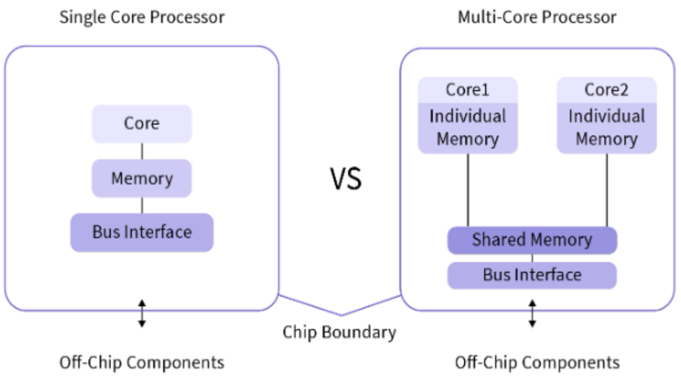
Multithreading on a Single-Core Platform:

On a single-core platform, the operating system (OS) uses a technique known as time-sharing or time-slicing to give the illusion of parallel execution even though there is only one physical processing core. Here's how multithreading works on a single-core platform:

1. Time-Slicing: The CPU time is divided into small time slices. The OS scheduler switches between different threads at a rapid rate, giving each thread a turn to execute for a short period.
2. Concurrency: Although only one thread is executing at any given moment, the frequent switching between threads creates the illusion of concurrency. This is particularly useful when one thread is waiting for I/O operations or other blocked resources.
3. Benefits: Multithreading on a single-core platform can improve responsiveness and the ability to perform multiple tasks simultaneously, even though they are not executed truly in parallel.

4. Drawbacks: True parallelism is not achieved, as only one thread runs at a time. If threads are computationally intensive, the overall performance may not improve significantly because they compete for the same core's processing power.
Multithreading on a Multi-Core Platform:
In a multi-core platform, there are multiple physical processing cores available for concurrent execution.
Multithreading here offers more substantial benefits:
1. True Parallelism: On multi-core platforms, multiple threads can run simultaneously on separate cores. This results in true parallel execution and can significantly boost performance, especially for CPU-bound tasks.
2. Load Balancing: The OS scheduler can distribute threads across available cores to balance the workload and maximize core utilization.
3. Scalability: Multithreading on multi-core platforms allows applications to scale well with the increasing number of cores. As more cores are added, the potential for parallelism and performance improvements increases.
4. Efficiency: On multi-core systems, multithreading is highly efficient for multi-threaded applications, as it takes full advantage of the available hardware resources.

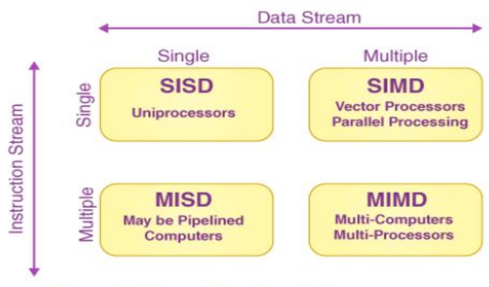
In summary, the concept of multithreading is applicable to both single-core and multi-core platforms, but the benefits and implications differ significantly. On single-core platforms, multithreading offers concurrency and improved multitasking but not true parallelism. On multi-core platforms, multithreading takes full advantage of the available processing cores, achieving true parallelism and offering significant performance improvements, making it a fundamental technique for exploiting modern hardware capabilities.



(b) Explain Flynn’s Taxanomy for Parallel Computing.

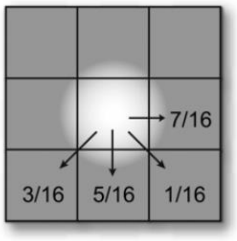
Ans: Parallel computing is computing where the jobs are broken into discrete parts that can be executed concurrently. Each part is further broken down into a series of instructions. Instructions from each piece execute simultaneously on different CPUs. The breaking up of different parts of a task among multiple processors will help to reduce the amount of time to run a program. Parallel systems deal with the simultaneous use of multiple computer resources that can include a single computer with multiple processors, a number of computers connected by a network to form a parallel processing cluster, or a combination of both. Parallel systems are more difficult to program than computers with a single processor because the architecture of parallel computers varies accordingly and the processes of multiple CPUs must be coordinated and synchronized. The difficult problem of parallel processing is portability.
There are four categories in Flynn’s taxonomy:

1. **Single Instruction Single Data (SISD):** In a SISD architecture, there is a single processor that executes a single instruction stream and operates on a single data stream. This is the simplest type of computer architecture and is used in most traditional computers.
2. **Single Instruction Multiple Data (SIMD):** In a SIMD architecture, there is a single processor that executes the same instruction on multiple data streams in parallel. This type of architecture is used in applications such as image and signal processing.
3. **Multiple Instruction Single Data (MISD)**: In a MISD architecture, multiple processors execute different instructions on the same data stream. This type of architecture is not commonly used in practice, as it is difficult to find applications that can be decomposed into independent instruction streams.
4. **Multiple Instruction Multiple Data (MIMD):** In a MIMD architecture, multiple processors execute different instructions on different data streams. This type of architecture is used in distributed computing, parallel processing, and other high-performance computing applications.
Flynn’s taxonomy is a useful tool for understanding different types of computer architectures and their strengths and weaknesses. The taxonomy highlights the importance of parallelism in modern computing and shows how different types of parallelism can be exploited to improve performance.



Flynn’s Classification of Computers

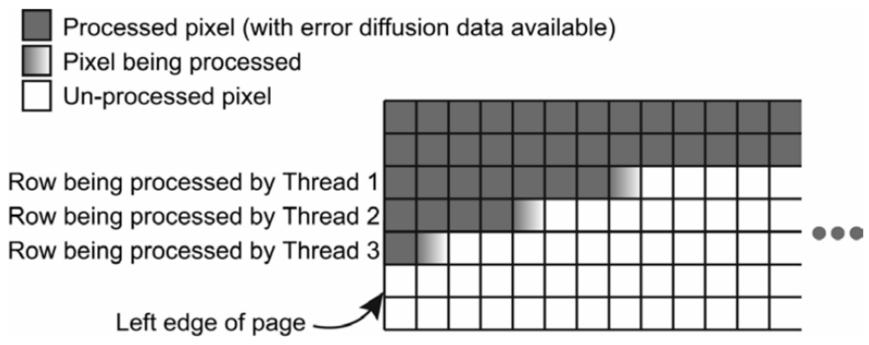
4. (a) Demonstrate Error Diffusion algorithm? Illustrate how it can be parallelized in multi-threaded environment.
Ans: The basic error diffusion algorithm does its work in a simple three step process:
1. Determine the output value given the input value of the current pixel. This step often uses quantization, or in the binary case, thresholding. For an 8-bit grayscale image that is displayed on a 1-bit output device, all input values in the range [0, 127] are to be displayed as a 0 and all input values between [128, 255] are to be displayed as a 1 on the output device.
2. Once the output value is determined, the code computes the error between what should be displayed on the output device and what is actually displayed. As an example, assume that the current input pixel value is 168. Given that it is greater than our threshold value (128), we determine that the output value will be a 1. This value is stored in the output array. To compute the error, the program must normalize output first, so it is in the same scale as the input value. That is, for the purposes of computing the display error, the output pixel must be 0 if the output pixel is 0 or 255 if the output pixel is 1. In this case, the display error is the difference between the actual value that should have been displayed (168) and the output value (255), which is –87.
3. Finally, the error value is distributed on a fractional basis to the neighboring pixels in the region, as shown in Figure



Distributing Error Values to Neighboring Pixels

An Alternate Approach: Parallel Error Diffusion Given that a pixel may not be processed until its spatial predecessors have been processed, the problem appears to lend itself to an approach where we have a producer—or in this case, multiple producers—producing data (error values) which a consumer (the current pixel) will use to compute the proper output pixel. The flow of error data to the current pixel is critical. Therefore, the problem seems to break down into a data-flow decomposition. Now that we identified the approach, the next step is to determine the best pattern that can be applied to this particular problem. Each independent thread of execution should process an equal amount of work (load balancing). How should the work be partitioned? One way, based on the algorithm presented in the previous section, would be to have a thread that processed the even pixels in a given row, and another thread that processed the odd pixels in the same row. This approach is ineffective however; each thread will be blocked waiting for the other to complete,and the performance could be worse than in the sequential case.

.


Processed pixel (with error diffusion data available)
Pixel being processed
Un-processed pixel
Row being processed by Thread 1
Row being processed by Thread 2
Row being processed by Thread 3
Left edge of page

Multiple threads are able to process multiple rows simultaneously.

(b) Illustrate different types of parallel programming patterns.

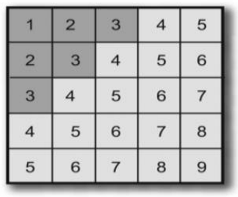
Ans:

Pattern	**Decomposition**
Task-level parallelism	Task
Divide and Conquer	Task/Data
Geometric Decomposition	Data
Pipeline	Data Flow
Wavefront	Data Flow

Brief overview of each pattern and the types of problems that each pattern may be applied to.

1)Task-level Parallelism Pattern. In many cases, the best way to achieve parallel execution is to focus directly on the tasks themselves. In this case, the task-level parallelism pattern makes the most sense. In this pattern, the problem is decomposed into a set of tasks that operate independently. It is often necessary remove dependencies between tasks or separate dependencies

using replication. Problems that fit into this pattern include the so-called embarrassingly parallel problems, those where there are no dependencies between threads, and replicated data problems, those where the dependencies between threads may be removed from the individual threads.
2)Divide and Conquer Pattern. In the divide and conquer pattern, the problem is divided into a number of parallel sub-problems.
Each sub-problem is solved independently. Once each subproblem is solved, the results are aggregated into the final solution. Since each sub-problem can be independently solved, these sub-problems may be executed in a parallel fashion.
The divide and conquer approach is widely used on sequential algorithms such as merge sort. These algorithms are very easy to parallelize. This pattern typically does a good job of load balancing and exhibits good locality; which is important for effective cache usage.
Geometric Decomposition Pattern. The geometric decomposition pattern is based on the parallelization of the data structures 44 Multi-Core Programming used in the problem being solved. In geometric decomposition, each thread is responsible for operating on data 'chunks'. This pattern may be applied to problems such as heat flow and wave propagation.
Pipeline Pattern. The idea behind the pipeline pattern is identical to that of an assembly line. The way to find concurrency here is to break down the computation into a series of stages and have each thread work on a different stage simultaneously.
„ **Wavefront Pattern**. The wavefront pattern is useful when processing data elements along a diagonal in a two-dimensional grid. This is shown in Figure.Figure Wavefront Data Access Pattern
The numbers in Figure 3.1 illustrate the order in which the data elements are processed. For example, elements in the diagonal that contains the number "3" are dependent on data elements
"1" and "2" being processed previously. The shaded data elements in Figure indicate data that has already been processed. In this pattern, it is critical to minimize the idle time spent by each thread. Load balancing is the key to success with this pattern.



Wavefront Data Access Pattern

5. (a) Explain the threading API's for microsoft .NET framework.
Ans: The Microsoft .NET Framework provides a powerful and comprehensive set of threading APIs for managing and working with threads in C# and other .NET languages. These APIs are part of the System.Threading namespace and allow developers to create, control, and synchronize threads within their applications. Here are some of the key threading APIs available in the Microsoft .NET Framework:
1. Thread Class:
- `System.Threading.Thread` is a fundamental class for working with threads in .NET.
- You can create and start threads using the `Thread` class constructor and the `Start` method.
- It provides properties and methods for thread management, such as `Name`, `Priority`, `IsAlive`, `Join`, and `Sleep`.
2. Thread Pooling:
- .NET offers a thread pool that manages a pool of worker threads, which can be used to execute tasks concurrently.
- You can use `ThreadPool.QueueUserWorkItem` to add work items (delegates) to the thread pool for asynchronous execution.
- Thread pool threads are managed by the runtime and are reused, which can help reduce the

overhead of creating and destroying threads.

3. Task Parallel Library (TPL):
   - The TPL is a higher-level threading API introduced in .NET Framework 4.0 and later versions.
   - It simplifies parallelism and asynchronous programming with features like the `Task` class.
   - You can create and manage tasks, use parallel loops (`Parallel.ForEach` and `Parallel.For`), and handle exceptions asynchronously.
4. Parallel LINQ (PLINQ):
   - PLINQ is an extension of LINQ (Language Integrated Query) that enables parallel query execution.
   - It is part of the TPL and allows you to parallelize LINQ queries for improved performance.
   - You can use `AsParallel()` to enable parallel processing of sequences.
5. Synchronization Primitives:
   - .NET provides various synchronization primitives for thread synchronization and coordination:
     - `Monitor`: Allows you to create critical sections using `lock`.
     - `Mutex`: Provides exclusive access to a resource across multiple processes.
     - `Semaphore`: Controls access to a resource with a specified number of permits.
     - `AutoResetEvent` and `ManualResetEvent`: Used for signaling between threads.
     - `CountdownEvent`: Allows one or more threads to wait until a countdown reaches zero.
6. Thread Local Storage (TLS):
   - .NET supports thread-local storage through the `ThreadLocal<T>` class, which allows each thread to have its own instance of a variable.
7. Async and Await:
   - Asynchronous programming is simplified in .NET with the `async` and `await` keywords.
   - You can use these keywords to write asynchronous code that doesn't block the main thread, improving responsiveness in applications.
8. Thread Safety and Concurrent Collections:
   - .NET offers thread-safe collections like `ConcurrentQueue`, `ConcurrentStack`, and `ConcurrentDictionary` that can be used in multi-threaded scenarios without additional synchronization.

(b) Compare and contrast Mutual Exclusive(mutex) and locks.

Ans: Locks Locks are similar to semaphores except that a single thread handles a lock at one instance. Two basic atomic operations get performed on a lock:

1) Acquire (): Atomically waits for the lock state to be unlocked and sets the lock state to lock.
2) Release(): Atomically changes the lock state from locked to unlocked. At most one thread acquires the lock. A thread has to acquire a lock before using a shared resource; otherwise it waits until the lock becomes

available. When one thread wants to access shared data, it first acquires the lock, exclusively performs operations on the shared data and later releases the lock for other threads to use. The level of granularity can be either coarse or fine depending on the type of shared data that needs to be protected from threads. The coarse granular locks have higher lock contention than finer granular ones. To remove issues with lock granularity, most of the processors support the Compare and Swap (CAS) operation, which provides a way to implement lock-free synchronization.

The atomic CAS operations guarantee that the shared data remains synchronized among threads. If you require the use of locks, it is recommended that you use the lock inside a critical section with a single entry and single exit, as shown in code.

```
{define all necessary locks}
<Start multithreading blocks>
...
<critical section start>
<acquire lock L>
.. operate on shared memory protected by lock L ..
<release lock L>
<critical section end>
..
<End multithreading blocks>
```

Lock Types :

An application can have different types of locks according to the constructs required to accomplish the task. The following sections cover these locks and define their purposes.

Mutexes. The mutex is the simplest lock an implementation can use. Some texts use the mutex as the basis to describe locks in general. The release of a mutex does not depend on the release() operation only. A timer attribute can be added with a mutex. If the timer expires before a release operation, the mutex releases the code block or shared memory to other threads. A try-finally clause can be used to make sure that the mutex gets released when an exception occurs. The use of a timer or tryfinally clause helps to prevent a deadlock scenario.

Recursive Locks. Recursive locks are locks that may be repeatedly acquired by the thread that currently owns the lock without causing the thread to deadlock. No other thread may acquire a recursive lock until the owner releases it once for each time the owner acquired it. Thus when using a recursive lock, be sure to balance acquire operations with release operations. The best way to do this is to lexically balance the operations around single-entry single-exit blocks, as was shown for ordinary locks. The recursive lock is most useful inside a recursive function. In general, the recursive locks are slower than non recursive locks.

Read-Write Locks. Read-Write locks are also called shared-exclusive or multiple-read/single-write locks or non-mutual exclusion semaphores. Read-write locks allow simultaneous read access to multiple threads but limit the write access to only one thread. This type of lock can be used efficiently for those instances where multiple threads need to read shared data simultaneously but do not necessarily need to perform a write operation. For lengthy shared data, it is sometimes better to break the data into smaller segments and operate multiple read-write locks on the dataset rather than having a data lock for a longer period of time.

Spin Locks. Spin locks are non-blocking locks owned by a thread. Waiting threads must "spin," that is, poll the state of a lock rather than get blocked. Spin locks are used mostly on multiprocessor systems. This is because while the thread spins in a single-core processor system, no process resources are available to run the other thread that will release the lock. The appropriate condition for using spin locks is whenever the hold time of a lock is less than the time of blocking and waking up a thread. The change of control for threads involves context switching of threads and updating thread data structures, which could require more instruction cycles than spin locks.

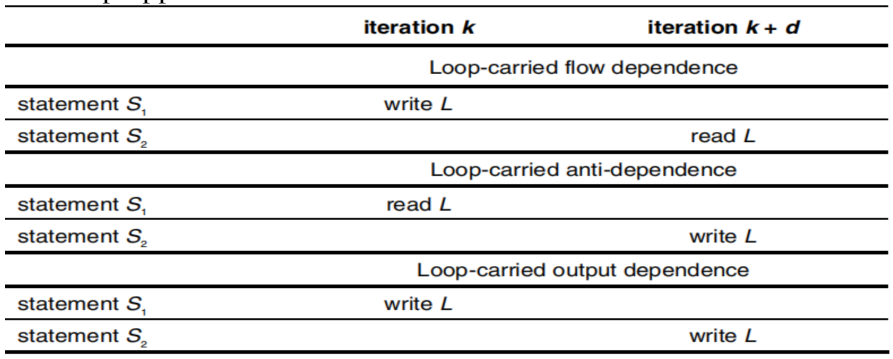
6. (a) What are the challenges in threading a loop? Explain each with an example.

Ans: Threading a loop is to convert independent loop iterations to threads and run these threads in parallel. In some sense, this is a re-ordering transformation in which the original order of loop iterations can be converted to into an undetermined order. In addition, because the loop body is not an atomic operation, statements in the two different iterations may run simultaneously. In theory, it is valid to convert a sequential loop to a threaded loop if the loop carries no dependence.

Therefore, the first challenge for you is to identify or restructure the hot loop to make sure that it has no loop-carried dependence before adding
OpenMP pragmas.

Loop-carried Dependence

Even if the loop meets all five loop criteria and the compiler threaded the loop, it may still not work correctly, given the existence of data dependencies that the compiler ignores due to the presence of OpenMP pragmas. The theory of data dependence imposes two requirements 138
Multi-Core Programming that must be met for a statement S2 and to be data dependent on
statement S1.
There must exist a possible execution path such that statement S1 and S2 both reference the same memory location L.
The execution of S1 that references L occurs before the execution of S2 that references L.
In order for S2 to depend upon S1 , it is necessary for some execution of S1 to write to a memory location L that is later read by an execution of S2 . This is also called flow dependence. Other dependencies exist when two statements write the same memory location L, called an output
dependence, or a read occurs before a write, called an anti-dependence.
This pattern can occur in one of two ways:
S1 can reference the memory location L on one iteration of a loop; on a subsequent iteration S2 can reference the same memory location L.
S1 and S2 can reference the same memory location L on the same loop iteration, but with S1 preceding S2 during execution of the loop iteration. The first case is an example of loop-carried dependence, since the dependence exists when the loop is iterated. The second case is an example of loop-independent dependence; the dependence exists because of the position of the code within the loops. Table 6.1 shows three cases of loop-carried dependences with dependence distance d, where 1 ≤ d ≤ n, and n is the loop upper bound.



	iteration *k*	**iteration *k* + *d***
	Loop-carried flow dependence	
statement S_1	write L	
statement S_2		read L
	Loop-carried anti-dependence	
statement S_1	read L	
statement S_2		write L
	Loop-carried output dependence	
statement S_1	write L	
statement S_2		write L

Data-race Conditions

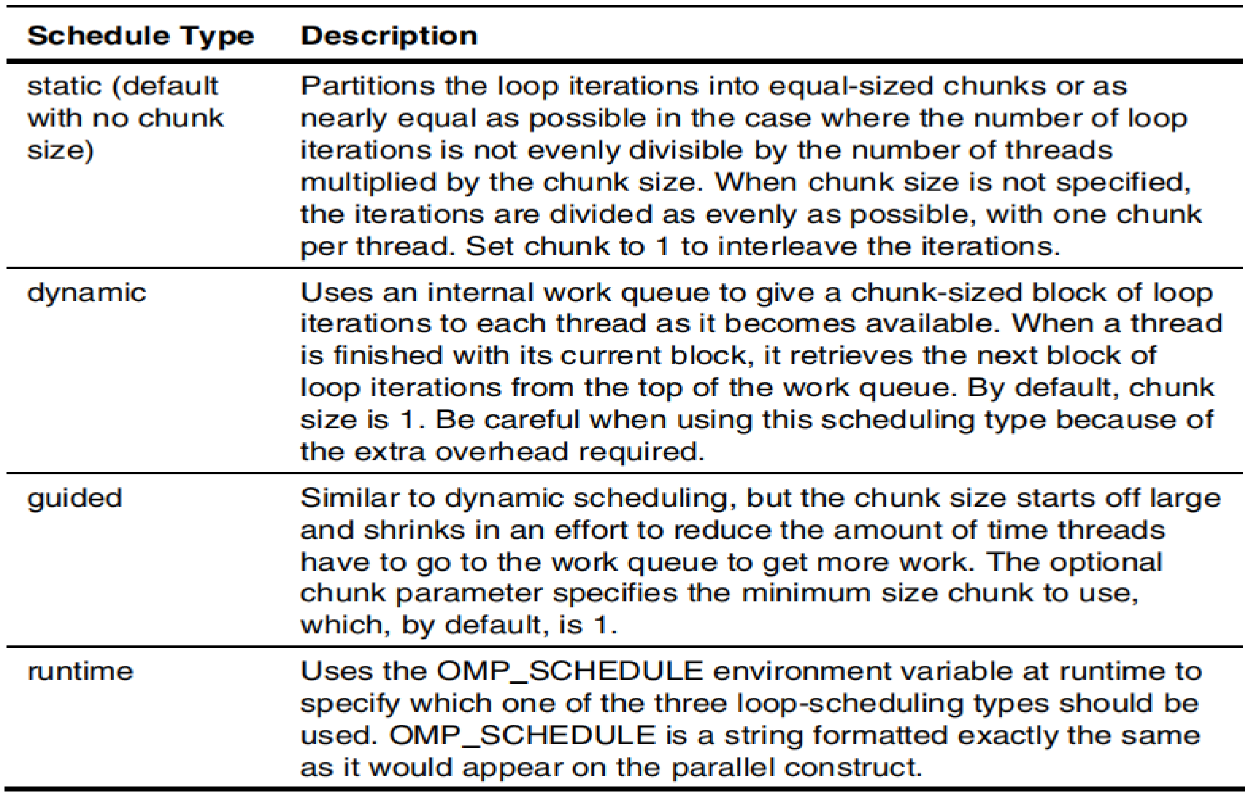
Data-race conditions that are mentioned in the previous chapters could be due to output dependences, in which multiple threads attempt to update the same memory location, or variable, after threading. In general, the OpenMP C++ and Fortran compilers do honor OpenMP pragmas or directives while encountering them during compilation phase, however, the compiler does not perform or ignores the detection of data-race conditions. Thus, a loop similar to the following example, in which multiple threads are updating the variable x will lead to undesirable results. In such a situation, the code needs to be modified via privatization or synchronized using mechanisms like mutexes.

Managing Shared and Private Data

In writing multithreaded programs, understanding which data is shared and which is private becomes extremely important, not only to performance, but also for program correctness. OpenMP makes this distinction apparent to the programmer through a set of clauses such as
shared, private, and default, and it is something that you can set manually. With OpenMP, it is the developer's responsibility to indicate to the compiler which pieces of memory should be shared among the threads and which pieces should be kept private. When memory is identified as shared, all threads access the exact same memory location. When memory is identified as private, however, a separate copy of the variable is made for each thread to access in private. When the loop exits, these private copies become undefined. By default, all the variables in a parallel region are shared, with three exceptions.

Loop Scheduling and Partitioning
To have good load balancing and thereby achieve optimal performance in a multithreaded application, you must have effective loop scheduling and partitioning. The ultimate goal is to ensure that the execution cores are busy most, if not all, of the time, with minimum overhead of scheduling, context switching and synchronization.

Table 6.2 The Four Schedule Schemes in OpenMP



Schedule Type	Description
static (default with no chunk size)	Partitions the loop iterations into equal-sized chunks or as nearly equal as possible in the case where the number of loop iterations is not evenly divisible by the number of threads multiplied by the chunk size. When chunk size is not specified, the iterations are divided as evenly as possible, with one chunk per thread. Set chunk to 1 to interleave the iterations.
dynamic	Uses an internal work queue to give a chunk-sized block of loop iterations to each thread as it becomes available. When a thread is finished with its current block, it retrieves the next block of loop iterations from the top of the work queue. By default, chunk size is 1. Be careful when using this scheduling type because of the extra overhead required.
guided	Similar to dynamic scheduling, but the chunk size starts off large and shrinks in an effort to reduce the amount of time threads have to go to the work queue to get more work. The optional chunk parameter specifies the minimum size chunk to use, which, by default, is 1.
runtime	Uses the OMP_SCHEDULE environment variable at runtime to specify which one of the three loop-scheduling types should be used. OMP_SCHEDULE is a string formatted exactly the same as it would appear on the parallel construct.

(b) What is OpenMP. Explain OpenMP library functions and environment variable.
Ans: The OpenMP standard was formulated in 1997 as an API for writing portable, multithreaded applications. It started as a Fortran-based standard, but later grew to include C and C++. The current version is OpenMP Version 2.5, which supports Fortran, C, and C++. Intel C++ and Fortran compilers support the OpenMP Version 2.5 standard (www.openmp.org). The OpenMP programming model provides a platform-independent set of compiler pragmas, directives, function calls, and environment variables that explicitly instruct the compiler how and where to use parallelism in the application. Many loops can be threaded by inserting only one pragma right before the loop, as demonstrated by examples in this chapter. By leaving the nitty-gritty details to the compiler and OpenMP runtime library, you can spend more time determining which loops should be threaded and how to best restructure the algorithms for performance on multi-core processors.

Library Functions:

OpenMP provides a set of library functions that can be called from within your parallel programs to control and query the execution of threads and parallel regions. Some of the commonly used OpenMP library functions include:

1. omp_get_num_threads(): This function returns the number of threads in the current team (parallel region).
2. omp_get_thread_num(): It returns the thread number of the calling thread within the current team.

3. omp_set_num_threads(int num_threads): You can use this function to set the number of threads for subsequent parallel regions.
4. omp_get_max_threads(): This function returns the maximum number of threads that can be used in parallel regions.
5. omp_get_num_procs(): It returns the number of available processors on the system.
6. omp_get_dynamic(): This function queries whether dynamic adjustment of the number of threads is enabled or disabled.
7. omp_set_dynamic(int dynamic): You can use this function to enable or disable dynamic thread adjustment.
8. omp_get_wtime(): This function returns the wall-clock time in seconds, which can be used for measuring execution time.
9. omp_get_wtick():It returns the timer resolution in seconds.
These library functions allow you to control and query various aspects of your parallel program's execution, such as the number of threads, timing, and thread-specific information.

Environment Variables:

OpenMP also provides environment variables that allow you to control the behavior of your parallel program without modifying the source code. Some of the important OpenMP environment variables include:

1. OMP_NUM_THREADS: This variable sets the default number of threads for all parallel regions in your program. For example, you can set it to control how many threads should be used without modifying your code.
2. OMP_SCHEDULE: It determines how loop iterations are scheduled among threads in a parallel loop. Common scheduling options include static, dynamic, and guided scheduling.
3. OMP_DYNAMIC: This variable controls whether the number of threads can be adjusted dynamically during program execution.
4. OMP_NESTED: If set to "true," it enables nested parallelism, allowing you to create parallel regions within other parallel regions.
5. OMP_PROC_BIND: It controls the binding of threads to processors or cores. Options include "true," "false," and "spread."
6. OMP_WAIT_POLICY: Specifies the waiting policy for threads in a parallel region when they are idle.

These environment variables provide a way to fine-tune the behavior of your OpenMP program without modifying the source code, which can be particularly useful for performance optimization.

In summary, OpenMP library functions and environment variables are essential components of the OpenMP API, allowing you to control and optimize the parallel execution of your programs. They provide flexibility and ease of use when developing parallel software.

7. (a) Explain Current IA-32 Architecture in detail.
Ans: IA-32 approximates sequential consistency, because it evolved in the single-core age. The virtue is how IA-32 preserves legacy software. Extreme departures from sequential consistency would have broken old code. However, adhering to sequential consistency would have yielded
poor performance, so a balance had to be struck. For the most part, the balance yields few surprises, yet achieves most of the possible performance improvements (Hill 1998). Two rules cover typical programming:

■ Relaxation for performance. A thread sees other threads' reads and writes in the original order, except that a read may pass a write to a different location. This reordering rule allows a thread
to read from its own cache even if the read follows a write to main memory. This rule does not cover "nontemporal" writes, which are discussed later.
■ Strictness for correctness. An instruction with the LOCK prefix acts as a memory fence. No read or write may cross the fence. This rule stops relaxations from breaking typical synchronization
idioms based on the LOCK instructions. Furthermore, the instruction xchg has an implicit LOCK prefix in order to preserve old code written before the LOCK prefix was introduced. This slightly relaxed memory consistency is called processor order. For efficiency, the IA-32 architecture also allows loads to pass loads but hides this from the programmer. But if the processor detects that the
reordering might have a visible effect, it squashes the affected instructions and reruns them. Thus the only visible relaxation is that reads can pass writes. The IA-32 rules preserve most idioms, but ironically break the textbook algorithm for mutual exclusion called Dekker's Algorithm1. This algorithm enables mutual exclusion for processors without special atomic instructions. Figure 7.23(a) demonstrates the key sequence in Dekker's Algorithm. Two variables X and Y are initially zero. Thread 1 1 The first published software-only, two-process mutual exclusion algorithm. 206
Multi-Core Programming writes X and reads Y. Thread 2 writes Y and reads X. On a sequentially consistent machine, no matter how the reads and writes are interleaved, no more than one of the threads reads a zero. The thread reading the zero is the one allowed into the exclusion region. On IA-32, and just about every other modern processor, both threads might read 0, because the reads might pass the writes.

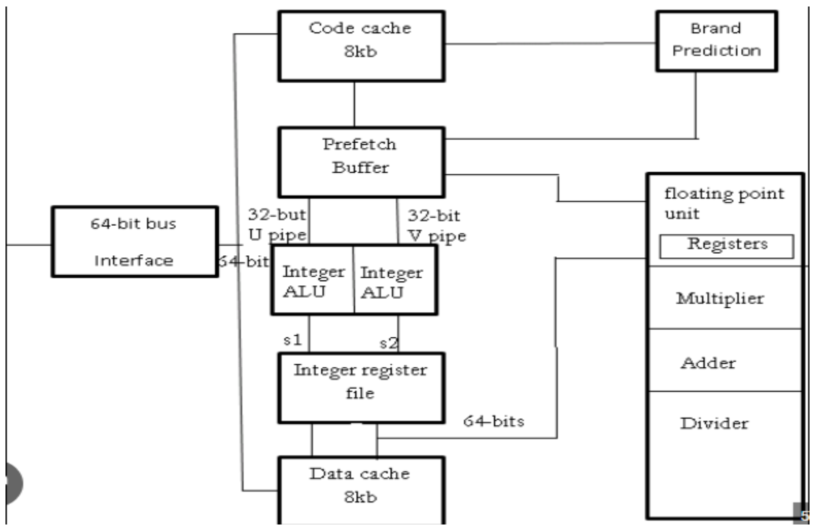


Table 7.1 Types of IA-32 Fence Instructions

Mnemonic	Name	Description
mfence	Memory **fence**	neither reads nor writes may cross
lfence	load **fence**	reads may not cross
sfence	Store **fence**	writes may not cross

(b) Describe types of common parallel programming problems.

Ans: Deadlock

Race conditions are typically cured by adding a lock that protects the invariant that might otherwise be violated by interleaved operations. Unfortunately, locks have their own hazards, most notably deadlock. Though deadlock is often associated with locks, it can happen any time a thread tries to acquire exclusive access to two more shared resources. For example, the locks in could be files

instead, where the threads are trying to acquire exclusive file access. Deadlock can occur only if the following four conditions hold true:
1. Access to each resource is exclusive.
2. A thread is allowed to hold one resource while requesting
another.
3. No thread is willing to relinquish a resource that it has acquired.
4. There is a cycle of threads trying to acquire resources, where each resource is held by one thread and requested by another.

Deadlock can be avoided by breaking any one of these conditions. Often the best way to avoid deadlock is to replicate a resource that requires exclusive access, so that each thread can have its own private copy. Each thread can access its own copy without needing a lock. The copies can be merged into a single shared copy of the resource at the end if necessary. By eliminating locking, replication avoids deadlock and has the further benefit of possibly improving scalability, because the lock that was removed might have been a source of contention. If replication cannot be done, that is, in such cases where there really must be only a single copy of the resource, common wisdom is to always acquire the resources (locks) in the same order. Consistently ordering acquisition prevents deadlock cycles. For instance, the deadlock cannot occur if threads always acquire lock A before they acquire lock B.

For multiple locks in a data structure, the order is often based on the topology of the structure. In a linked list, for instance, the agreed upon order might be to lock items in the order they appear in the list. In a tree structure, the order might be a pre-order traversal of the tree. Somewhat similarly, components often have a nested structure, where bigger components are built from smaller components. For components nested that way, a common order is to acquire locks in order from the outside to the inside.

Heavily Contended Locks

Proper use of lock to avoid race conditions can invite performance problems if the lock becomes highly contended. The lock becomes like a tollgate on a highway. If cars arrive at the tollgate faster than the toll taker can process them, the cars will queue up in a traffic jam behind the tollgate. Similarly, if threads try to acquire a lock faster than the rate at which a thread can execute the corresponding critical section, then program performance will suffer as threads will form a “convoy” waiting to acquire the lock. Indeed, this behavior is sometimes referred to as convoying.

Non-blocking Algorithms : One way to solve the problems introduced by locks is to not use locks. Algorithms designed to do this are called non-blocking. The defining characteristic of a non-blocking algorithm is that stopping a thread does not prevent the rest of the system from making progress. There are different non-blocking guarantees:
■ Obstruction freedom. A thread makes progress as long as there is no contention, but live lock is possible. Exponential backoff can be used to work around live lock.
■ Lock freedom. The system as a whole makes progress.
■ Wait freedom. Every thread makes progress, even when faced with contention. Very few non-blocking algorithms achieve this. Non-blocking algorithms are immune from lock contention, priority inversion, and convoying. Non-blocking algorithms have a lot of advantages, but with these come a new set of problems that need to be understood.