

Multi - Core Architectures and Programming

www.BrainKart.com

[Click Here !!!](#) for [Multi - Core Architectures and Programming](#) full study material.

[Click Here !!!](#) for [other subjects](#) (Anna University)

[Click Here !!!](#) for [Anna University Notes](#) Android App.

[Click Here !!!](#) for [BrainKart](#) Android App.

CS6801 – MULTICORE ARCHITECTURES AND PROGRAMMING

2 MARKS Q & A

UNIT – I – MULTI CORE PROCESSORS

1) What is the purpose of multi-core integrated circuits?

Because of physical limitations the rate of performance improvement in conventional processors is decreasing. In order to increase the power of processors, chipmakers have turned to multicore integrated circuits, that is, integrated circuits with multiple conventional processors on a single chip.

2) Does Serial programs exploit the presence of multiple cores?

Ordinary serial programs, which are programs written for a conventional singlecore processor, usually cannot exploit the presence of multiple cores, and it's unlikely that translation programs will be able to shoulder all the work of parallelizing serial programs, meaning converting them into parallel programs, which can make use of multiple cores.

3) How to write Parallel programs?

When we write parallel programs, we usually need to coordinate the work of the cores. This can involve communication among the cores, load balancing, and synchronization of the cores.

4) Why we need ever increasing performance?

The vast increases in computational power that we've been enjoying for decades now have been at the heart of many of the most dramatic advances in fields as diverse as science, the Internet, and entertainment. For example, decoding the human genome, ever more accurate medical imaging, astonishingly fast and accurate Web searches, and ever more realistic computer games would all have been impossible without these increases.

5) List of problems considering the increases in performance of multi core architectures?

- Climate modeling
- Protein folding
- Drug discovery
- Energy research
- Data analysis

6) Why we are building Parallel systems?

Much of the tremendous increase in single processor performance has been driven by the ever-increasing density of transistors—the electronic switches—on integrated circuits. As the size of transistors decreases, their speed can be increased, and the overall speed of the integrated circuit can be increased. However, as the speed of transistors increases, their power consumption also increases. Most of this power is dissipated as heat, and when an integrated circuit gets too hot, it becomes unreliable.

How then, can we exploit the continuing increase in transistor density? The answer is *parallelism*. Rather than building ever-faster, more complex, monolithic processors, the industry has decided to put multiple, relatively simple, complete processors on a single chip. Such integrated circuits are called **multicore** processors.

7) Why we need to write Parallel Programs?

Most programs that have been written for conventional, single-core systems cannot exploit the presence of multiple cores. We can run multiple instances of a program on a multicore system, but this is often of little help. For example, being able to run multiple instances of our favorite game program isn't really what we want—we want the program to run faster with more realistic graphics. In order to do this, we need to either rewrite our serial programs so that they're *parallel*, so that they can make use of multiple cores.

8) Give an Example for Serial Code and Parallel Code.

Serial Code:

```
sum = 0;
for (i = 0; i < n; i++) {
    x = Compute next value(. . .);
    sum += x; }
```

Parallel Code :

```
my sum = 0;
my first i = . . . ; my last i = . . . ;
for (my i = my first i; my i < my last i;
    my i++) {
    my x = Compute next value(. . .);
    my sum += my x; }
```

9) How do we write parallel programs?

There are a number of possible answers to this question, but most of them depend on the basic idea of *partitioning* the work to be done among the cores. There are two widely used approaches: **task-parallelism** and **data-parallelism**. In task-parallelism, we partition the various tasks carried out in solving the problem among the cores. In data-parallelism, we partition the data used in solving the problem among the cores, and each core carries out more or less similar operations on its part of the data.

10) What are the three different extensions to C Language?

Basics of programming parallel computers using the C language and three different extensions to C: the **Message-Passing Interface** or **MPI**, **POSIX threads** or **Pthreads**, and **OpenMP**. MPI and Pthreads are libraries of type definitions, functions, and macros that can be used in C programs. OpenMP consists of a library and some modifications to the C compiler.

11) What are the two extensions to shared memory?

Pthreads and OpenMP were designed for programming shared-memory systems. They provide mechanisms for accessing shared-memory locations. OpenMP is a relatively high-level extension to C. For example, it can “parallelize” our addition loop with a single directive, while Pthreads requires that we do something similar to our example. On the other hand, Pthreads provides some coordination constructs that are unavailable in OpenMP. OpenMP allows us to parallelize many programs with relative ease, while Pthreads provides us with some constructs that make other programs easier to parallelize.

12) What is the extension to distributed memory?

MPI, on the other hand, was designed for programming distributed-memory systems. It provides mechanisms for sending messages.

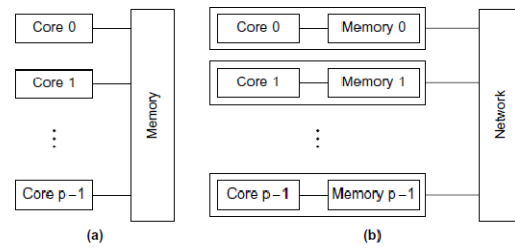


FIGURE 1.2

(a) A shared-memory system and (b) a distributed-memory system

13) Define the types of computing.

There are three types of computing: Concurrent , parallel and distributed computing.

- In concurrent computing, a program is one in which multiple tasks can be *in progress* at any instant
- In parallel computing, a program is one in which multiple tasks *cooperate closely* to solve a problem.
- In distributed computing, a program may need to cooperate with other programs to solve a problem.

14) Define Von Neumann Architecture

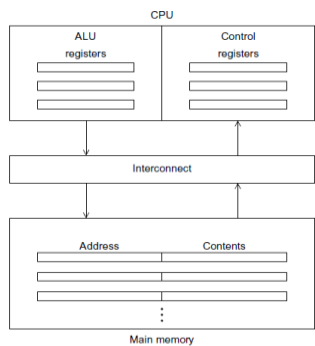


FIGURE 2.1

The von Neumann architecture

The “classical” **von Neumann architecture** consists of **main memory**, a **central processing unit (CPU)** or **processor** or **core**, and an **interconnection** between the memory and the CPU. Main memory consists of a collection of locations, each of which is capable of storing both instructions and data. Every location consists of an address, which is used to access the location and the contents of the location—the instructions or data stored in the location.

15) Define Operating systems?

operating system, or OS, is a major piece of software whose purpose is to manage hardware and software resources on a computer. It determines which programs can run and when they can run. It also controls the allocation of memory to running programs and access to peripheral devices such as hard disks and network interface cards.

16) Define Process

When a user runs a program, the operating system creates a **process**—an instance of a computer program that is being executed.

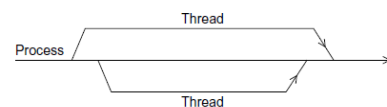


FIGURE 2.2

A process and two threads

17) List Process Entities.

- Executable machine language program
- Call Stack
- Heap
- Descriptors of resources
- Security Information
- Information about state of process

18) Define Multitasking.

Most modern operating systems are **multitasking**. This means that the operating system provides support for the apparent simultaneous execution of multiple programs. This is possible even on a system with a single core, since each process runs for a small interval of time (typically a few milliseconds), often called a **time slice**.

After one running program has executed for a time slice, the operating system can run a different program. A multitasking OS may change the running process many times a minute, even though changing the running process can take a long time.

19) Define Blocks.

In a multitasking OS if a process needs to wait for a resource—for example, it needs to read data from external storage—it will **block**. This means that it will stop executing and the operating system can run another process. However, many programs can continue to do useful work even though the part of the program that is currently executing must wait on a resource. For example, an airline reservation system that is blocked waiting for a seat map for one user could provide a list of available flights to another user.

20) Define Threads.

Threading provides a mechanism for programmers to divide their programs into more or less independent tasks with the property that when one thread is blocked another thread can be run. Furthermore, in most systems it's possible to switch between threads much faster than it's possible to switch between processes. This is because threads are “lighter weight” than processes. Threads are contained within processes, so they can use the same executable, and they usually share the same memory and the same I/O devices. In fact, two threads belonging to one process can share most of the process' resources.

21) Define Cache Miss and Cache Hit.

When a cache is checked for information and the information is available, it's called a **cache hit** or just a hit. If the information isn't available, it's called a **cache miss or a miss**. Hit or miss is often modified by the level. For example, when the CPU attempts to access a variable, it might have an L1 miss and an L2 hit.

22) Define Fully Associative cache.

A cache where data from any address can be stored in any cache location. The whole address must be used as the tag. All tags must be compared simultaneously (associatively) with the requested address and if one matches then its associated data is accessed. This requires an associative memory to hold the tags which makes this form of cache more expensive.

23) Define direct mapped cache.

Direct mapped cache - A cache where the cache location for a given address is determined from the middle address bits. If the cache line size is 2^n then the bottom n address bits correspond to an offset within a cache entry.

24) What is virtual memory?

Virtual memory was developed so that main memory can function as a cache for secondary storage. It exploits the principle of spatial and temporal locality by keeping in main memory only the active parts of the many running programs.

25) Define ILP.

Instruction-level parallelism, or ILP, attempts to improve processor performance by having multiple processor components or **functional units** simultaneously executing instructions. There are two main approaches to ILP: **pipelining**, in which functional units are arranged in stages, and **multiple issue**, in which multiple instructions can be simultaneously initiated. Both approaches are used in virtually all modern CPUs.

26) Define TLP.

Thread-level parallelism, or TLP, attempts to provide parallelism through the simultaneous execution of different threads, so it provides a **coarser-grained** parallelism than ILP, that is, the program units that are being simultaneously executed—threads—are larger or coarser than the **finer-grained** units—individual instructions.

27) What is hardware multi-threading?

Hardware multithreading provides a means for systems to continue doing useful work when the task being currently executed has stalled—for example, if the current task has to wait for data to be loaded from memory.

28) Define Fine grained multithreading.

In **fine-grained** multithreading, the processor switches between threads after each instruction, skipping threads that are stalled. While this approach has the potential to avoid wasted machine time due to stalls, it has the drawback that a thread that's ready to execute a long sequence of instructions may have to wait to execute every instruction.

29) Define Coarse grained multithreading?

Coarse-grained multithreading attempts to avoid this problem by only switching threads that are stalled waiting for a time-consuming operation to complete (e.g., a load from main memory). This has the

virtue that switching threads doesn't need to be nearly instantaneous. However, the processor can be idled on shorter stalls, and thread switching will also cause delays.

30) What is simultaneous multithreading?

Simultaneous multithreading, or SMT, is a variation on fine-grained multithreading. It attempts to exploit superscalar processors by allowing multiple threads to make use of the multiple functional units. If we designate “preferred” threads— threads that have many instructions ready to execute—we can somewhat reduce the problem of thread slowdown.

31) Define SISD and SIMD.

a **single instruction stream, single data stream**, or SISD system, since it executes a single instruction at a time and it can fetch or store one item of data at a time.

Single instruction, multiple data, or SIMD, systems are parallel systems. As the name suggests, SIMD systems operate on multiple data streams by applying the same instruction to multiple data items, so an abstract SIMD system can be thought of as having a single control unit and multiple ALUs. An instruction is broadcast from the control unit to the ALUs, and each ALU either applies the instruction to the current data item, or it is idle.

32) What is vector Processor?

Although what constitutes a vector processor has changed over the years, their key characteristic is that they can operate on arrays or *vectors* of data, while conventional CPUs operate on individual data elements or *scalars*.

33) List the characteristics of vector processors?

- Vector registers
- Vectorized and pipelined functional units.
- Vector instructions
- Interleaved memory
- Strided memory access
- Hardware Scatter/Gather

34) Define Strided memory access.

In *strided memory access*, the program accesses elements of a vector located at fixed intervals. For example, accessing the first element, the fifth element, the ninth element, and so on, would be strided access with a stride of four.

35) Define Hardware Scatter/Gather.

Scatter/gather (in this context) is writing (scatter) or reading (gather) elements of a vector located at irregular intervals— for example, accessing the first element, the second element, the fourth element, the eighth element, and so on. Typical vector systems provide special hardware to accelerate strided access and scatter/gather.

36) Is Vector Processors Scalable?

Vector systems have very high memory bandwidth, and every data item that's loaded is actually used, unlike cache-based systems that may not make use of every item in a cache line. On the other hand, they don't handle irregular data structures as well as other parallel architectures, and there seems to be a very finite limit to their **scalability**, that is, their ability to handle ever larger problems.

37) What are graphical processing Units?

Real-time graphics application programming interfaces, or APIs, use points, lines, and triangles to internally represent the surface of an object. They use a **graphics processing pipeline** to convert the internal representation into an array of pixels that can be sent to a computer screen.

38) What are shader functions?

Several of the stages of this pipeline are programmable. The behavior of the programmable stages is specified by functions called **shader functions**. The shader functions are typically quite short—often just a few lines of C code. They're also implicitly parallel, since they can be applied to multiple elements (e.g., vertices) in the graphics stream.

39) Define MIMD Systems.

Multiple instruction, multiple data, or MIMD, systems support multiple simultaneous instruction streams operating on multiple data streams. Thus, MIMD systems typically consist of a collection of fully independent processing units or cores, each of which has its own control unit and its own ALU. Furthermore, unlike SIMD systems, MIMD systems are usually **asynchronous**, that is, the processors can operate at their own pace.

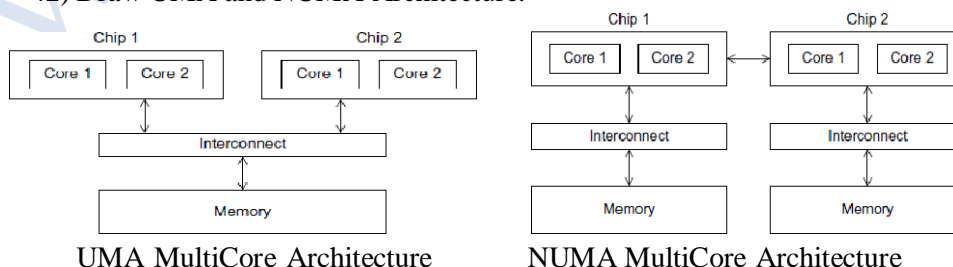
40) What are Shared Memory Systems?

In a **shared-memory system** a collection of autonomous processors is connected to a memory system via an interconnection network, and each processor can access each memory location. In a shared-memory system, the processors usually communicate implicitly by accessing shared data structures.

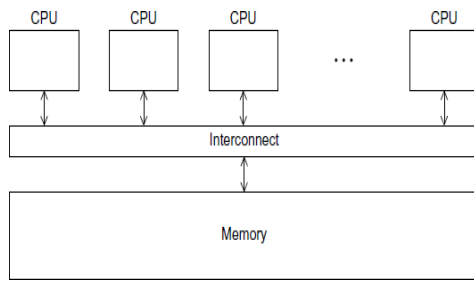
41) What are Distributed Memory Systems?

In a **distributed-memory system**, each processor is paired with its own *private* memory, and the processor-memory pairs communicate over an interconnection network. So in distributed-memory systems the processors usually communicate explicitly by sending messages or by using special functions that provide access to the memory of another processor.

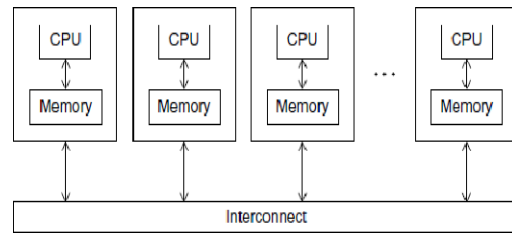
42) Draw UMA and NUMA Architecture.



43) Draw the SMA and DMA.



Shared Memory Architecture



Distributed Memory Architecture

44) What are UMA and NUMA?

In the first type of system, the time to access all the memory locations will be the same for all the cores, while in the second type a memory location to which a core is directly connected can be accessed more quickly than a memory location that must be accessed through another chip. Thus, the first type of system is called a **uniform memory access**, or UMA, system, while the second type is called a **nonuniform memory access**, or NUMA, system.

45) What are Hybrid Systems?

The **nodes** of these systems, the individual computational units joined together by the communication network, are usually shared-memory systems with one or more multicore processors. To distinguish such systems from pure distributed-memory systems, they are sometimes called **hybrid systems**.

46) Define the type of interconnects.

- Switch interconnects
- Crossbar interconnects
- Direct interconnect – hypercube
- Indirect interconnect – crossbar , omega network
- Toroidal mesh
- Ring interconnect
- Fully connected network

47) Define 2 figures used to measure the performance of interconnects.

There are two figures that are often used to describe the performance of an interconnect (regardless of what it's connecting): the **latency** and the **bandwidth**.

48) Define Latency and Bandwidth with Example.

The latency is the time that elapses between the source's beginning to transmit the data and the destination's starting to receive the first byte. The bandwidth is the rate at which the destination receives data after it has started to receive the first byte. So if the latency of an interconnect is l seconds and the bandwidth is b bytes per second, then the time it takes to transmit a message of n bytes is
 message transmission time = $l + n/b$

49) Define bandwidth and bisection bandwidth

The **bandwidth** of a link is the rate at which it can transmit data. It's usually given in megabits or megabytes per second. **Bisection bandwidth** is often used as a measure of network quality. It's similar to bisection width. However, instead of counting the number of links joining the halves, it sums the bandwidth of the links.

50) Define Cache Coherence Problem.

There are several problems here, but the one we want to look at right now is that the caches we described for single processor systems provide no mechanism for insuring that when the caches of multiple processors store the same variable, an update by one processor to the cached variable is "seen" by the other processors. That is, that the cached value stored by the other processors is also updated. This is called the **cache coherence** problem.

51) What are the 2 main approaches to cache coherence?

There are two main approaches to insuring cache coherence: **snooping cache coherence** and directory-based cache coherence.

52) Describe working of snooping cache coherence.

The idea behind snooping comes from bus-based systems: When the cores share a bus, any signal transmitted on the bus can be "seen" by all the cores connected to the bus. Thus, when core 0 updates the copy of x stored in its cache, if it also broadcasts this information across the bus, and if core 1 is "snooping" the bus, it will see that x has been updated and it can mark its copy of x as invalid. This is more or less how snooping cache coherence works.

53) What is the disadvantage of snooping cache coherence?

In large networks broadcasts are expensive, and snooping cache coherence requires a broadcast every time a variable is updated, so snooping cache coherence isn't scalable, because for larger systems it will cause performance to degrade.

54) Describe Directory based coherence.

Directory-based cache coherence protocols attempt to solve this problem through the use of a data structure called a **directory**. The directory stores the status of each cache line. Typically, this data structure is distributed; in our example, each core/memory pair might be responsible for storing the part of the structure that specifies the status of the cache lines in its local memory.

55) Define False Sharing.

So if n is large, we would expect that a large percentage of the assignments $y[i] += f(i,j)$ will access main memory—in spite of the fact that core 0 and core 1 never access each others' elements of y . This is called **false sharing**, because the system is behaving *as if* the elements of y were being shared by the cores. Note that false sharing does not cause incorrect results.

56) Define Shared and Private Variables.

In shared-memory programs, variables can be **shared** or **private**. Shared variables can be read or written by any thread, and private variables can ordinarily only be accessed by one thread. Communication among the threads is usually done through shared variables, so communication is implicit, rather than explicit.

57) Define Dynamic threads.

Shared-memory programs use **dynamic threads**. In this paradigm, there is often a master thread and at any given instant a (possibly empty) collection of worker threads. The master thread typically waits for work requests—for example, over a network—and when a new request arrives, it forks a worker thread, the thread carries out the request, and when the thread completes the work, it terminates and joins the master thread.

58) Define Shared Threads.

An alternative to the dynamic paradigm is the **static thread** paradigm. In this paradigm, all of the threads are forked after any needed setup by the master thread and the threads run until all the work is completed. After the threads join the master thread, the master thread may do some cleanup (e.g., free memory) and then it also terminates. In terms of resource usage, this may be less efficient: if a thread is idle, its resources (e.g., stack, program counter, and so on.) can't be freed.

59) When does non-determinism occur?

In any MIMD system in which the processors execute asynchronously it is likely that there will be **nondeterminism**. A computation is nondeterministic if a given input can result in different outputs. If multiple threads are executing independently, the relative rate at which they'll complete statements varies from run to run, and hence the results of the program may be different from run to run.

60) What is race condition?

The nondeterminism here is a result of the fact that two threads are attempting to more or less simultaneously update the memory location x . When threads or processes attempt to simultaneously access a resource, and the accesses can result in an error, we often say the program has a **race condition**, because the threads or processes are in a "horse race." That is, the outcome of the computation depends on which thread wins the race. In our example, the threads are in a race to execute $x += \text{my val}$.

61) Define Block.

A block of code that can only be executed by one thread at a time is called a **critical section**, and it's usually our job as programmers to insure **mutually exclusive** access to the critical section. In other words, we need to insure that if one thread is executing the code in the critical section, then the other threads are excluded.

62) How does mutex work?

The most commonly used mechanism for insuring mutual exclusion is a **mutual exclusion lock** or **mutex** or **lock**. A mutex is a special type of object that has support in the underlying hardware. The basic idea is that each critical section is *protected* by a lock. Before a thread can execute the code in the

critical section, it must “obtain” the mutex by calling a mutex function, and, when it’s done executing the code in the critical section, it should “relinquish” the mutex by calling an unlock function.

63) What are points to be noted for serialization?

Mutex enforces **serialization** of the critical section. Since only one thread at a time can execute the code in the critical section, this code is effectively serial. Thus, we want our code to have as few critical sections as possible, and we want our critical sections to be as short as possible.

64) Define Semaphores.

Semaphores are similar to mutexes, although the details of their behavior are slightly different, and there are some types of thread synchronization that are easier to implement with semaphores than mutexes.

65) Define Monitor.

A **monitor** provides mutual exclusion at a somewhat higher-level: it is an object whose methods can only be executed by one thread at a time.

66) What is a transaction?

In database management systems, a **transaction** is an access to a database that the system treats as a single unit. For example, transferring \$1000 from your savings account to your checking account should be treated by your bank’s software as a transaction, so that the software can’t debit your savings account without also crediting your checking account. If the software was able to debit your savings account, but was then unable to credit your checking account, it would *rollback* the transaction. In other words, the transaction would either be fully completed or any partial changes would be erased.

67) What is the basic idea of transactional memory?

The basic idea behind transactional memory is that critical sections in shared-memory programs should be treated as transactions. Either a thread successfully completes the critical section or any partial results are rolled back and the critical section is repeated.

68) What are the features of distributed memory?

- Message passing
- One sided communication
- Partitioned global address space languages PGAS
- Programming hybrid systems

69) Define speed-up linear speed-up.

If we define the **speedup** of a parallel program to be $S = T_{\text{serial}} / T_{\text{parallel}}$, then linear speedup has $S = p$, which is unusual. Furthermore, as p increases, we expect S to become a smaller and smaller fraction of the ideal, linear speedup p . Another way of saying this is that $S=p$ will probably get smaller and smaller as p .

increases. If we call the serial run-time T_{serial} and our parallel run-time T_{parallel} , then the best we can hope for is $T_{\text{parallel}} \leq T_{\text{serial}}/p$. When this happens, we say that our parallel program has **linear speedup**.

70) Define Efficiency.

This value, $E = \frac{S}{p}$, is sometimes called the **efficiency** of the parallel program. If we substitute the formula for S , we see that the efficiency is

$$E = \frac{S}{p} = \frac{\left(\frac{T_{\text{serial}}}{T_{\text{parallel}}}\right)}{p} = \frac{T_{\text{serial}}}{p \cdot T_{\text{parallel}}}$$

71) State Amdahl's law.

In computer architecture, **Amdahl's law** (or **Amdahl's argument**) gives the theoretical speedup in latency of the execution of a task *at fixed workload* that can be expected of a system whose resources are improved. It is named after computer scientist Gene Amdahl, and was presented at the AFIPS Spring Joint Computer Conference in 1967.

Amdahl's law can be formulated the following way:

$$S_{\text{latency}}(s) = \frac{1}{(1-p) + \frac{p}{s}}$$

where

- S_{latency} is the theoretical speedup in latency of the execution of the whole task;
- s is the speedup in latency of the execution of the part of the task that benefits from the improvement of the resources of the system;
- p is the percentage of the execution time of the whole task concerning the part that benefits from the improvement of the resources of the system *before the improvement*.

72) Define scalability.

If we can find a corresponding rate of increase in the problem size so that the program always has efficiency E , then the program is **scalable**. As an example, suppose that $T_{\text{serial}} \propto n$, where the units of T_{serial} are in microseconds, and n is also the problem size. Also suppose that $T_{\text{parallel}} \propto n/p + C$. Then

$$E = \frac{n}{p(n/p + 1)} = \frac{n}{n + p}$$

73) List the outline of steps in building Parallel programs.

- Partitioning
- Communication
- Agglomeration or Aggregation
- Mapping

CS6801 – MULTICORE ARCHITECTURES AND PROGRAMMING

2 MARKS Q & A

UNIT II – PARALLEL PROGRAM CHALLENGES

1) What is a data race?

Data races are the most common programming error found in parallel code. A data race occurs when multiple threads use the same data item and one or more of those threads are updating it. It is best illustrated by an example.

Table 4.1 Two Threads Updating the Same Variable

Value of variable a = 10	
Thread 1	Thread 2
ld [%0], %01 // Load %01 = 10	ld [%0], %01 // Load %01 = 10
add %01, 4, %01 // Add %01 = 14	add %01, 4, %01 // Add %01 = 14
st %01, [%0] // Store %01	st %01, [%0] // Store %01
Value of variable a = 14	

Eg : Updating the Value
at an Address
void update(int * a)
{ *a = *a + 4; }

In the example, each thread adds 4 to the variable, but because they do it at exactly the same time, the value 14 ends up being stored into the variable. If the two threads had executed the code at different times, then the variable would have ended up with the value of 18. This is the situation where both threads are running simultaneously. This illustrates a common kind of data race and possibly the easiest one to visualize.

2) What are the tools used for detecting data races?

- Helgrind
- Sun Studio Thread Analyzer

3) How to avoid data races?

It can be hard to identify data races, avoiding them can be very simple: Make sure that only one thread can update the variable at a time. The easiest way to do this is to place a *synchronization lock* around all accesses to that variable and ensure that before referencing the variable, the thread must acquire the lock.

4) What are synchronization primitives?

- Mutexes
- Locks
- Semaphores
- Barriers.

5) Define Mutex

The simplest form of synchronization is a mutually exclusive (*mutex*) lock. Only one thread at a time can acquire a mutex lock, so they can be placed around a data structure to ensure that the data structure is modified by only one thread at a time.

Eg : Placing mutex locks around accesses to variables

```
int counter;
mutex_lock mutex;
```

```
void Increment()
{
    acquire( &mutex );
    counter++;
    release( &mutex );
}
```

```
void Decrement()
{
    acquire( &mutex );
    counter--;
    release( &mutex );
}
```

6) Define Contended Mutex.

If multiple threads are attempting to acquire the same mutex at the same time, then only one thread will succeed, and the other threads will have to wait. This situation is known as a *contended* mutex.

7) Define Critical Region

The region of code between the acquisition and release of a mutex lock is called a *critical section*, or *critical region*. Code in this region will be executed by only one thread at a time.

8) Compare Spin locks and Mutex Locks.

Spin Locks	Mutex Locks
1. <i>Spin locks</i> are essentially mutex locks.	1. Simplest form of synchronization is mutex lock
2. A thread waiting to acquire a spin lock will keep trying to acquire the lock without sleeping	2. Acquire the lock after sleeping.
3. Adv : spin locks is that they will acquire the lock as soon as it is released	3. Dsdv : a mutex lock will need to be woken by the operating system before it can get the lock.
4. Dsdv: A spin lock will spin on a virtual CPU monopolizing that resource	4. Adv : mutex lock will sleep and free the virtual CPU for another thread to use.
5. The thread attempting to acquire the mutex spins for a short while before blocking.	5. Mutex locks are implemented to be a hybrid of spin locks
6. Held for a long period of time and continuing to spin for a long period of time consumes hardware resources that could be better used in allowing other software threads to run	6. Most mutex locks are held for only a short period of time, it is quite likely that the lock will quickly become free for the waiting thread to acquire.

9) Define Semaphores.

Semaphores are counters that can be either incremented or decremented. They can be used in situations where there is a finite limit to a resource and a mechanism is needed to impose that limit. An example might be a buffer that has a fixed size. Every time an element is added to a buffer, the number of available positions is decreased. Every time an element is removed, the number available is increased.

Depending on the implementation, the method that acquires a semaphore might be called *wait*, *down*, or *acquire*, and the method to release a semaphore might be called *post*, *up*, *signal*, or *release*.

10) What is reader- writer lock?

However, sometimes data that is typically read-only needs to be updated. A *readers writer lock* (or *multiple-reader lock*) allows many threads to read the shared data but can then lock the readers threads out to allow one thread to acquire a writer lock to modify the data.

A writer cannot acquire the write lock until all the readers have released their reader locks. For this reason, the locks tend to be biased toward writers; as soon as one is queued, the lock stops allowing further readers to enter. This action causes the number of readers holding the lock to

Listing 4.9 Using a Readers-Writer Lock

```
int readData( int cell1, int cell2 )
{
    acquireReaderLock( &lock );
    int result = data[cell1] + data[cell2];
    releaseReaderLock( &lock );
    return result;
}

void writeData( int cell1, int cell2, int value )
{
    acquireWriterLock( &lock );
    data[cell1] += value;
    data[cell2] -= value;
    releaseWriterLock( &lock );
}
```


diminish and will eventually allow the writer to get exclusive access to the lock.

11) Define Barriers.

There are situations where a number of threads have to all complete their work before any of the threads can start on the next task. In these situations, it is useful to have a barrier where the threads will wait until all are present. One common example of using a barrier arises when there is dependence between different sections of code.

Eg : Using a Barrier to Order Computation

```
Compute_values_held_in_matrix();
```

```
Barrier();
```

```
total = Calculate_value_from_matrix();
```

12) Define Atomic Operation.

An *atomic operation* is one that will either successfully complete or fail; it is not possible for the operation to either result in a “bad” value or allow other threads on the system to observe a transient value. An example of this would be an atomic increment, which would mean that the calling thread would replace a variable that currently holds the value N with the value N+1.

13) What is the advantage of Atomic Operation?

Using synchronization primitives can add a high overhead cost. This is particularly true if they are implemented as calls into the operating system rather than calls into a supporting library. These overheads lower the performance of the parallel application and can limit scalability. In some cases, either *atomic operations* or *lock-free code* can produce functionally equivalent code without introducing the same amount of overhead.

14) Define Deadlock.

First is the *deadlock*, where two or more threads cannot make progress because the resources that they need are held by the other threads.

15) Give an example for deadlock.

Suppose two threads need to acquire mutex locks A and B to complete some task. If thread 1 has already acquired lock A and thread 2 has already acquired lock B, then A cannot make forward progress because it is waiting for lock B, and thread 2 cannot make progress because it is waiting for lock A. The two threads are *deadlocked*.

Listing 4.13 Two Threads in a Deadlock

Thread 1	Thread 2
<pre>void update1() { acquire(A); acquire(B); <<< Thread 1 waits here variable1++; release(B); release(A); }</pre>	<pre>void update2() { acquire(B); acquire(A); <<< Thread 2 waits here variable1++; release(B); release(A); }</pre>

16) Define Livelock

A *livelock* traps threads in an unending loop releasing and acquiring locks. Livelocks can be caused by code to back out of deadlocks. In Listing below, the programmer has tried to implement a mechanism that avoids deadlocks. If the thread cannot obtain the second lock it requires, it releases the lock that it already holds.

17) List the mechanisms for communication.

- Memory –Direct Communication
- Shared Memory communication
- Memory Mapped Communication
- Communication using condition variables
- Communication by signals
- Event Communication
- Message Queues
- Named Pipes
- Communication through Network stacks

Definitions for all the mechanisms of communication.

CS6801 – MULTICORE ARCHITECTURES AND PROGRAMMING

2 MARKS Q & A

UNIT III – SHARED MEMORY PROGRAMMING WITH OPENMP

1) How to Compile and run OpenMP programs?

To compile this with gcc we need to include the -fopenmp option:

```
$ gcc -g -Wall -fopenmp -o omp hello omp hello.c
```

To run the program, we specify the number of threads on the command line.

For example, we might run the program with four threads and type \$./omp hello 4

2) Define structured block of code.

A structured block is a C statement or a compound C statement with one point of entry and one point of exit, although calls to the function exit are allowed. This definition simply prohibits code that branches into or out of the middle of the structured block.

3) Define thread.

Recollect that **thread** is short for *thread of execution*. The name is meant to suggest a sequence of statements executed by a program.

4) How a thread is started?

Threads are typically started or **forked** by a process, and they share most of the resources of the process that starts them—for example, access to stdin and stdout—but each thread has its own stack and program counter. When a thread completes execution it **joins** the process that started it. This terminology comes from diagrams that show threads as directed lines.

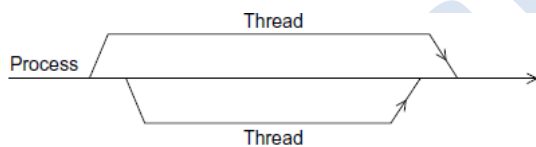


FIGURE 5.2

A process forking and joining two threads

5) What actually happens when the program gets to the parallel directive?

Prior to the parallel directive, the program is using a single thread, the process started when the program started execution. When the program reaches the parallel directive, the original thread continues executing and threads count - 1 additional thread is started.

6) Define Team, Master, slave.

In OpenMP parlance, the collection of threads executing the parallel block—the original thread and the new threads—is called a **team**, the original thread is called the **master**, and the additional threads are called **slaves**.

7) What is race condition?

Multiple threads are attempting to access a shared resource, at least one of the accesses is an update, and the accesses can result in an error. Also recall that the code that causes the race condition.

8) What is trap function in parallel directive?

The parallel directive specifies that the Trap function should be executed by thread count threads. After returning from the call to Trap, any new threads that were started by the parallel directive are terminated, and the program resumes execution with only one thread. The one thread prints the result and terminates. In the Trap function, each thread gets its rank and the total number of threads in the team started by the parallel directive.

9) Define Scope of a variable?

In serial programming, the *scope* of a variable consists of those parts of a program in which the variable can be used. For example, a variable declared at the beginning of a C function has “function-wide” scope, that is, it can only be accessed in the body of the function. On the other hand, a variable declared at the beginning of a .c file but outside any function has “file-wide” scope. The *default* scope of a variable can change with other directives, and that OpenMP provides clauses to modify the default scope.

10) List the Parallel directives.

- Trap function
- Scope of variable
- Reduction clause
- For Directive
- Caveats
- Data dependences
- Loop carried dependences
- Estimating

11) List the types of sorting in loops

- Bubble sort
- Transposition sort
- Odd-Even sort

12) What are the different types of schedules in schedule clause?

In general, the schedule clause has the form `schedule(<type> [, <chunksize>])`

- Static
- Dynamic or guided
- Auto
- Runtime

13) The chunksize is a positive integer. In OpenMP parlance, a **chunk** of iterations is a block of iterations that would be executed consecutively in the serial loop. The number of iterations in the block is the chunksize. Only static, dynamic, and guided schedules can have a chunksize.

14) How do we decide which type of schedule we should use and what the chunksize should be?

As you may have guessed, there *is* some overhead associated with the use of a schedule clause. Furthermore, the overhead is greater for dynamic schedules than static schedules, and the overhead associated with guided schedules is the greatest of the three. Thus, if we're getting satisfactory performance without a schedule clause, we should go no further. However, if we suspect that the performance of the default schedule can be substantially improved, we should probably experiment with some different schedules.

15) List the different producers and consumers.

- Queues
- Message Passing
- Sending Messages
- Receiving Messages
- Termination Detection
- StartUp

16) List the type of directives.

- Parallel Directive
- Atomic Directive
- Critical Directive

17) How the critical directive behaves?

The OpenMP specification allows the atomic directive to enforce mutual exclusion across *all* atomic directives in the program—this is the way the unnamed critical directive behaves. If this might be a problem—for example, you have multiple different critical sections protected by atomic directives—you should use named critical directives or locks.

18) Define Cache Memory.

If a processor must read data from main memory for each operation, it will spend most of its time simply waiting for the data from memory to arrive. Also recall that in order to address this problem, chip designers have added blocks of relatively fast memory to processors. This faster memory is called **cache memory**.

19) What is the principle of temporal and spatial locality?

The design of cache memory takes into consideration the principles of *temporal and spatial locality*: if a processor accesses main memory location x at time t , then it is likely that at times close to t , it will access main memory locations close to x .

20) Define Cache Block.

If a processor needs to access main memory location x , rather than transferring only the contents of x to/from main memory, a block of memory containing x is transferred from/to the processor's cache. Such a block of memory is called a **cache line** or **cache block**.

21) Give the formula for efficiency.

If T_{serial} is the run-time of the serial program and T_{parallel} is the run-time of the parallel program, recall that the *efficiency* E of the parallel program is the speedup S divided by the number of threads, t : Since $S \geq 1$, $E \geq 1$.

$$E = \frac{S}{t} = \frac{\left(\frac{T_{\text{serial}}}{T_{\text{parallel}}}\right)}{t} = \frac{T_{\text{serial}}}{t \times T_{\text{parallel}}}.$$

22) How frequently does the CPU need to access the page table in main memory?

Though, are the differences in efficiency as the number of threads is increased. The two-thread efficiency of the program with the 8_8,000,000 input is more than 20% less than the efficiency of the program with the

8,000,000_8 and the 8000_8000 inputs. The four-thread efficiency of the program with the 8_8,000,000 input is more than 50% less than the program's efficiency with the 8,000,000_8 and the 8000_8000 inputs.

23) Define Thread safety.

A block of code is **thread-safe** if it can be simultaneously executed by multiple threads without causing problems.

24) When a loop carried dependency does occurs?

A loop-carried dependence occurs when a memory location is read or written in one iteration and written in iteration. OpenMP won't detect loop-carried dependences; it's up to us, the programmers, to detect them and eliminate them. It may, however, be impossible to eliminate them, in which case, the loop isn't a candidate for parallelization.

25) Define Block Partitioning.

Most systems use a **block partitioning** of the iterations in a parallelized **for** loop. If there are n iterations, this means that roughly the first $n/\text{thread count}$ are assigned to thread 0, the next $n/\text{thread count}$ are assigned to thread 1, and so on.

CS6801 – MULTICORE ARCHITECTURES AND PROGRAMMING

2 MARKS Q & A

UNIT IV – DISTRIBUTED MEMORY PROGRAMMING WITH MPI

1) What does the distributed memory consist of in MIMD?

In the world of parallel multiple instruction, multiple data, or MIMD, computers is, for the most part, divided into **distributed-memory** and **shared-memory** systems. From a programmer's point of view, a distributed-memory system consists of a collection of core-memory pairs connected by a network, and the memory associated with a core is directly accessible only to that core.

2) What does the shared memory consist of in MIMD?

In the world of parallel multiple instruction, multiple data, or MIMD, computers is, for the most part, divided into **distributed-memory** and **shared-memory** systems. From a programmer's point of view, a shared-memory system consists of a collection of cores connected to a globally accessible memory, in which each core can have access to any memory location.

3) Draw the architecture of distributed memory system?

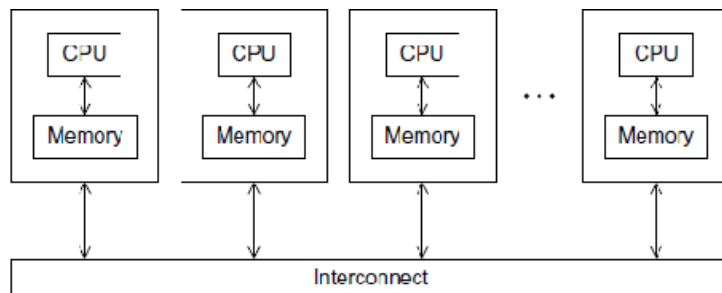


FIGURE 3.1

A distributed-memory system

4) Draw the architecture of shared memory system?

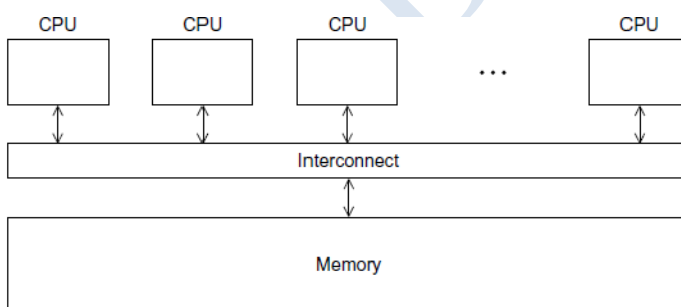


FIGURE 3.2

A shared-memory system

5) What is MPI? Write a brief note on it.

In message-passing programs, a program running on one core-memory pair is usually called a **process**, and two processes can communicate by calling functions: one process calls a *send* function and the other calls a *receive* function. The implementation of message-passing that we'll be using is called **MPI**, which is an abbreviation of **Message-Passing Interface**. MPI is not a new programming language. It defines a *library* of functions that can be called from C, C++, and Fortran programs.

6) What are Collective Communication functions?

Some “global” communication functions that can involve more than two processes. These functions are called **collective** communications.

7) write a program “hello, world” that makes some use of MPI.

```
#include <stdio.h>
int main(void) {
    printf("hello, world\n");
    return 0; }
```

8) How to compile and execute MPI programs?

The details of compiling and running the program depend on your system, so you may need to check with a local expert. We are using a text editor to write the program source and command line to compile and run.

Many systems use a command called mpicc for compilation:

```
$ mpicc -g -Wall -o mpi_hello mpi_hello.c
```

Many systems also support program startup with mpiexec:

```
$ mpiexec -n <number of processes> ./mpi_hello
```

So to run the program with one process, we’d type

```
$ mpiexec -n 1 ./mpi_hello
```

9) What is wrapper?

Typically, mpicc is a script that’s a **wrapper** for the C compiler. A **wrapper script** is a script whose main purpose is to run some program. In this case, the program is the C compiler. However, the wrapper simplifies the running of the compiler by telling it where to find the necessary header files and which libraries to link with the object file.

10) What are the two things to be observed in MPI Programs?

The first thing to observe is that this *is* a C program. For example, it includes the standard C header files `stdio.h` and `string.h`. It also has a main function just like any other C program. However, there are many parts of the program which are new. Line 3 includes the `mpi.h` header file. This contains prototypes of MPI functions, macro definitions, type definitions, and so on...

The second thing to observe is that all of the identifiers defined by MPI start with the string `MPI`. The first letter following the underscore is capitalized for function names and MPI-defined types. All of the letters in MPI-defined macros and constants are capitalized.

11) What is MPI_Init?

The call to MPI Init tells the MPI system to do all of the necessary setup. For example, it might allocate storage for message buffers, and it might decide which process gets which rank. As a rule of thumb, no other MPI functions should be called before the program calls MPI Init. Its syntax is

```
int MPI_Init(
    int* argc p /* in/out*_/,
    char*** argv p /* in/out */);
```

12) Define MPI_Finalize.

The call to MPI Finalize tells the MPI system that we're done using MPI, and that any resources allocated for MPI can be freed. The syntax is quite simple:

```
int MPI_Finalize(void);
```

In general, no MPI functions should be called after the call to MPI Finalize.

13) Give the List of MPI Functions

- MPI_Init
- MPI_Finalize
- MPI_Comm_Size
- MPI_Comm_Rank
- MPI_Send
- MPI_Recv
- MPI_Datatype
- MPI_Status
- MPI_Source
- MPI_Tag
- MPI_Error
- MPI_Get_Count

14) Write About MPI Communicator.

In MPI a **communicator** is a collection of processes that can send messages to each other. One of the purposes of MPI Init is to define a communicator that consists of all of the processes started by the user when she started the program. This communicator is called MPI COMM WORLD. The function calls in Lines are getting information about MPI COMM WORLD. Their syntax is

```
int MPI_Comm_size(  
    MPI_Comm comm      /* in */,  
    int* comm_sz_p     /* out */);  
  
int MPI_Comm_rank(  
    MPI_Comm comm      /* in */,  
    int* my_rank_p     /* out */);
```

15) Define SPMD.

In fact, *most* MPI programs are written in this way. That is, a single program is written so that different processes carry out different actions, and this is achieved by simply having the processes branch on the basis of their process rank. Recall that this approach to parallel programming is called **single program, multiple data**, or **SPMD**.

16) Write the Syntax for MPI_Send.

The first three arguments, msg buf p, msg size, and msg type, determine the contents of the message. The remaining arguments, dest, tag, and communicator, determine the destination of the message.

The first argument, msg buf p, is a pointer to the block of memory containing the contents of the message.

The second and third arguments, msg size and msg type, determine the amount of data to be sent. In our program, the msg size argument is the number of characters in the message plus one character for the 'n0' character that terminates C strings. The msg type argument is MPI CHAR. These two arguments together tell the system that the message contains strlen(greeting)+1 **chars**.


```
int MPI_Send(
    void*      msg_buf_p    /* in */,
    int        msg_size     /* in */,
    MPI_Datatype msg_type    /* in */,
    int        dest         /* in */,
    int        tag          /* in */,
    MPI_Comm   communicator /* in */);
```

The fourth argument, dest, specifies the rank of the process that should receive the message. **The fifth argument**, tag, is a nonnegative **int**. It can be used to distinguish messages that are otherwise identical. The **final argument** to MPI Send is a communicator. All MPI functions that involve communication have a communicator argument. One of the most important purposes of communicators is to specify communication universes

17) Write some of the pre-defined Datatypes of MPI.

Table 3.1 Some Predefined MPI Datatypes

MPI datatype	C datatype
MPI_CHAR	signed char
MPI_SHORT	signed short int
MPI_INT	signed int
MPI_LONG	signed long int
MPI_LONG_LONG	signed long long int
MPI_UNSIGNED_CHAR	unsigned char
MPI_UNSIGNED_SHORT	unsigned short int
MPI_UNSIGNED	unsigned int
MPI_UNSIGNED_LONG	unsigned long int
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_LONG_DOUBLE	long double
MPI_BYTE	
MPI_PACKED	

18) Give the Syntax for MPI_Recv.

```
int MPI_Recv(
    void*      msg_buf_p    /* out */,
    int        buf_size     /* in */,
    MPI_Datatype buf_type    /* in */,
    int        source       /* in */,
    int        tag          /* in */,
    MPI_Comm   communicator /* in */,
    MPI_Status* status_p    /* out */);
```

19) How the messages can be matched?

If recv type = send type and recv buf sz _ send buf sz, then the message sent by *q* can be successfully received by *r*. Of course, it can happen that one process is receiving messages from multiple processes, *and* the receiving process doesn't know the order in which the other processes will send the messages.

20) What are the couple of points to be considered in connection with these wildcard arguments?

A couple of points should be stressed in connection with these "wildcard" arguments:

1. Only a receiver can use a wildcard argument. Senders must specify a process rank and a nonnegative tag. Thus, MPI uses a "push" communication mechanism rather than a "pull" mechanism.

2. There is no wildcard for communicator arguments; both senders and receivers must always specify communicators.

21) Write in brief the status_p argument.

Notice that a receiver can receive a message without knowing

1. the amount of data in the message, 2. the sender of the message, or 3. the tag of the message.

So how can the receiver find out these values? Recall that the last argument to MPI Recv has type MPI Status_. The MPI type MPI Status is a struct with at least the three members MPI SOURCE, MPI TAG, and MPI ERROR. Suppose our program contains the definition **MPI Status status**

Then, after a call to MPI Recv in which &status is passed as the last argument, we can determine the sender and tag by examining the two members

status.MPI_SOURCE

status.MPI_TAG

22) Give the syntax for MPI_Get_Count.

```
MPI_Get_count(&status, recv_type, &count)
```

```
int MPI_Get_count(
    MPI_Status* status_p /* in */,
    MPI_Datatype type /* in */,
    int* count_p /* out */);
```

In our call to MPI Recv, the type of the receive buffer is recv type and, once again, we passed in &status. Then the call will return the number of elements received in the count argument.

23) Does MPI require messages to be non overtaking?

MPI requires that messages be **nonovertaking**. This means that if process q sends two messages to process r , then the first message sent by q must be available to r before the second message. However, there is no restriction on the arrival of messages sent from different processes. That is, if q and t both send messages to r , then even if q sends its message before t sends its message, there is no requirement that q 's message become available to r before t 's message. This is essentially because MPI can't impose performance on a network.

24) Draw the architecture for tree structured Communication.

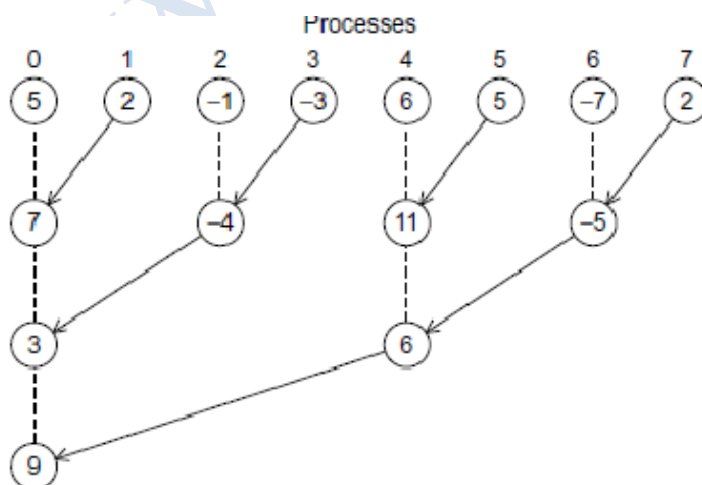


FIGURE 3.6

A tree-structured global sum

25) Define Collective communication and Point-to-Point Communication.

A “global-sum function” will obviously require communication. However, unlike the MPI Send-MPI Recv pair, the global-sum function may involve more than two processes. In fact, in our trapezoidal rule program it will involve all the processes in MPI COMM WORLD. In MPI parlance, communication functions that involve all the processes in a communicator are called **collective communications**. To distinguish between collective communications and functions such as MPI Send and MPI Recv, MPI Send and MPI Recv are often called **point-to-point** communications. In fact, global sum is just a special case of an entire class of collective communications.

26) Write the syntax for MPI_Reduce.

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

27) What are the pre-defined reduction operators in MPI?

Table 3.2 Predefined Reduction Operators in MPI

Operation Value	Meaning
MPI_MAX	Maximum
MPI_MIN	Minimum
MPI_SUM	Sum
MPI_PROD	Product
MPI_LAND	Logical and
MPI_BAND	Bitwise and
MPI_LOR	Logical or
MPI_BOR	Bitwise or
MPI_LXOR	Logical exclusive or
MPI_BXOR	Bitwise exclusive or
MPI_MAXLOC	Maximum and location of maximum
MPI_MINLOC	Minimum and location of minimum

28) Differentiate Collective and Point-to-Point Communication

S.No	Collective Communication	Point-to-Point Communication
1	All the processes in the communicator must call the same collective function.	All the processes call the different function.
2	The arguments passed by each process to an MPI collective communication must be “compatible.”	The arguments passed by each process to an MPI collective communication need not be “compatible.”
3	The output data p argument is only used on dest process. However, all of the processes still need to pass in an actual argument corresponding to output data p, even if it’s just NULL.	The output data p argument is only used on dest process. However, all of the processes still need to pass in an dummy argument corresponding to output data p
4	Collective communications don’t use tags, so they’re matched solely on the basis of the communicator and the <i>order</i> in which they’re called.	Point-to-point communications are matched on the basis of tags and communicators.

29) Define Butterfly.

In this situation, we encounter some of the same problems we encountered with our original global sum. For example, if we use a tree to compute a global sum, we might “reverse” the branches to distribute the global sum. Alternatively, we might have the processes *exchange* partial results instead of using one-way communications. Such a communication pattern is sometimes called a **butterfly**.

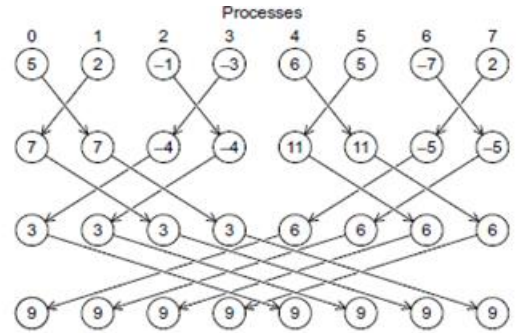


FIGURE 3.9

A butterfly-structured global sum

30) Give the syntax for MPI_Allreduce.

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

31) Define Broadcast in MPI.

Collective communication in which data belonging to a single process is sent to all of the processes in the communicator is called a **broadcast**.

a broadcast function:

```
int MPI_Bcast(
    void*      data_p        /* in/out */,
    int        count        /* in */,
    MPI_Datatype datatype    /* in */,
    int        source_proc   /* in */,
    MPI_Comm    comm        /* in */);
```

32) Give the syntax for MPI_Scatter.

```
int MPI_Scatter(
    void*      send_buf_p /* in */,
    int        send_count /* in */,
    MPI_Datatype send_type /* in */,
    void*      recv_buf_p /* out */,
    int        recv_count /* in */,
    MPI_Datatype recv_type /* in */,
    int        src_proc   /* in */,
    MPI_Comm    comm      /* in */);
```

33) Give the Syntax for MPI_Gather.

```
int MPI_Gather(
    void*      send_buf_p /* in */,
    int        send_count /* in */,
    MPI_Datatype send_type /* in */,
    void*      recv_buf_p /* out */,
    int        recv_count /* in */,
    MPI_Datatype recv_type /* in */,
    int        dest_proc  /* in */,
    MPI_Comm    comm      /* in */);
```

34) What are the different partitions available in MPI?

- (1) Block partitioning (2) Cyclic Partitioning (3) Block-Cyclic Partitioning

35) What are the 3 approaches to consolidating data?

MPI provides three basic approaches to consolidating data that might otherwise require multiple messages: the count argument to the various communication functions, derived datatypes, and MPI Pack/Unpack. We've already seen the count argument—it can be used to group contiguous array elements into a single message. discuss one method for building derived datatypes MPI Pack/Unpack.

36) Define MPI Derived Data Types.

In MPI, a **derived data type** can be used to represent any collection of data items in memory by storing both the types of the items and their relative locations in memory. The idea here is that if a function that sends data knows the types and the relative locations in memory of a collection of data items, it can collect the items from memory before they are sent. Similarly, a function that receives data can distribute the items into their correct destinations in memory when they're received.

37) Define MPI_Wtime.

MPI provides a function, MPI Wtime, that returns the number of seconds that have elapsed since some time in the past: **double MPI Wtime(void);**

Thus, we can time a block of MPI code as follows:

double start, finish;

```
start = MPI_Wtime();
/* Code to be timed */
finish = MPI_Wtime();
printf("Proc %d > Elapsed time = %e seconds\n",
      my_rank, finish-start);
```

38) Define Speedup and Linear Speedup.

Recall that the most widely used measure of the relation between the serial and the parallel run-times is the **speedup**. It's just the ratio of the serial run-time to the parallel run-time:

$$S(n,p) = \frac{T_{\text{serial}}(n)}{T_{\text{parallel}}(n,p)}.$$

The ideal value for $S(n,p)$ is p . If $S(n,p) \geq p$, then our parallel program with comm sz $\geq p$ processes is running p times faster than the serial program. In practice, this speedup, sometimes called **linear speedup**, is rarely achieved.

39) Give the syntax for ODD EVEN transportation sort.

```
Sort local keys;
for (phase = 0; phase < comm_sz; phase++) {
    partner = Compute_partner(phase, my_rank);
    if (I'm not idle) {
        Send my keys to partner;
        Receive keys from partner;
        if (my_rank < partner)
            Keep smaller keys;
        else
            Keep larger keys;
    }
}
```

CS6801 – MULTICORE ARCHITECTURES AND PROGRAMMING

2 MARKS Q & A

UNIT V – PARALLEL PROGRAM DEVELOPMENT

1) What does the n -body problem do?

In an n -body problem, we need to find the positions and velocities of a collection of interacting particles over a period of time. For example, an astrophysicist might want to know the positions and velocities of a collection of stars, while a chemist might want to know the positions and velocities of a collection of molecules or atoms. An n -body solver is a program that finds the solution to an n -body problem by simulating the behavior of the particles.

2) Give the pseudocode for serial n -body solver.

```
1  Get input data;
2  for each timestep {
3      if (timestep output) Print positions and velocities of
        particles;
4      for each particle q
5          Compute total force on q;
6      for each particle q
7          Compute position and velocity of q;
8  }
9  Print positions and velocities of particles;
```

3) Write the pseudocode for computing n -body forces.

```
for each particle q
    forces[q] = 0;
for each particle q {
    for each particle k > q {
        x_diff = pos[q][X] - pos[k][X];
        y_diff = pos[q][Y] - pos[k][Y];
        dist = sqrt(x_diff*x_diff + y_diff*y_diff);
        dist_cubed = dist*dist*dist;
        force_qk[X] = G*masses[q]*masses[k]/dist_cubed * x_diff;
        force_qk[Y] = G*masses[q]*masses[k]/dist_cubed * y_diff;

        forces[q][X] += force_qk[X];
        forces[q][Y] += force_qk[Y];
        forces[k][X] -= force_qk[X];
        forces[k][Y] -= force_qk[Y];
    }
}
```

Program 6.1: A reduced algorithm for computing n -body forces

4) State the Euler's method.

There are *many* possible choices for numerical methods, but we'll use the simplest one: Euler's method, which is named after the famous Swiss mathematician Leonhard Euler (1707–1783). In Euler's method, we use the tangent line to approximate a function. The basic idea is that if we know the value of a function $g(t_0)$ at time t_0 and we also know its derivative $g'(t_0)$ at time t_0 , then we can approximate its value at time $t_0 + \Delta t$ by using the tangent line to the graph of $g(t_0)$. Now if we know a point $(t_0, g(t_0))$ on a line, and we know the slope of the line $g'(t_0)$, then an equation for the line is given by

$$y = g(t_0) + g'(t_0)(t - t_0).$$

5) Give the communication among the n -body solvers.

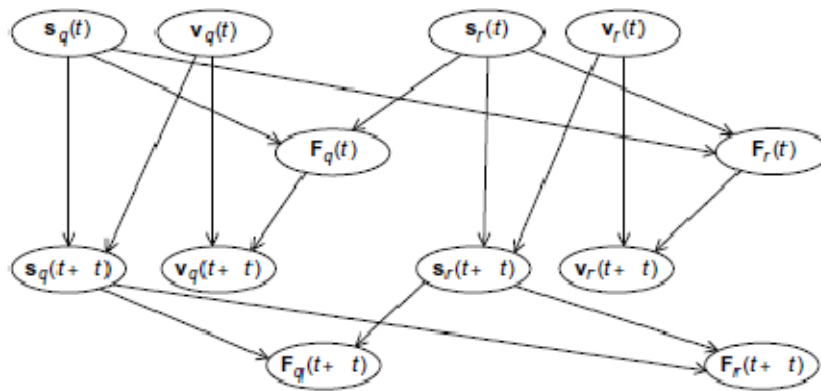


FIGURE 6.3

Communications among tasks in the basic n -body solver

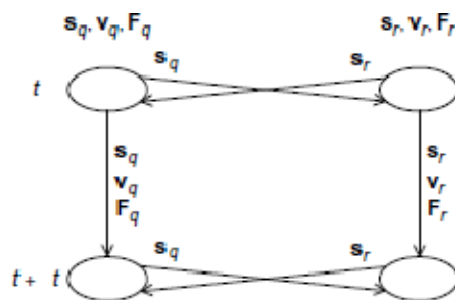


FIGURE 6.4

Communications among agglomerated tasks in the basic n -body solver

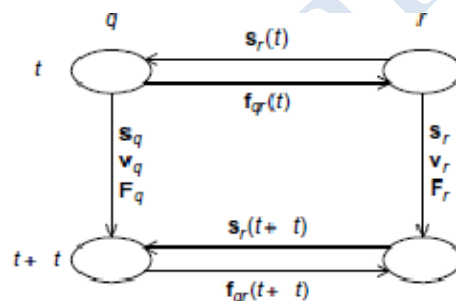


FIGURE 6.5

Communications among agglomerated tasks in the reduced n -body solver ($q < r$)

6) How will you parallelize the reduced solver using OpenMP?

```
# pragma omp parallel
for each timestep {
    if (timestep output) {
        # pragma omp single
        Print positions and velocities of particles;
    }
    # pragma omp for
    for each particle q
        forces[q] = 0.0;
    # pragma omp for
    for each particle q
        Compute total force on q;
    # pragma omp for
    for each particle q
        Compute position and velocity of q;
}
```

7) What are the principal differences between parallelizing the solvers using n-body solvers and pthreads.

- By default local variables in Pthreads are private, so all shared variables are global in the Pthreads version.
- The principal data structures in the Pthreads version are identical to those in the OpenMP version: vectors are two-dimensional arrays of **doubles**, and the mass, position, and velocity of a single particle are stored in a struct. The forces are stored in an array of vectors.
- Startup for Pthreads is basically the same as the startup for OpenMP: the main thread gets the command-line arguments, and allocates and initializes the principal data structures.
- The main difference between the Pthreads and the OpenMP implementations is in the details of parallelizing the inner loops. Since Pthreads has nothing analogous to a parallel **for** directive, we must explicitly determine which values of the loop variables correspond to each thread's calculations.

8) How to parallelize the n-body solver by MPI?

The only communication among the tasks occurs when we're computing the forces, and, in order to compute the forces, each task/particle needs the position and mass of every other particle. MPI Allgather is expressly designed for this situation, since it collects on each process the same information from every other process. We've already noted that a block distribution will probably have the best performance, so we should use a block mapping of the particles to the processes.

9) Give the Pseudocode for the MPI version of n-body solver.

```

1  Get input data;
2  for each timestep {
3      if (timestep output)
4          Print positions and velocities of particles;
5      for each local particle loc_q
6          Compute total force on loc_q;
7      for each local particle loc_q
8          Compute position and velocity of loc_q;
9      Allgather local positions into global pos array;
10 }
11 Print positions and velocities of particles;

```

Program 6.2: Pseudocode for the MPI version of the basic n -body solver

10) Discuss the performance of MPI Solvers

The run-times of the serial solvers differed from the single-process MPI solvers by less than 1%, so we haven't included them. Clearly, the performance of the reduced solver is much superior to the performance of the basic solver, although the basic solver achieves higher efficiencies.

Table 6.5 Performance of the MPI n -Body Solvers (times in seconds)

Processes	Basic	Reduced
1	17.30	8.68
2	8.65	4.45
4	4.35	2.30
8	2.20	1.26
16	1.13	0.78

Table 6.6 Run-Times for OpenMP and MPI n -Body Solvers (times in seconds)

Processes/ Threads	OpenMP		MPI	
	Basic	Reduced	Basic	Reduced
1	15.13	8.77	17.30	8.68
2	7.62	4.42	8.65	4.45
4	3.85	2.26	4.35	2.30

11) Give the pseudocode for recursive solution to TSP using depth first search.

```

1 void Depth_first_search(tour_t tour) {
2     city_t city;
3
4     if (City_count(tour) == n) {
5         if (Best_tour(tour))
6             Update_best_tour(tour);
7     } else {
8         for each neighboring city
9             if (Feasible(tour, city)) {
10                 Add_city(tour, city);
11                 Depth_first_search(tour);
12                 Remove_last_city(tour, city);
13             }
14     }
15 } /* Depth_first_search */

```

Program 6.4: Pseudocode for a recursive solution to TSP using depth-first search

12) Give the pseudocode for non recursive solution to TSP using depth first search.

```

1 for (city = n-1; city >= 1; city--)
2     Push(stack, city);
3 while (!Empty(stack)) {
4     city = Pop(stack);
5     if (city == NO_CITY) // End of child list, back up
6         Remove_last_city(curr_tour);
7     else {
8         Add_city(curr_tour, city);
9         if (City_count(curr_tour) == n) {
10             if (Best_tour(curr_tour))
11                 Update_best_tour(curr_tour);
12             Remove_last_city(curr_tour);
13         } else {
14             Push(stack, NO_CITY);
15             for (nbr = n-1; nbr >= 1; nbr--)
16                 if (Feasible(curr_tour, nbr))
17                     Push(stack, nbr);
18         }
19     } /* if Feasible */
20 } /* while !Empty */

```

Program 6.5: Pseudocode for an implementation of a depth-first solution to TSP that doesn't use recursion

13) Write the data structures for Serial implementations.

```

void Push(my_stack_t stack, int city) {
    int loc = stack->list_sz;
    stack->list[loc] = city;
    stack->list_sz++;
} /* Push */

void Push_copy(my_stack_t stack, tour_t tour) {
    int loc = stack->list_sz;

    tour_t tmp = Alloc_tour();
    Copy_tour(tour, tmp);
    stack->list[loc] = tmp;
    stack->list_sz++;
} /* Push */

```

14) How to parallelize tree search?

We need to keep in mind that the best tour data structure requires additional communication that is not explicit in the tree edges. Thus, it's convenient to add an additional task that corresponds to the best tour. It "sends" data to every tree node task, and receives data from some of the leaves. This latter view is convenient for shared-memory, but not so convenient for distributed-memory.

A natural way to agglomerate and map the tasks is to assign a subtree to each thread or process, and have each thread/process carry out all the tasks in its subtree. For example, if we have three threads or processes, as shown earlier and we the subtree rooted at 0->1 to thread/process 0, the subtree rooted at 0->2 to thread/process 1, and the subtree rooted at 0->3 to thread/process 2.

15) What is the problem with depth first search?

The problem with depth-first search is that we expect a subtree whose root is deeper in the tree to require less work than a subtree whose root is higher up in the tree, so we would probably get better load balance if we used something like **breadth-first search** to identify the subtrees.

16) Why dynamic mapping scheme is used?

A second issue we should consider is the problem of load imbalance. Although the use of breadth-first search ensures that all of our subtrees have approximately the same number of nodes, there is no guarantee that they all have the same amount of work. It's entirely possible that one process or thread will have a subtree consisting of very expensive tours, and, as a consequence, it won't need to search very deeply into its assigned subtree. However, with our current, *static* mapping of tasks to threads/processes, this one thread or process will simply have to wait until the other threads/processes are done. An alternative is to implement a **dynamic** mapping scheme.

17) Define Dynamic Mapping Scheme.

In a dynamic scheme, if one thread/process runs out of useful work, it can obtain additional work from another thread/process. In our final implementation of serial depth-first search, each stack record contains a partial tour. With this data structure a thread or process can give additional work to another thread/process by dividing the contents of its stack. This might at first seem to have the potential for causing problems with the program's correctness, since if we give part of one thread's or one process' stack to another, there's a good chance that the order in which the tree nodes will be visited will be changed.

18) Why is the stack splitted for checking the thread termination?

Since our goal is to balance the load among the threads, we would like to insure that the amount of work in the new stack is roughly the same as the amount remaining in the original stack. We have no way of knowing in advance of searching the subtree rooted at a partial tour how much work is actually associated with the partial tour, so we'll never be able to guarantee an equal division of work.

19) What are the details to be checked for thread termination?

There are several details that we should look at more closely. Notice that the code executed by a thread before it splits its stack is fairly complicated. In Lines 1–2 the thread

- checks that it has at least two tours in its stack,
- checks that there are threads waiting, and
- checks whether the new stack variable is NULL.

The reason for the check that the thread has enough work should be clear: if there are fewer than two records on the thread's stack, "splitting" the stack will either do nothing or result in the active thread's trading places with one of the waiting threads.

20) How will the implementation of tree search using MPI and static partitioning carried out?

The vast majority of the code used in the static parallelizations of tree search using Pthreads and OpenMP is taken straight from the second implementation of serial, iterative tree search. In fact, the only differences are in starting the threads, the initial partitioning of the tree, and the **Update best tour** function. We might therefore expect that an MPI implementation would also require relatively few changes to the serial code, and this is, in fact, the case.

21) How to solve the problem of distributing input data and collecting results?

There is the usual problem of distributing the input data and collecting the results. In order to construct a complete tour, a process will need to choose an edge into each vertex and out of each vertex. Thus, each tour will require an entry from each row and each column for each city that's added to the tour, so it would clearly be advantageous for each process to have access to the entire adjacency matrix. Note that the adjacency matrix is going to be relatively small.

22) How to partition the tree?

Using MPI ScatterV function.

```
int MPI_Scatterv(
    void*      sendbuf      /* in */,

    int*       sendcounts   /* in */,
    int*       displacements /* in */,
    MPI_Datatype sendtype    /* in */,
    void*      recvbbuf     /* out */,
    int        recvcoun     /* in */,
    MPI_Datatype recvttype  /* in */,
    int        root         /* in */,
    MPI_Comm   comm         /* in */);
```

23) Give the MPI code to check for new best tour costs.

```
MPI_Iprobe(MPI_ANY_SOURCE, NEW_COST_TAG, comm, &msg_avail,
            &status);
while (msg_avail) {
    MPI_Recv(&received_cost, 1, MPI_INT, status.MPI_SOURCE,
             NEW_COST_TAG, comm, MPI_STATUS_IGNORE);
    if (received_cost < best_tour_cost)
        best_tour_cost = received_cost;
    MPI_Iprobe(MPI_ANY_SOURCE, NEW_COST_TAG, comm, &msg_avail,
              &status);
} /* while */
```

Program 6.9: MPI code to check for new best tour costs

24) Give the code for buffer attach and buffer detach

The buffer that's used by MPI_Bsend must be turned over to the MPI implementation with a call to MPI_Buffer_attach:

```
int MPI_Buffer_attach(
    void* buffer      /* in */,
    int   buffer_size /* in */);
```

The buffer argument is a pointer to a block of memory allocated by the user program and buffer_size is its size in bytes. A previously "attached" buffer can be reclaimed by the program with a call to

```
int MPI_Buffer_detach(
    void* buf_p      /* out */,
    int*  buf_size_p /* out */);
```

25) Give the code for MPI_Pack and MPI_Unpack

```
int MPI_Pack(
    void*      data_to_be_packed /* in */,
    int        to_be_packed_count /* in */,
    MPI_Datatype datatype /* in */,
    void*      contig_buf /* out */,
    int        contig_buf_size /* in */,
    int*       position_p /* in/out */,
    MPI_Comm   comm /* in */);

int MPI_Unpack(
    void*      contig_buf /* in */,
    int        contig_buf_size /* in */,
    int*       position_p /* in/out */,
    void*      unpacked_data /* out */,
    int        unpack_count /* in */,
    MPI_Datatype datatype /* in */,
    MPI_Comm   comm /* in */);
```

26) How to detect Distributed Termination?

The functions **Out of work and No work left** (Lines 11 and 15) implement the termination detection algorithm. As we noted earlier, an algorithm that's modeled on the termination detection algorithm we used in the shared-memory programs will have problems. To see this, suppose each process stores a variable oow, which stores the number of processes that are out of work. The variable is set to 0 when the program starts. Each time a process runs out of work, it sends a message to all the other processes saying it's out of work so that all the processes will increment their copies of oow. Similarly, when a process receives work from another process, it sends a message to every process informing them of this, and each process will decrement its copy of oow.

27) How to send "request for work" message?

However, there are many possibilities for choosing a destination:

1. Loop through the processes in round-robin fashion. Start with (my rank + 1) % comm sz and increment this destination (modulo comm sz) each time a new request is made.
2. Keep a global destination for requests on process 0.
3. Each process uses a random number generator to generate destinations.

28) How can we decide which API, MPI, Pthreads, or OpenMP is best for our application?

As a first step, decide whether to use distributed-memory, or shared-memory. In order to do this, first consider the amount of memory the application will need. In general, distributed-memory systems can provide considerably more main memory than shared-memory systems, so if the memory requirements are very large, you may need to write the application using MPI.

If the problem will fit into the main memory of your shared-memory system, you may still want to consider using MPI. Since the total available cache on a distributed-memory system will probably be much greater than that available on a shared-memory system, it's conceivable that a problem that requires lots of main memory accesses on a shared-memory system will mostly access cache on a distributed-memory system, and, consequently, have much better overall performance.

However, even if you'll get a big performance improvement from the large aggregate cache on a distributed-memory system, if you already have a large and complex serial program, it often makes sense to write a shared-memory program. Another consideration is the communication requirements of the parallel algorithm. If the processes/threads do little communication, an MPI program should be fairly easy to develop, and very scalable.