FRANCIS XAVIER™
ENGINEERING COLLEGE
AN AUTONOMOUS INSTITUTION

Accredited BY NBA | AICTE Sponsored Margdarshan Mentor Institution
DST-FIST Supported Institution | ISO 9001:2015 Certified
Recognized under Section 2(f) & 12(B) of the UGC Act, 1950

/fxengg /fxengg /fx_ec /fxengg
/francisxavierengineeringcollege

Department of Computer Science and Engineering

Academic Year: 2022 – 2023 [EVEN SEMESTER]

SUBJECT CODE: 19CS8708

SUBJECT NAME: MULTI-CORE ARCHITECTURES AND PROGRAMMING

Regulation: 2019 **Year and Semester: IV & VIII**

Prepared by,

Dr. S. Balaji,

Professor,

Department of Computer Science and Engineering.

FRANCIS XAVIER ENGINEERING COLLEGE

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

SYLLABUS

REGULATION 2019

19CS8708 - MULTI-CORE ARCHITECTURES AND PROGRAMMING

UNIT I - MULTI-CORE PROCESSORS 9

Single core to Multi-core architectures – SIMD and MIMD systems – Interconnection networks - Symmetric and Distributed Shared Memory Architectures - Cache coherence - Performance Issues – Parallel program design.

UNIT II - PARALLEL PROGRAM CHALLENGES 9

Performance – Scalability – Synchronization and data sharing – Data races – Synchronization primitives (mutexes, locks, semaphores, barriers) - deadlocks and livelocks - communication between threads (condition variables, signals, message queues and pipes).

UNIT III - SHARED MEMORY PROGRAMMING WITH OpenMP 9

OpenMP Execution Model – Memory Model - OpenMP Directives – Work-sharing Constructs – Library functions – Handling Data and Functional Parallelism – Handling Loops – Performance Considerations.

UNIT IV - DISTRIBUTED MEMORY PROGRAMMING WITH MPI 9

MPI program execution – MPI constructs – libraries – MPI send and receive – Point-to-point and Collective communication – MPI derived datatypes – Performance evaluation

UNIT V - PARALLEL PROGRAM DEVELOPMENT 9

Case studies - n-Body solvers – Tree Search – OpenMP and MPI implementations and comparison.

TOTAL: 45 PERIODS

TEXT BOOKS:

1. Peter S. Pacheco, ―An Introduction to Parallel Programming‖, Morgan-Kauffman/Elsevier, 2011.
2. Darryl Gove, ―Multicore Application Programming for Windows, Linux, and Oracle Solaris‖, Pearson, 2011 (unit 2)

REFERENCES:

1. Michael J Quinn, ―Parallel programming in C with MPI and OpenMP‖, Tata McGraw Hill, 2003.
2. Shameem Akhter and Jason Roberts, ―Multi-core Programming‖, Intel Press, 2006.

TABLE OF CONTENTS

SL.NO	TOPIC	PAGE. NO
1	AIM & OBJECTIVE OF THE SUBJECT	1
2	DETAILED LESSON PLAN	2
UNIT-I : MULTI-CORE PROCESSORS		
3	PART-A	4
4	SIMD SYSTEMS	6
5	MIMD SYSTEMS	10
6	INTERCONNECTION NETWORKS	13
7	CACHE COHERENCE	17
8	PERFORMANCE ISSUES	19
UNIT – II : PARALLEL PROGRAM CHALLENGES		
9	PART-A	23
10	PERFORMANCE	24
11	DATA RACES	27
12	SYNCHRONIZATION PRIMITIVES	31
13	DEADLOCKS AND LIVELOCKS	36
14	COMMUNICATION BETWEEN THREADS	38
UNIT – III : SHARED MEMORY PROGRAMMING WITH OpenMP		
15	PART-A	43
16	OpenMP EXECUTION MODEL	45
17	OpenMP DIRECTIVES	49
18	WORK-SHARING CONSTRUCTS	52
19	LIBRARY FUNCTIONS	54
20	HANDLING LOOPS	57
UNIT-IV : DISTRIBUTED MEMORY PROGRAMMING WITH MPI		
21	PART-A	61
22	MPI PROGRAM EXECUTION	63
23	MPI SEND AND RECEIVE	66
24	COLLECTIVE COMMUNICATION	69
25	MPI DERIVED DATATYPES	73
UNIT-5 PARALLEL PROGRAM DEVELOPMENT		
26	PART-A	77
27	nBODY SOLVERS	79
28	TREE SEARCH	83
29	OpenMP AND MPI IMPLEMENTATIONS AND COMPARISONS	87

OBJECTIVES:

- To understand the need for multi-core processors, and their architecture.
- To understand the challenges in parallel and multi-threaded programming.
- To learn about the various parallel programming paradigms
- To learn the shared and distributed Programming
- To develop multicore programs and design parallel solutions.

COURSE OUTCOME(S):

CO808.1 Describe multicore architectures and identify their characteristics and challenges.
CO808.2 Identify the issues in Parallel programming Processors.
CO808.3 Write programs using OpenMP and MPI.
CO808.3 Design parallel programming solutions to common problems.
CO808.5 Compare and contrast programming for serial processors and programming for parallel processors

PO Vs CO Mapping

CO No	PO1	PO2	PO3	PO4	PO5	PO6	PO7	PO8	PO9	PO10	PO11	PO12
1	3	2										
2	3	2										
3	3	2	1									
4	3	2	1									
5	3	2	1	1								

1→Low 2→Medium 3→High

DETAILED LESSON PLAN

TEXT BOOKS:

1. Peter S. Pacheco, ―An Introduction to Parallel Programming‖, Morgan-Kauffman/Elsevier, 2011.
2. Darryl Gove, ―Multicore Application Programming for Windows, Linux, and Oracle Solaris‖, Pearson, 2011 (unit 2)

REFERENCES:

1. Michael J Quinn, ―Parallel programming in C with MPI and OpenMP‖, Tata McGraw Hill, 2003.
2. Shameem Akhter and Jason Roberts, ―Multi-core Programming‖, Intel Press, 2006.

Sl. No	Unit	Topic / Portions to be Covered	Hours Required / Planned	Cumulative Hrs	Books Referred
UNIT-I : MULTI-CORE PROCESSORS					
1	I	Single core to Multi-core architectures	1	1	TB1
2		SIMD Systems	1	2	TB1, RB2
3		MIMD systems	1	3	TB1, RB2
4		Interconnection networks	1	4	TB1, RB2
5		Symmetric and Distributed Shared Memory Architectures	2	6	TB1, RB2
6		Cache coherence	1	7	TB1, RB2
7		Performance Issues	1	8	TB1, RB2
8		Parallel program design	1	9	TB1, RB2
UNIT – II : PARALLEL PROGRAM CHALLENGES					
9	II	Performance	1	10	TB2
10		Scalability	1	11	TB2
11		Synchronization and data sharing	1	12	TB2
12		Data races	1	13	TB2
13		Synchronization primitives	2	15	TB2
14		Deadlocks and Livelocks	1	16	TB2
15		Communication between threads	2	18	TB2

Sl. No	Unit	Topic / Portions to be Covered	Hours Required / Planned	Cumulative Hrs	Books Referred
UNIT – III : SHARED MEMORY PROGRAMMING WITH OpenMP					
16	III	OpenMP Execution Model	1	19	TB1, RB1
17		Memory Model	1	20	TB1, RB1
18		OpenMP Directives	1	21	TB1, RB1
19		Work-sharing Constructs	1	22	TB1, RB1
20		Library functions	1	23	TB1, RB1
21		Handling Data and Functional Parallelism	2	25	TB1, RB1
22		Handling Loops	1	26	TB1, RB1
23		Performance Considerations	1	27	TB1, RB1
UNIT-IV : DISTRIBUTED MEMORY PROGRAMMING WITH MPI					
24	IV	MPI program execution	1	28	TB1, RB1
25		MPI constructs	1	29	TB1, RB1
26		MPI libraries	2	31	TB1, RB1
27		MPI send and receive	1	32	TB1, RB1
28		Point-to-point and Collective communication	2	34	TB1, RB1
29		MPI derived datatypes	1	35	TB1, RB1
30		Performance evaluation	1	36	TB1, RB1
UNIT-V : PARALLEL PROGRAM DEVELOPMENT					
31	V	n-Body solvers	3	39	TB1
32		Tree Search	3	42	TB1
33		OpenMP and MPI implementations and comparison	3	45	TB1

UNIT-1 MULTI-CORE PROCESSORS

PART-A

1. Define: Multi-core processor.

A multi-core processor is a single computing component with two or more independent actual processing units (called "cores"), which are units that read and execute program instructions. The instructions are ordinary CPU instructions (such as add, move data, and branch), but the multiple cores can run multiple instructions at the same time, increasing overall speed for programs amenable to parallel computing.[2]Manufacturers typically integrate the cores onto a single integrated circuit die (known as a chip multiprocessor or CMP), or onto multiple dies in a single chip package.

2. Define: Vector processor.

In computing, a vector processor or array processor is a central processing unit (CPU) that implements an instruction set containing instructions that operate on one-dimensional arrays of data called vectors, compared to scalar processors, whose instructions operate on single data items.

3. What is meant by NUMA?

Non-uniform memory access (NUMA) is a computer memory design used in multiprocessing, where the memory access time depends on the memory location relative to the processor. Under NUMA, a processor can access its own local memory faster than non-local memory (memory local to another processor or memory shared between processors). The benefits of NUMA are limited to particular workloads, notably on servers where the data are often associated strongly with certain tasks or users.

4. Define: SIMD Systems.

Single instruction, multiple data (SIMD), is a class of parallel computers in Flynn's taxonomy. It describes computers with multiple processing elements that perform the same operation on multiple data points simultaneously. Thus, such machines exploit data level parallelism, but not concurrency: there are simultaneous (parallel) computations, but only a single process (instruction) at a given moment. SIMD is particularly applicable to common tasks like adjusting the contrast in a digital image or adjusting the volume of digital audio. Most modern CPU designs include SIMD instructions in order to improve the performance of multimedia use.

5. Define: MIMD Systems.

MIMD (multiple instruction, multiple data) is a technique employed to achieve parallelism. Machines using MIMD have a number of processors that function asynchronously and independently. At any time, different processors may be executing different instructions on different pieces of data. MIMD architectures may be used in a number of application areas such as computer-aided design/computer-aided manufacturing, simulation, modeling, and as communication switches.

6. Define: False sharing.

False sharing is a performance-degrading usage pattern that can arise in systems with distributed, coherent caches at the size of the smallest resource block managed by the caching mechanism. When a system participant attempts to periodically access data that will never be altered by another party, but that data shares a cache block with data that *is* altered, the caching protocol may force the first participant to reload the whole unit despite a lack of logical necessity. The caching system is unaware of activity within this block and forces the first participant to bear the caching system overhead required by true shared access of a resource.

7. Define: Cache Coherence.

Cache coherence is the consistency of shared resource data that ends up stored in multiple local caches. When clients in a system maintain caches of a common memory resource, problems may arise with inconsistent data, which is particularly the case with CPUs in a multiprocessing system.

8. Write a mathematical formula for speedup and efficiency of parallel program.

Speedup:

$$\boxed{S = \frac{T_{\text{serial}}}{T_{\text{parallel}}},}$$

Efficiency:

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

T_{serial}	-	Serial run time of parallel program
T_{parallel}	-	Parallel run time of parallel program

S	-	Speedup of a parallel program
P	-	Number of cores
E	-	Efficiency

9. What are the issues available in handling the performance?

- Speedup and Efficiency
- Amdahl‘s Law
- Scalability
- Taking Timings

10. What is the need for snooping protocol?

Snooping protocol ensures memory cache coherency in symmetric multiprocessing (SMP) systems. Each processor cache on a bus monitors, or snoops, the bus to verify whether it has a copy of a requested data block. Before a processor writes data, other processor cache copies must be invalidated or updated. Snooping protocol is also known as bus-snooping protocol.

PART-B

SIMD SYSTEMS

1. Explain briefly about SIMD systems. (16)

In parallel computing, Flynn‘s taxonomy is frequently used to classify computer architectures. It classifies a system according to the number of instruction streams and the number of data streams it can simultaneously manage. A classical von Neumann system is therefore 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. As an example, suppose we want to carry out a ―vector addition.‖ That is, suppose we have two arrays x and y, each with n elements, and we want to add the elements of y to the elements of x:

```
for (i = 0; i < n; i++)
        x[i] += y[i];
```

Suppose further that our SIMD system has n ALUs. Then we could load x[i] and y[i] into the ith ALU, have the ith ALU add y[i] to x[i], and store the result in x[i]. If the system has m ALUs and m < n, we can simply execute the additions in blocks of m elements at a time. For example, if m D 4 and n D 15, we can first add elements 0 to 3, then elements 4 to 7, then elements 8 to 11, and finally elements 12 to 14. Note that in the last group of elements in our example—elements 12 to 14—we‘re only operating on three elements of x and y, so one of the four ALUs will be idle.

The requirement that all the ALUs execute the same instruction or are idle can seriously degrade the overall performance of a SIMD system. For example, suppose we only want to carry out the addition if y[i] is positive:

```
for (i = 0; i < n; i++)
        if (y[i] > 0.0) x[i] += y[i];
```

In this setting, we must load each element of y into an ALU and determine whether it‘s positive. If y[i] is positive, we can proceed to carry out the addition. Otherwise, the ALU storing y[i] will be idle while the other ALUs carry out the addition.

Note also that in a ―classical‖ SIMD system, the ALUs must operate synchronously, that is, each ALU must wait for the next instruction to be broadcast before proceeding. Further, the ALUs have no instruction storage, so an ALU can‘t delay execution of an instruction by storing it for later execution.

Finally, as our first example shows, SIMD systems are ideal for parallelizing simple loops that operate on large arrays of data. Parallelism that‘s obtained by dividing data among the processors and having the processors all apply (more or less) the same instructions to their subsets of the data is called data-parallelism. SIMD parallelism can be very efficient on large data parallel problems, but SIMD systems often don‘t do very well on other types of parallel problems.

SIMD systems have had a somewhat checkered history. In the early 1990s a maker of SIMD systems (Thinking Machines) was the largest manufacturer of parallel supercomputers. However, by the late 1990s the only widely produced SIMD systems were vector processors. More recently, graphics processing units, or GPUs, and desktop CPUs are making use of aspects of SIMD computing.

Vector processors

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. Typical recent systems have the following characteristics:

Vector registers: These are registers capable of storing a vector of operands and operating simultaneously on their contents. The vector length is fixed by the system, and can range from 4 to 128 64-bit elements.

Vectorized and pipelined functional units: Note that the same operation is applied to each element in the vector, or, in the case of operations like addition, the same operation is applied to each pair of corresponding elements in the two vectors. Thus, vector operations are SIMD.

Vector instructions: These are instructions that operate on vectors rather than scalars. If the vector length is vector length, these instructions have the great virtue that a simple loop such as

```
for (i = 0; i < n; i++)
        x[i] += y[i];
```

requires only a single load, add, and store for each block of vector length elements, while a conventional system requires a load, add, and store for each element.

Interleaved memory: The memory system consists of multiple ―banks‖ of memory, which can be accessed more or less independently. After accessing one bank, there will be a delay before it can be re-accessed, but a different bank can be accessed much sooner. So if the elements of a vector are distributed across multiple banks, there can be little to no delay in loading/storing successive elements.

Strided memory access and hardware scatter/gather: 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. 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.

Vector processors have the virtue that for many applications, they are very fast and very easy to use. Vectorizing compilers are quite good at identifying code that can be vectorized. Further, they identify loops that cannot be vectorized, and they often provide information about why a loop couldn‘t be vectorized. The user can thereby make informed decisions about whether it‘s possible to rewrite the loop so that it will vectorize. 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. It‘s difficult to see how systems could be created that would operate on ever longer vectors. Current generation systems scale by increasing the number of vector processors, not the vector length. Current commodity systems provide limited support for operations on very short vectors, while processors that operate on long vectors are custom manufactured, and, consequently, very expensive.

Graphics 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. 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. Since the application of a shader function to nearby elements often results in the same flow of control, GPUs can optimize performance by using SIMD parallelism, and in the current generation all GPUs use SIMD parallelism. This is obtained by including a large number of ALUs (e.g., 80) on each GPU processing core.

Processing a single image can require very large amounts of data—hundreds of megabytes of data for a single image is not unusual. GPUs therefore need to maintain very high rates of data movement, and in order to avoid stalls on memory accesses, they rely heavily on hardware multithreading; some systems are capable of storing the state of more than a hundred suspended threads for each executing thread. The actual number of threads depends on the amount of resources (e.g., registers) needed by the shader function. A drawback here is that many threads processing a lot of data are needed to keep the ALUs busy, and GPUs may have relatively poor performance on small problems.

It should be stressed that GPUs are not pure SIMD systems. Although the ALUs on a given core do use SIMD parallelism, current generation GPUs can have dozens of cores, which are capable of executing independent instruction streams.

GPUs are becoming increasingly popular for general, high-performance computing, and several languages have been developed that allow users to exploit their power.

MIMD SYSTEMS

2. Explain briefly about MIMD systems. (16)

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. In many MIMD systems there is no global clock, and there may be no relation between the system times on two different processors. In fact, unless the programmer imposes some synchronization, even if the processors are executing exactly the same sequence of instructions, at any given instant they may be executing different statements.

There are two principal types of MIMD systems: shared-memory systems and distributed-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. 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. See the figures 1.1 and 1.2.

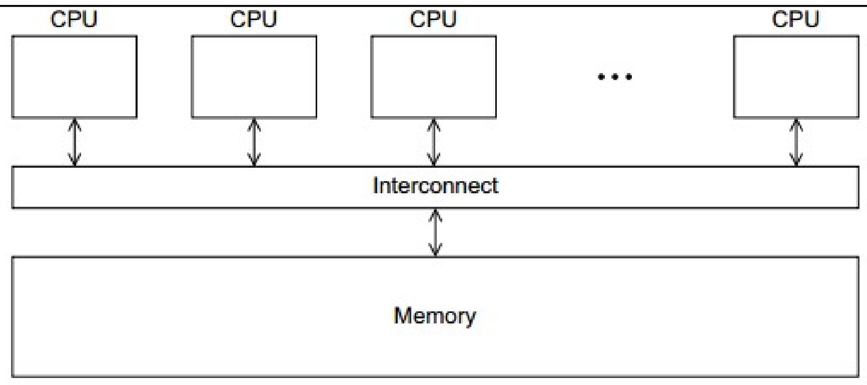


Fig 1.1:
A shared-memory system

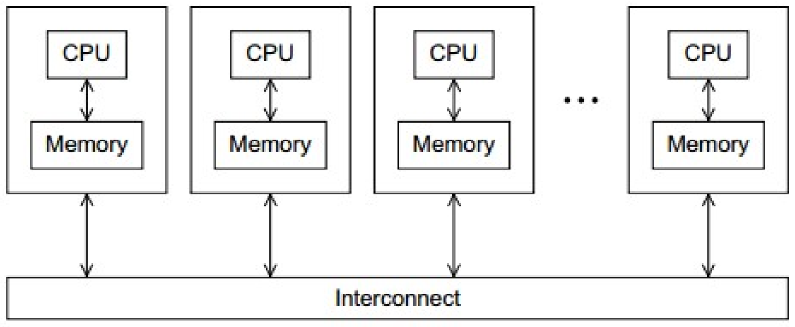


Fig 1.2:
A distributed-memory system

Shared-memory systems

The most widely available shared-memory systems use one or more multicore processors. A multicore processor has multiple CPUs or cores on a single chip. Typically, the cores have private level 1 caches, while other caches may or may not be shared between the cores.

In shared-memory systems with multiple multicore processors, the interconnect can either connect all the processors directly to main memory or each processor can have a direct connection to a block of main memory, and the processors can access each others‘ blocks of main memory through special hardware built into the processors. See Figures 1.3 and 1.4.

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.

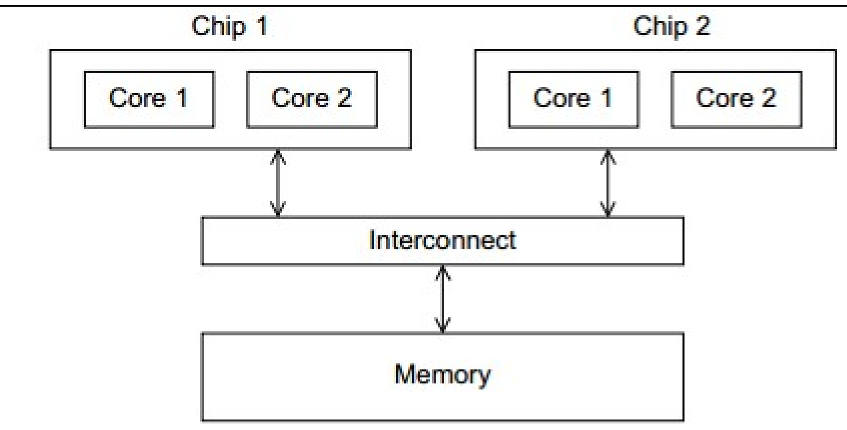


Fig 1.3:

A UMA multicore system

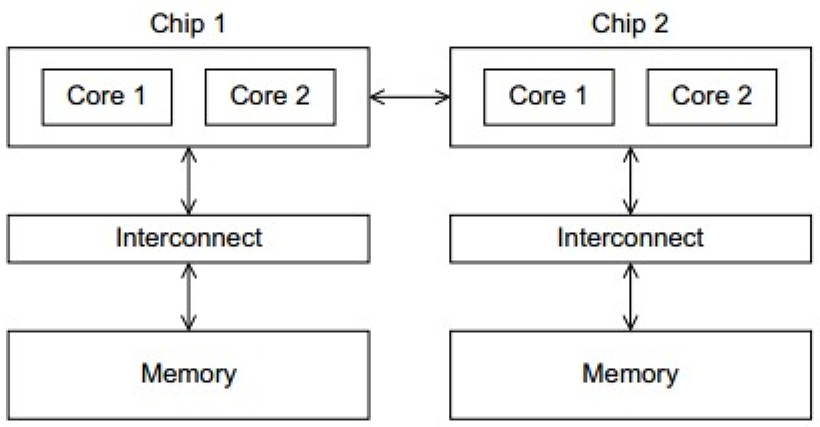


Fig 1.4:

A NUMA multicore system

Thus, the first type of system is called a uniform memory access, or UMA, system, while the second type is called a non-uniform memory access, or NUMA, system. UMA systems are usually easier to program, since the programmer doesn‛t need to worry about different access times for different memory locations. This advantage can be offset by the faster access to the directly connected memory in NUMA systems. Furthermore, NUMA systems have the potential to use larger amounts of memory than UMA systems.

Distributed-memory systems

The most widely available distributed-memory systems are called clusters. They are composed of a collection of commodity systems–for example, PCs–connected by a commodity interconnection network–for example, Ethernet. In fact, 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. Nowadays, it‘s usually understood that a cluster will have shared-memory nodes.

The grid provides the infrastructure necessary to turn large networks of geographically distributed computers into a unified distributed-memory system. In general, such a system will be heterogeneous, that is, the individual nodes may be built from different types of hardware.

INTERCONNECTION NETWORKS

3. Explain in briefly about Interconnection networks. (16)

The interconnect plays a decisive role in the performance of both distributed- and shared-memory systems: even if the processors and memory have virtually unlimited performance, a slow interconnect will seriously degrade the overall performance of all but the simplest parallel program.

Shared-memory interconnects

Currently the two most widely used interconnects on shared-memory systems are buses and crossbars. Recall that a bus is a collection of parallel communication wires together with some hardware that controls access to the bus. The key characteristic of a bus is that the communication wires are shared by the devices that are connected to it. Buses have the virtue of low cost and flexibility; multiple devices can be connected to a bus with little additional cost. However, since the communication wires are shared, as the number of devices connected to the bus increases, the likelihood that there will be contention for use of the bus increases, and the expected performance of the bus decreases. Therefore, if we connect a large number of processors to a bus, we would expect that the processors would frequently have to wait for access to main memory. Thus, as the size of shared-memory systems increases, buses are rapidly being replaced by switched interconnects. As the name suggests, switched interconnects use switches to control the routing of data among the connected devices. A crossbar is illustrated in Figure 1.5. The lines are bidirectional communication links, the squares are cores or memory modules, and the circles are switches.

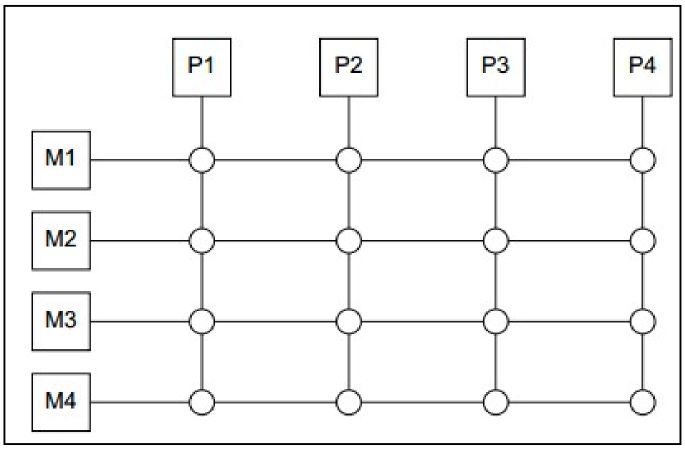


Fig 1.5: A crossbar switch connecting four processors (Pi) and four memory modules (Mj)

The individual switches can assume one of the two configurations shown in Figure 1.6. With these switches and at least as many memory modules as processors, there will only be a conflict between two cores attempting to access memory if the two cores attempt to simultaneously access the same memory module.

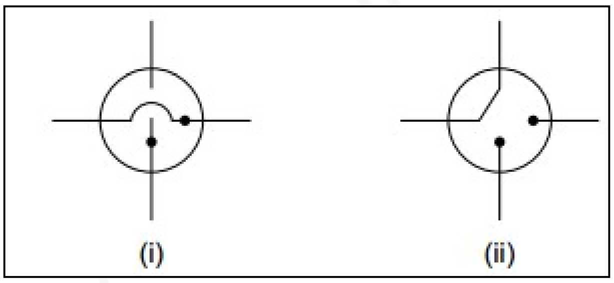


Fig 1.6: Configuration of internal switches in a crossbar

For example, Figure 1.7 shows the configuration of the switches if P1 writes to M4, P2 reads from M3, P3 reads from M1, and P4 writes to M2.

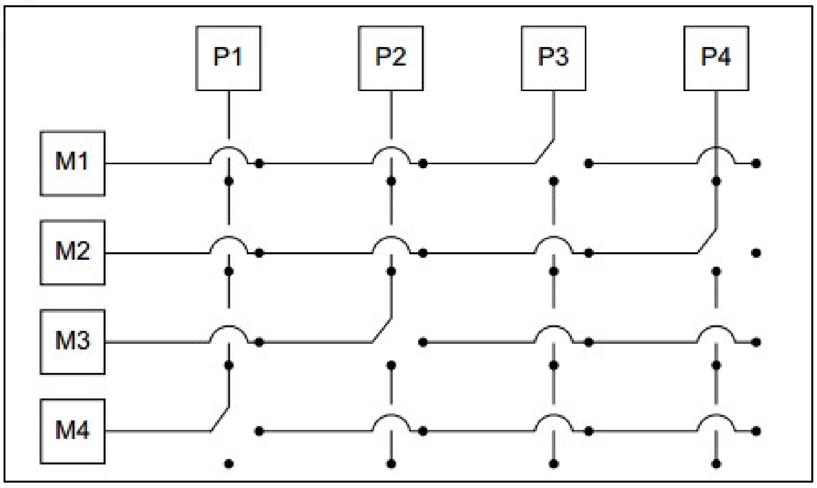


Fig 1.7: Simultaneous memory accesses by the processors

Crossbars allow simultaneous communication among different devices, so they are much faster than buses. However, the cost of the switches and links is relatively high. A small bus-based system will be much less expensive than a crossbar-based system of the same size.

Distributed-memory interconnects

Distributed-memory interconnects are often divided into two groups: direct interconnects and indirect interconnects. In a direct interconnect each switch is directly connected to a processor-memory pair, and the switches are connected to each other. Figure 1.8 shows a ring and a two-dimensional toroidal mesh. As before, the circles are switches, the squares are processors, and the lines are bidirectional links. A ring is superior to a simple bus since it allows multiple simultaneous communications. However, it‘s easy to devise communication schemes in which some of the processors must wait for other processors to complete their communications. The toroidal mesh will be more expensive than the ring, because the switches are more complex–they must support five links instead of three–and if there are p processors, the number of links is 3p in a toroidal mesh, while it‘s only 2p in a ring. However, it‘s not difficult to convince yourself that the number of possible simultaneous communications patterns is greater with a mesh than with a ring.

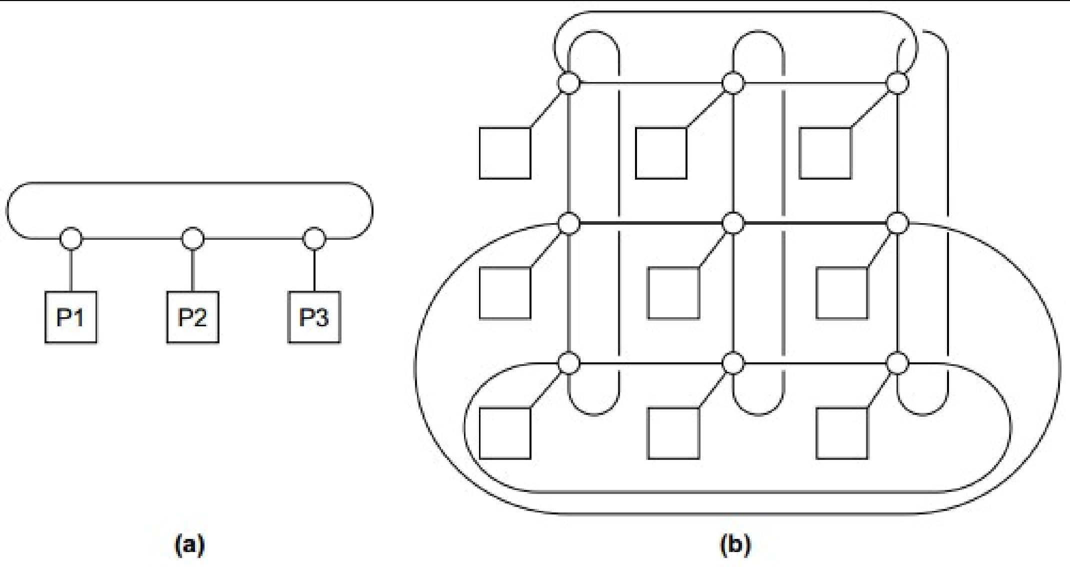


Fig 1.8:

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

One measure of ―number of simultaneous communications‖ or ―connectivity‖ is bisection width. To understand this measure, imagine that the parallel system is divided into two halves, and each half contains half of the processors or nodes.

In Figure 1.9(a) we‘ve divided a ring with eight nodes into two groups of four nodes, and we can see that only two communications can take place between the halves. In Figure 1.9(b) we‘ve divided the nodes into two parts so that four simultaneous communications can take place.

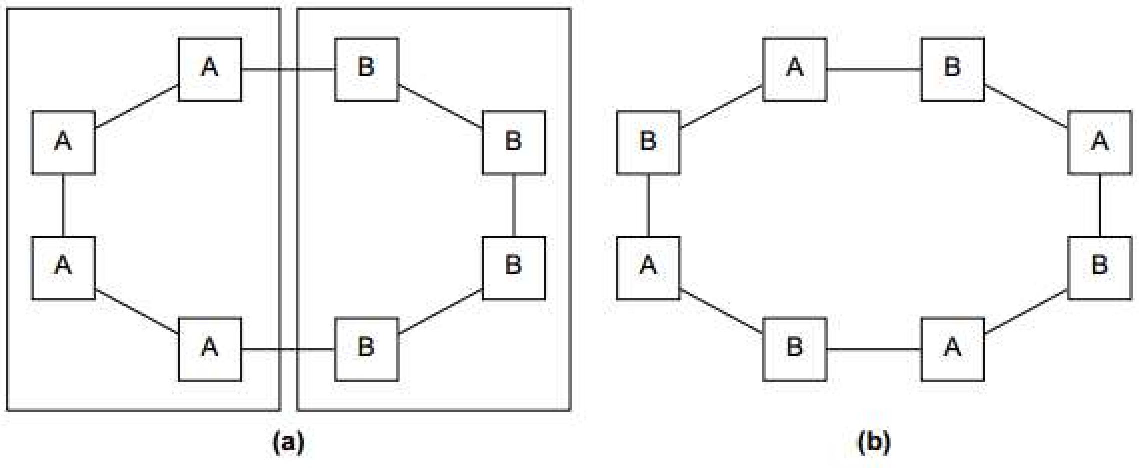


Fig 1.9:

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

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. For example, if the links in a ring have a bandwidth of one billion bits per second, then the bisection bandwidth of the ring will be two billion bits per second or 2000 megabits per second.

Latency and bandwidth in interconnection networks

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

CACHE COHERENCE

4. Explain in detail about cache coherence in parallel hardware systems. (16)

Cache coherence

In a shared memory multiprocessor with a separate cache memory for each processor , it is possible to have many copies of any one instruction operand : one copy in the main memory and one in each cache memory. When one copy of an operand is changed, the other copies of the operand must be changed also. Cache coherence is the discipline that ensures that changes in the values of shared operands are propagated throughout the system in a timely fashion.

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

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. The principal difference between our description and the actual snooping protocol is that the broadcast only informs the other cores that the cache line containing x has been updated, not that x has been updated.

A couple of points should be made regarding snooping. First, it‘s not essential that the interconnect be a bus, only that it support broadcasts from each processor to all the other processors. Second, snooping works with both write-through and write-back caches. In principle, if the interconnect is shared–as with a bus–with write through caches there‘s no need for additional traffic on the interconnect, since each core can simply ―watch‖ for writes. With write-back caches, on the other hand, an extra communication is necessary, since updates to the cache don‘t get immediately sent to memory.

Directory-based cache coherence

Unfortunately, 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. For example, suppose we have a system with the basic distributed-memory architecture. However, the system provides a single address space for all the memories. So, for example, core 0 can access the variable x stored in core 1‘s memory, by simply executing a statement such as y = x. Such a system can, in principle, scale to very large numbers of cores. However, snooping cache coherence is clearly a problem since a broadcast across the interconnect will be very slow relative to the speed of accessing local memory.

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. Thus, when a line is read into, say, core 0‘s cache, the directory entry corresponding to that line would be updated indicating that core 0 has a copy of the line. When a variable is updated, the directory is consulted, and the cache controllers of the cores that have that variable‘s cache line in their caches are invalidated.

Clearly there will be substantial additional storage required for the directory, but when a cache variable is updated, only the cores storing that variable need to be contacted.

False sharing

It‘s important to remember that CPU caches are implemented in hardware, so they operate on cache lines, not individual variables. This can have disastrous consequences for performance. As an example, suppose we want to repeatedly call a function f(i,j) and add the computed values into a vector:

```
int i, j, m, n;
double y[m];
/* Assign y = 0 */
. . .
for (i = 0; i < m; i++)
for (j = 0; j < n; j++)
```

```
y[i] += f(i,j);
```

We can parallelize this by dividing the iterations in the outer loop among the cores. If we have core count cores, we might assign the first m/core count iterations to the first core, the next m/core count iterations to the second core, and so on.

```
/* Private variables */
int i, j, iter count;
/* Shared variables initialized by one core */
int m, n, core count
double y[m];
iter count = m/core count
/* Core 0 does this */
for (i = 0; i < iter count; i++)
for (j = 0; j < n; j++)
y[i] += f(i,j);
/* Core 1 does this */
for (i = iter count+1; i < 2*iter_count; i++)
for (j = 0; j < n; j++)
y[i] += f(i,j);
. . .
```

Now suppose our shared-memory system has two cores, m = 8, doubles are eight bytes, cache lines are 64 bytes, and y[0] is stored at the beginning of a cache line. A cache line can store eight doubles, and y takes one full cache line. What happens when core 0 and core 1 simultaneously execute their codes? Since all of y is stored in a single cache line, each time one of the cores executes the statement y[i] += f(i,j), the line will be invalidated, and the next time the other core tries to execute this statement it will have to fetch the updated line from memory! 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.

PERFORMANCE ISSUES

5. Discuss in detail about various performance issues of multicore processors.(16)

i) Speedup and efficiency

Usually the best we can hope to do is to equally divide the work among the cores, while at the same time introducing no additional work for the cores. If we succeed in doing this, and we run our program with p cores, one thread or process on each core, then our parallel program will run p times faster than the serial program. If we call the serial run-time T_{serial} and our parallel run-time $T_{parallel}$, then the best we can hope for is ***$T_{parallel} = T_{serial}/p$***. When this happens, we say that our parallel program has linear speedup.

For example, shared memory programs will almost always have critical sections, which will require that we use some mutual exclusion mechanism such as a mutex. The calls to the mutex functions are overhead that‘s not present in the serial program, and the use of the mutex forces the parallel program to serialize execution of the critical section. Distributed-memory programs will almost always need to transmit data across the network, which is usually much slower than local memory access. Serial programs, on the other hand, won‘t have these overheads. Thus, it will be very unusual for us to find that our parallel programs get linear speedup. Furthermore, it‘s likely that the overheads will increase as we increase the number of processes or threads, that is, more threads will probably mean more threads need to access a critical section. More processes will probably mean more data needs to be transmitted across the network.

So if we define the speedup of a parallel program to be

$$\boxed{S = \frac{T_{\text{serial}}}{T_{\text{parallel}}},}$$

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. This value, S=p, is sometimes called the efficiency of the parallel program. If we substitute the formula for S, we see that the efficiency is

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

ii) Amdahl’s law

It says, roughly, that unless virtually all of a serial program is parallelized, the possible speedup is going to be very limited—regardless of the number of cores available. Suppose, for example, that we‘re able to parallelize 90% of a serial program. Further suppose that the parallelization is ¯perfect,‖ that is, regardless of the number of cores p we use, the speedup of this part of the program will be p. If the serial run-time is **T_{Serial} = 20** seconds, then the run-time of the parallelized part will be 0.9 * T_{serial}/p = 18/p and the run-time of the ―unparallelized‖ part will be 0.1 * T_{serial} = 2. The overall parallel run-time will be

$$\boxed{T_{\text{parallel}} = 0.9 \times T_{\text{serial}}/p + 0.1 \times T_{\text{serial}} = 18/p + 2,}$$

and the speedup will be

$$\boxed{S = \frac{T_{\text{serial}}}{0.9 \times T_{\text{serial}}/p + 0.1 \times T_{\text{serial}}} = \frac{20}{18/p + 2}.}$$

Now as p gets larger and larger, 0.9 * T_{serial}/p = 18/p gets closer and closer to 0, so the total parallel run-time can‘t be smaller than 0.1 * T_{serial} = 2. That is, the denominator in S can‘t be smaller than 0.1 * T $_{serial}$ = 2. The fraction S must therefore be smaller than

$$\boxed{S \le \frac{T_{\text{serial}}}{0.1 \times T_{\text{serial}}} = \frac{20}{2} = 10.}$$

That is, S ≤ 10.

iii) Scalability

Suppose we run a parallel program with a fixed number of processes/threads and a fixed input size, and we obtain an efficiency E. Suppose we now increase the number of processes/threads that are used by the program. 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_{serial} = n, where the units of T $_{serial}$ are in microseconds, and n is also the problem size. Also suppose that $T_{parallel}$ = n/p + 1. Then

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

To see if the program is scalable, we increase the number of processes/threads by a factor of k, and we want to find the factor x that we need to increase the problem size by so that E is unchanged. The number of processes/threads will be kp and the problem size will be xn, and we want to solve the following equation for x:

$$\boxed{E = \frac{n}{n+p} = \frac{xn}{xn+kp}.}$$

Well, if x = k, there will be a common factor of k in the denominator xn + kp = kn + kp = k(n + p), and we can reduce the fraction to get

$$\boxed{\frac{xn}{xn+kp} = \frac{kn}{k(n+p)} = \frac{n}{n+p}.}$$

In other words, if we increase the problem size at the same rate that we increase the number of processes/threads, then the efficiency will be unchanged, and our program is scalable.

iv) Talking timings

The first thing to note is that there are at least two different reasons for taking timings. During program development we may take timings in order to determine if the program is behaving as we intend. For example, in a distributed-memory program we might be interested in finding out how much time the processes are spending waiting for messages, because if this value is large, there is almost certainly something wrong either with our design or our implementation. On the other hand, once we‘ve completed development of the program, we‘re often interested in determining how good its performance is.

When we‘re timing parallel programs, we need to be a little more careful about how the timings are taken. In our example, the code that we want to time is probably being executed by multiple processes or threads and our original timing will result in the output of ***p*** elapsed times.

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

UNIT-2 PARALLEL PROGRAM CHALLENGES

PART-A

1. What are named pipes?

Named pipes are file-like objects that are given a specific name that can be shared between processes. Any process can write into the pipe or read from the pipe. There is no concept of a ―message‖; the data is treated as a stream of bytes. The method for using a named pipe is much like the method for using a file: The pipe is opened, data is written into it or read from it, and then the pipe is closed.

2. In a parallel programming what a data race occurs?

A data race occurs when:

- two or more threads in a single process access the same memory location concurrently, and
- at least one of the accesses is for writing, and
- the threads are not using any exclusive locks to control their accesses to that memory.

3. Define: Mutex.

In computer programming, a mutex is a program object that allows multiple program threads to share the same resource, such as file access, but not simultaneously. When a program is started, a mutex is created with a unique name. After this stage, any thread that needs the resource must lock the mutex from other threads while it is using the resource. The mutex is set to unlock when the data is no longer needed or the routine is finished

4. Define: Semaphore.

Semaphore is a variable or abstract data type that is used to control access to a common resource by multiple processes in a concurrent system such as a multiprogramming operating system. A trivial semaphore is a plain variable that is changed (for example, incremented or decremented, or toggled) depending on programmer-defined conditions. The variable is then used as a condition to control access to some system resource.

5. Define: Deadlock.

In concurrent computing, a deadlock is a state in which each member of a group of actions, is waiting for some other member to release a lock. Deadlock is a common problem in multiprocessing systems, parallel computing, and distributed systems, where software and hardware locks are used to handle shared resources and implement process synchronization.

6. What is condition variable?

A condition variable is basically a container of threads that are waiting for a certain condition. Monitors provide a mechanism for threads to temporarily give up exclusive access in order to wait for some condition to be met, before regaining exclusive access and resuming their task.

7. What is meant by message queue?

A message queue is a queue of messages sent between applications. It includes a sequence of work objects that are waiting to be processed. A message is the data transported between the sender and and the receiver application, it's essentially a byte array with some headers on top.

8. Why Algorithmic Complexity Is Important?

Algorithmic complexity represents the expected performance of a section of code as the number of elements being processed increases. In the limit, the code with the greatest algorithmic complexity will dominate the runtime of the application.

9. What is meant by cache conflict miss?

A conflict cache miss is where one thread has caused data needed by another thread to be evicted from the cache. The worst example of this is thrashing where multiple threads each require an item of data and that item of data maps to the same cache line for all the threads.

10. Define: Livelock.

A livelock is similar to a deadlock, except that the states of the processes involved in the livelock constantly change with regard to one another, none progressing. Livelock is a special case of resource starvation; the general definition only states that a specific process is not progressing.

PART-B

PERFORMANCE

1. Discuss briefly about the performance metrics of the parallel programming.(16)

There are two common metrics for performance:

Items per unit time: This might be transactions per second, jobs per hour, or some other combination of completed tasks and units of time. Essentially, this is a measure of bandwidth. It places the emphasis on the ability of the system to complete tasks rather than on the duration of each individual task. Many benchmarks are essentially a measure of bandwidth. If you examine the SPEC Java Application Server benchmark (SPEC jAppServer[1]), you'll find that final results are reported as transactions per second. Another example is the linpack benchmark used as a basis for the TOP500[2] list of supercomputers. The metric that is used to form the TOP500 list is the peak number of floating-point operations per second.

Time per item: This is a measure of the time to complete a single task. It is basically a measure of latency or response time. Fewer benchmarks specifically target latency. The most obvious example of a latency-driven benchmark is the SPEC CPU benchmark suite, which has a speed metric as well as a rate metric.

Although these are both common expressions of performance, it can be specified as a more complex mix. For example, the results that e-commerce benchmark SPECweb publishes are the number of simultaneous users that a system can support, subject to meeting criteria on throughput and response time. Many systems have a quality of service (QoS) metric that they must meet. The QoS metric will specify the expectations of the users of the system as well as penalties if the system fails to meet these expectations. These are two examples of alternative metrics:

- Number of transactions of latency greater than some threshold. This will probably be set together with an expectation for the average transaction. It is quite possible to have a system that exceeds the criteria for both the number of transactions per second that it supports and the average response time for a transaction yet have that same system fail due to the criteria for the number of responses taking longer than the threshold.
- The amount of time that the system is unavailable, typically called downtime or availability. This could be specified as a percentage of the time that the system is expected to be up or as a number of minutes per year that the system is allowed to be down.

The metrics that are used to specify and measure performance have many ramifications in the design of a system to meet those metrics. Consider a system that receives a nightly update. Applying this nightly update will make the system unavailable. Using the metrics that specify availability, it is possible to determine the maximum amount of time that the update can take while still meeting the availability criteria. If the designer knows that the system is allowed to be down for ten minutes per night, then they will make different design decisions than if the system has only a second to complete the update.

Knowing the available time for an update might influence the following decisions:

- How many threads should be used to process the update. A single thread may not be sufficient to complete the update within the time window. Using the data, it should be possible to estimate the number of threads that would be needed to complete the update within the time window. This will have ramifications for the design of the application, and it may even have ramifications for the method and format used to deliver the data to the application.
- If the update has to be stored to disk, then the write bandwidth of the disk storage becomes a consideration. This may be used to determine the number of drives necessary to deliver that bandwidth, the use of solid-state drives, or the use of a dedicated storage appliance.
- If the time it takes to handle the data, even with using multiple threads or multiple drives, exceeds the available time window, then the application might have to be structured so that the update can be completed in parallel with the application processing incoming transactions. Then the application can instantly switch between the old and new data. This kind of design might have some underlying complexities if there are pending transactions at the time of the swap. These transactions would need to either complete using the older data or be restarted to use the latest version.

In fact, defining the critical metrics and their expectations early in the design helps with three tasks:

- Clearly specified requirements can be used to drive design decisions, both for selecting appropriate hardware and in determining the structure of the software.
- Knowing what metrics are expected enables the developers to be confident that the system they deliver fulfills the criteria. Having the metrics defined up front makes it easy to declare a project a success or a failure.

- Defining the metrics should also define the expected inputs. Knowing what the inputs to the system are likely to look like will enable the generation of appropriate test cases. These test cases will be used by the designers and developers to test the program as it evolves.

DATA RACES

2. Explain in detail about data races. (16)

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. Suppose you have the code shown in Figure 2.1, where a pointer to an integer variable is passed in and the function increments the value of this variable by 4.

Fig 2.1: Updating the Value at an Address

```
void update(int * a)
{
   *a = *a + 4;
}
```

The SPARC disassembly for this code would look something like the code shown in Figure 2.2.

Fig 2.2: SPARC Disassembly for Incrementing a Variable Held in Memory

```
ld  [%o0], %o1   // Load *a
add  %o1, 4, %o1 // Add 4
st   %o1, [%o0]  // Store *a
```

Suppose this code occurs in a multithreaded application and two threads try to increment the same variable at the same time. Figure 2.3 shows the resulting instruction stream.

Fig 2.3: Two Threads Updating the Same Variable

Value of variable a = 10	
Thread 1	**Thread 2**
`ld  [%o0], %o1  // Load %o1 = 10`	`ld  [%o0], %o1  // Load %o1 = 10`
`add %01, 4, %o1 // Add  %o1 = 14`	`add %01, 4, %o1 // Add %o1 = 14`
`st  %o1, [%o0]  // Store %o1`	`st  %o1, [%o0]  // Store %o1`
Value of variable a = 14	

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.

Using tools to detect data races

The code shown in Figure 2.4 contains a data race. The code uses POSIX threads. The code creates two threads, both of which execute the routine func(). The main thread then waits for both the child threads to complete their work.

Fig 2.4: Code Containing Data Race

```
#include <pthread.h>

int counter = 0;

void * func(void * params)
{
   counter++;
}

void main()
{
  pthread_t thread1, thread2;
  pthread_create( &thread1, 0, func, 0);
  pthread_create( &thread2, 0, func, 0);
  pthread_join( thread1, 0 );
  pthread_join( thread2, 0 );
}
```

Both threads will attempt to increment the variable counter. We can compile this code with GNU gcc and then use Helgrind, which is part of the Valgrind suite, to identify the data race. Valgrind is a tool that enables an application to be instrumented and its runtime behavior examined. The Helgrind tool uses this instrumentation to gather data about data races. Figure 2.5 shows the output from Helgrind.

Fig 2.5: Using Helgrind to Detect Data Races

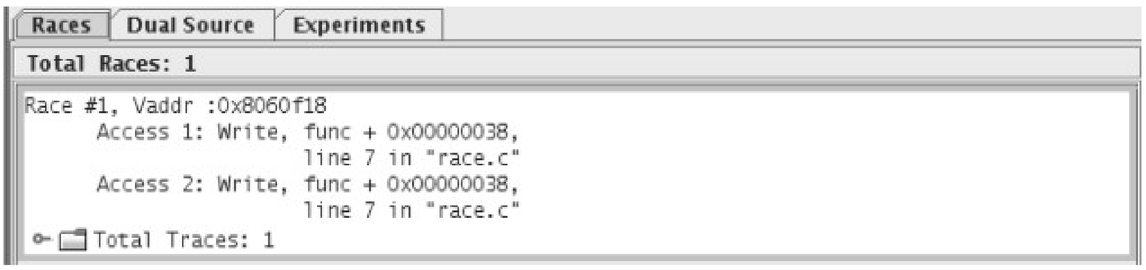
```
$ gcc -g race.c -lpthread
$ valgrind --tool=helgrind ./a.out
...
==4742==
==4742== Possible data race during write of size 4
         at 0x804a020 by thread #3
==4742==    at 0x8048482: func (race.c:7)
==4742==    by 0x402A89B: mythread_wrapper (hg_intercepts.c:194)
==4742==    by 0x40414FE: start_thread
            (in /lib/tls/i686/cmov/libpthread-2.9.so)
==4742==    by 0x413849D: clone (in /lib/tls/i686/cmov/libc-2.9.so)
==4742==  This conflicts with a previous write of size 4 by thread #2
==4742==    at 0x8048482: func (race.c:7)
==4742==    by 0x402A89B: mythread_wrapper (hg_intercepts.c:194)
==4742==    by 0x40414FE: start_thread
            (in /lib/tls/i686/cmov/libpthread-2.9.so)
==4742==    by 0x413849D: clone (in /lib/tls/i686/cmov/libc-2.9.so)
```

The output from Helgrind shows that there is a potential data race between two threads, both executing line 7 in the file race.c. This is the anticipated result, but it should be pointed out that the tools will find some false positives. The programmer may write code where different threads access the same variable, but the programmer may know that there is an enforced order that stops an actual data race. The tools, however, may not be able to detect the enforced order and will report the potential data race.

Another tool that is able to detect potential data races is the Thread Analyzer in Oracle Solaris Studio. This tool requires an instrumented build of the application, data collection is done by the collect tool, and the graphical interface is launched with the command tha. Figure 2.6 shows the steps to do this.

Fig 2.6: Detecting Data Races Using the Sun Studio Thread Analyzer

```
$ cc -g -xinstrument=datarace race.c
$ collect -r on ./a.out
Recording experiment tha.1.er ...
$ tha tha.1.er&
```

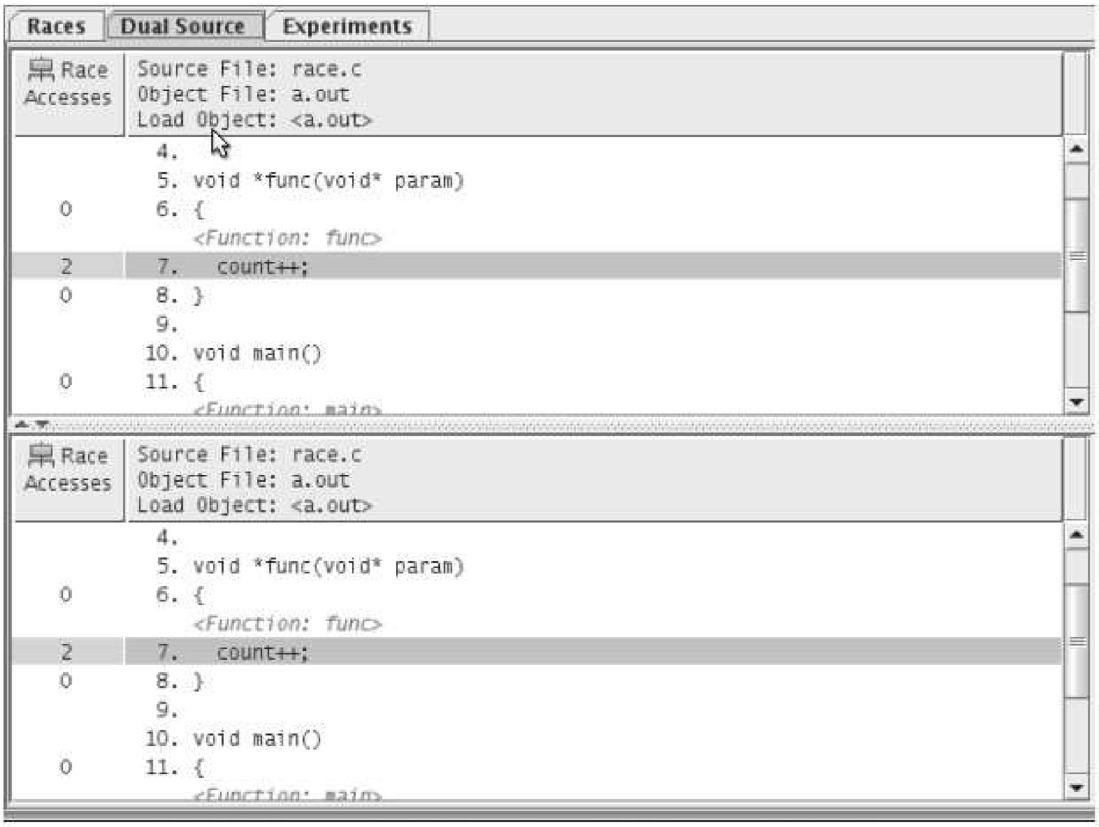


```
Total Races: 1
Race #1, Vaddr :0x8060f18
      Access 1: Write, func + 0x00000038,
                       line 7 in "race.c"
      Access 2: Write, func + 0x00000038,
                       line 7 in "race.c"
Total Traces: 1
```

Fig 2.7: List of data races detected by the Solaris Studio Thread Analyzer

The initial screen of the tool displays a list of data races, as shown in Figure 2.7.

Once the user has identified the data race they are interested in, they can view the source code for the two locations in the code where the problem occurs. In the example, shown in Figure 2.8, both threads are executing the same source line.



```
Source File: race.c
Object File: a.out
Load Object: <a.out>
      4.
      5. void *func(void* param)
0     6. {
         <Function: func>
2     7.   count++;
0     8. }
      9.
     10. void main()
0    11. {
         <Function: main>
```

```
Source File: race.c
Object File: a.out
Load Object: <a.out>
      4.
      5. void *func(void* param)
0     6. {
         <Function: func>
2     7.   count++;
0     8. }
      9.
     10. void main()
0    11. {
         <Function: main>
```

Fig 2.8: Source code with data race shown in Solaris Studio Thread Analyzer

Avoiding data races

Although 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. Figure 2.9 shows a modified version of the code. This version uses a mutex lock, described in more detail in the next section, to protect accesses to the variable counter. Although this ensures the correctness of the code, it does not necessarily give the best performance.

Fig 2.9: Code Modified to Avoid Data Races

```
void * func( void * params )
{
  pthread_mutex_lock( &mutex );
  counter++;
  pthread_mutex_unlock( &mutex );
}
```

SYNCHRONIZATION PRIMITIVES

3. What are synchronization primitives? Explain in detail about the following synchronization primitives: (16)

a) Mutexes and critical regions

b) Spin locks

c) Semaphores

d) Readers-Writer locks

e) Barriers

Synchronization is used to coordinate the activity of multiple threads. There are various situations where it is necessary; this might be to ensure that shared resources are not accessed by multiple threads simultaneously or that all work on those resources is complete before new work starts. Synchronization primitives provided by the operating system will usually be recognized by the tools provided with that operating system. The operating system will often provide support for sharing the primitives between threads or processes, which can be hard to do efficiently without operating system support.

a) Mutexes and critical regions

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. Figure 2.10 shows how a mutex lock could be used to protect access to a variable.

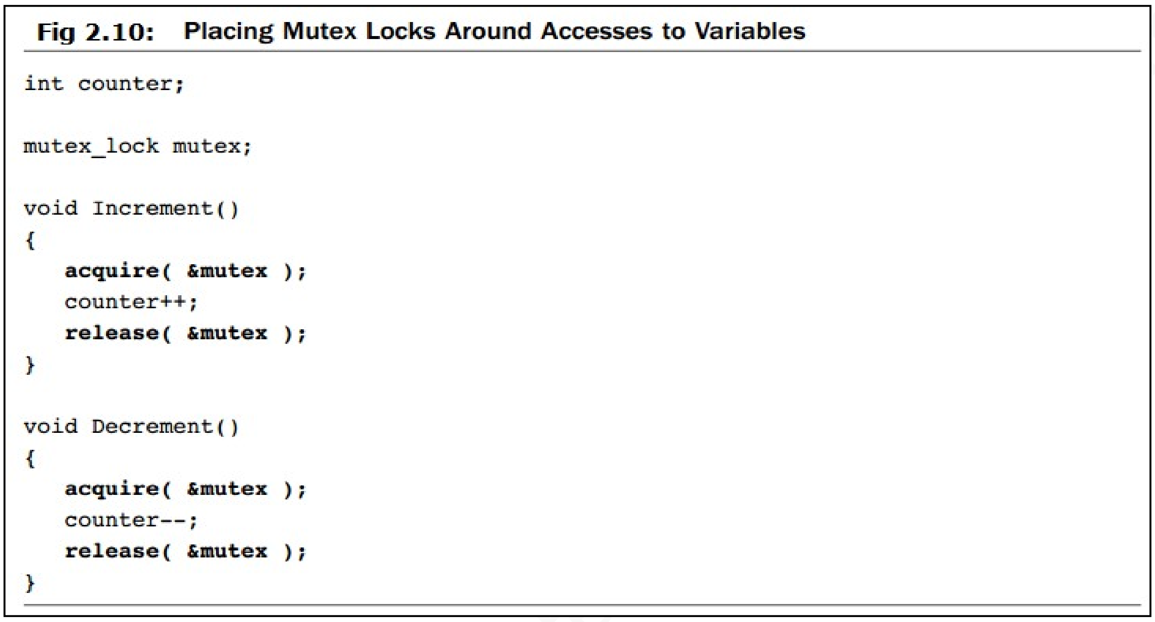


Fig 2.10: Placing Mutex Locks Around Accesses to Variables

```
int counter;

mutex_lock mutex;

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

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

In the example, the two routines Increment() and Decrement()will either increment or decrement the variable counter. To modify the variable, a thread has to first acquire the mutex lock. Only one thread at a time can do this; all the other threads that want to acquire the lock need to wait until the thread holding the lock releases it. Both routines use the same mutex; consequently, only one thread at a time can either increment or decrement the variable counter.

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.

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.

As an example of a critical section, imagine that an operating system does not have an implementation of malloc()that is thread-safe, or safe for multiple threads to call at the same time. One way to fix this is to place the call to malloc()in a critical section by surrounding it with a mutex lock, as shown in Figure 2.11.

Fig 2.11 Placing a Mutex Lock Around a Region of Code

```
void * threadSafeMalloc( size_t size )
{
   acquire( &mallocMutex );
   void * memory = malloc( size );
   release( &mallocMutex );
   return memory;
}
```

If all the calls to malloc()are replaced with the threadSafeMalloc()call, then only one thread at a time can be in the original malloc()code, and the calls to malloc()become thread-safe.

Threads block if they attempt to acquire a mutex lock that is already held by another thread. Blocking means that the threads are sent to sleep either immediately or after a few unsuccessful attempts to acquire the mutex.

One problem with this approach is that it can serialize a program. If multiple threads simultaneously call threadSafeMalloc(), only one thread at a time will make progress. This causes the multithreaded program to have only a single executing thread, which stops the program from taking advantage of multiple cores.

b) Spin locks

Spin locks are essentially mutex locks. The difference between a mutex lock and a spin lock is that a thread waiting to acquire a spin lock will keep trying to acquire the lock without sleeping. In comparison, a mutex lock may sleep if it is unable to acquire the lock. The advantage of using spin locks is that they will acquire the lock as soon as it is released, whereas a mutex lock will need to be woken by the operating system before it can get the lock. The disadvantage is that a spin lock will spin on a virtual CPU monopolizing that resource. In comparison, a mutex lock will sleep and free the virtual CPU for another thread to use.

Often mutex locks are implemented to be a hybrid of spin locks and more traditional mutex locks. The thread attempting to acquire the mutex spins for a short while before blocking. There is a performance advantage to this. Since 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. So, spinning for a short period of time makes it more likely that the waiting thread will acquire the mutex lock as soon as it is released. However, continuing to spin for a long period of time consumes hardware resources that could be better used in allowing other software threads to run.

c) 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.

Semaphores can also be used to mimic mutexes; if there is only one element in the semaphore, then it can be either acquired or available, exactly as a mutex can be either locked or unlocked.

Semaphores will also signal or wake up threads that are waiting on them to use available resources; hence, they can be used for signaling between threads. For example, a thread might set a semaphore once it has completed some initialization. Other threads could wait on the semaphore and be signaled to start work once the initialization is complete.

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. When the semaphore no longer has resources available, the threads requesting resources will block until resources are available.

d) Readers-Writer locks

Data races are a concern only when shared data is modified. Multiple threads reading the shared data do not present a problem. Read-only data does not, therefore, need protection with some kind of lock.   However, sometimes data that is typically read-only needs to be updated. A readerswriter 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 diminish and will eventually allow the writer to get exclusive access to the lock.

The code snippet in Figure 2.12 shows how a readers-writer lock might be used. Most threads will be calling the routine readData()to return the value from a particular pair of cells. Once a thread has a reader lock, they can read the value of the pair of cells, before releasing the reader lock.

To modify the data, a thread needs to acquire a writer lock. This will stop any reader threads from acquiring a reader lock. Eventually all the reader threads will have released their lock, and only at that point does the writer thread actually acquire the lock and is allowed to update the data.

Fig 2.12: **Using a Readers-Writer Lock**

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

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

e) 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 a dependence between different sections of code. For example, suppose a number of threads compute the values stored in a matrix. The variable total needs to be calculated using the values stored in the matrix. A barrier can be used to ensure that all the threads complete their computation of the matrix before the variable total is calculated. Figure 2.13 shows a situation using a barrier to separate the calculation of a variable from its use.

Fig 2.13: Using a Barrier to Order Computation

```
Compute_values_held_in_matrix();
Barrier();
total = Calculate_value_from_matrix();
```

The variable total can be computed only when all threads have reached the barrier. This avoids the situation where one of the threads is still completing its computations while the other threads start using the results of the calculations. Notice that another barrier could well be needed after the computation of the value for total if that value is then used in further calculations. Figure 2.14 shows this use of multiple barriers.

Fig 2.14: Use of Multiple Barriers

```
Compute_values_held_in_matrix();
Barrier();
total = Calculate_value_from_matrix();
Barrier();
Perform_next_calculation( total );
```

DEADLOCKS AND LIVELOCKS

4. Explain the following:

a) Deadlocks **(8)**

b) Livelocks **(8)**

a) Deadlocks:

Deadlock is a situation where two or more threads cannot make progress because the resources that they need are held by the other threads. It is easiest to explain this with an example. 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. Figure 2.22 shows this situation.

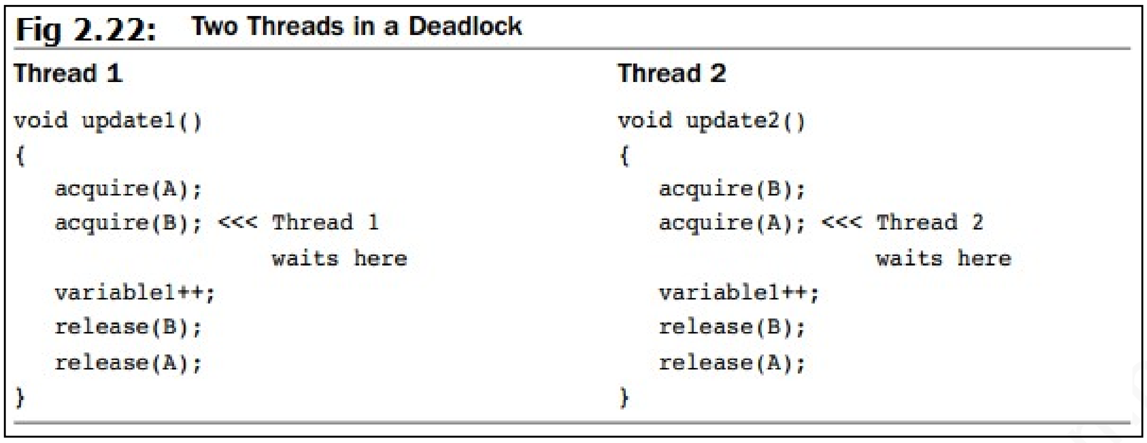


Fig 2.22: Two Threads in a Deadlock

Thread 1

```
void update1()
{
   acquire(A);
   acquire(B); <<< Thread 1
                   waits here
   variable1++;
   release(B);
   release(A);
}
```

Thread 2

```
void update2()
{
   acquire(B);
   acquire(A); <<< Thread 2
                   waits here
   variable1++;
   release(B);
   release(A);
}
```

The best way to avoid deadlocks is to ensure that threads always acquire the locks in the same order. So if thread 2 acquired the locks in the order A and then B, it would stall while waiting for lock A without having first acquired lock B. This would enable thread 1 to acquire Band then eventually release both locks, allowing thread 2 to make progress.

b) Livelocks:

A livelock traps threads in an unending loop releasing and acquiring locks. Livelocks can be caused by code to back out of deadlocks. In Figure 2.23, 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.

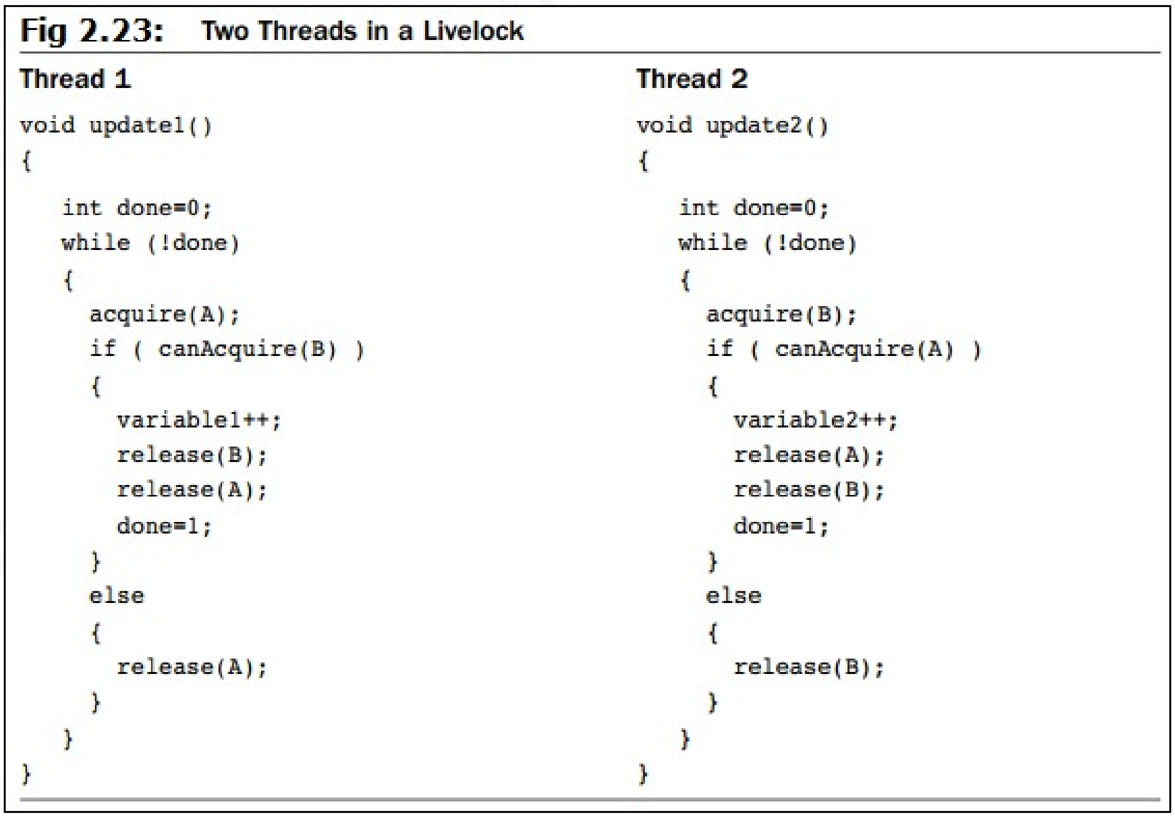


Fig 2.23: Two Threads in a Livelock

Thread 1

```
void update1()
{
   int done=0;
   while (!done)
   {
     acquire(A);
     if ( canAcquire(B) )
     {
       variable1++;
       release(B);
       release(A);
       done=1;
     }
     else
     {
       release(A);
     }
   }
}
```

Thread 2

```
void update2()
{
   int done=0;
   while (!done)
   {
     acquire(B);
     if ( canAcquire(A) )
     {
       variable2++;
       release(A);
       release(B);
       done=1;
     }
     else
     {
       release(B);
     }
   }
}
```

The two routines update1() and update2() each have an outer loop. Routine update1() acquires lock A and then attempts to acquire lock B, whereas update2() does this in the opposite order. This is a classic deadlock opportunity, and to avoid it, the developer has written some code that causes the held lock to be released if it is not possible to acquire the second lock. The routine canAquire(), in this example, returns immediately either having acquired the lock or having failed to acquire the lock.

If two threads encounter this code at the same time, they will be trapped in a livelock of constantly acquiring and releasing mutexes, but it is very unlikely that either will make progress. Each thread acquires a lock and then attempts to acquire the second lock that it needs. If it fails to acquire the second lock, it releases the lock it is holding, before attempting to acquire both locks again. The thread exits the loop when it manages to successfully acquire both locks, which will eventually happen, but until then, the application will make no forward progress.

COMMUNICATION BETWEEN THREADS

5. Explain how communication between threads is achieved by condition variables, signals, message queues and pipes? (16)

All parallel applications require some element of communication between either the threads or the processes. There is usually an implicit or explicit action of one thread sending data to another thread. For example, one thread might be signaling to another that work is ready for them. The following elements outline various mechanisms to enable processes or threads to pass messages or share data.

Condition variables:

Condition variables communicate readiness between threads by enabling a thread to be woken up when a condition becomes true. Without condition variables, the waiting thread would have to use some form of polling to check whether the condition had become true.

Condition variables work in conjunction with a mutex. The mutex is there to ensure that only one thread at a time can access the variable. For example, the producer consumer model can be implemented using condition variables. Suppose an application has one producer thread and one consumer thread. The producer adds data onto a queue, and the consumer removes data from the queue. If there is no data on the queue, then the consumer needs to sleep until it is signaled that an item of data has been placed on the queue. Figure 2.15 shows the pseudocode for a producer thread adding an item onto the queue.

Fig 2.15: Producer Thread Adding an Item to the Queue

```
Acquire Mutex();
Add Item to Queue();
If ( Only One Item on Queue )
{
   Signal Conditions Met();
}
Release Mutex();
```

The producer thread needs to signal a waiting consumer thread only if the queue was empty and it has just added a new item into that queue. If there were multiple items already on the queue, then the consumer thread must be busy processing those items and cannot be sleeping. If there were no items in the queue, then it is possible that the consumer thread is sleeping and needs to be woken up. Figure 2.16 shows the pseudocode for the consumer thread.

Fig 2.16: Code for Consumer Thread Removing Items from Queue

```
Acquire Mutex();
Repeat
  Item = 0;
  If ( No Items on Queue() )
  {
     Wait on Condition Variable();
  }
  If (Item on Queue())
  {
     Item = remove from Queue();
  }
Until ( Item != 0 );
Release Mutex();
```

The consumer thread will wait on the condition variable if the queue is empty. When the producer thread signals it to wake up, it will first check to see whether there is anything on the queue. It is quite possible for the consumer thread to be woken only to find the queue empty; it is important to realize that the thread waking up does not imply that the condition is now true, which is why the code is in a repeat loop in the example. If there is an item on the queue, then the consumer thread can handle that item; otherwise, it returns to sleep.

Signals:

Signals are a UNIX mechanism where one process can send a signal to another process and have a handler in the receiving process perform some task upon the receipt of the message. Many features of UNIX are implemented using signals. Stopping a running application by pressing ^C causes a SIGKILL signal to be sent to the process. Windows has a similar mechanism for events. The handling of keyboard presses and
mouse moves are performed through the event mechanism. Pressing one of the buttons on the mouse will cause a click event to be sent to the target window.

Signals and events are really optimized for sending limited or no data along with the signal, and as such they are probably not the best mechanism for communication when compared to other options.

Figure 2.17 shows how a signal handler is typically installed and how a signal can be sent to that handler. Once the signal handler is installed, sending a signal to that thread will cause the signal handler to be executed.

Fig 2.17 Installing and Using a Signal Handler

```
void signalHandler(void *signal)
{
  ...
}

int main()
{
  installHandler( SIGNAL, signalHandler );
  sendSignal( SIGNAL );
}
```

Message queues:

A message queue is a structure that can be shared between multiple processes. Messages can be placed into the queue and will be removed in the same order in which they were added. Constructing a message queue looks rather like constructing a shared memory segment. The first thing needed is a descriptor, typically the location of a file in the file system. This descriptor can either be used to create the message queue or be used to attach to an existing message queue. Once the queue is configured, processes can place messages into it or remove messages from it. Once the queue is finished, it needs to be deleted.

Figure 2.18 shows code for creating and placing messages into a queue. This code is also responsible for removing the queue after use.

Fig 2.18 Creating and Placing Messages into a Queue

```
ID = Open Message Queue Queue( Descriptor );
Put Message in Queue( ID, Message );
...
Close Message Queue( ID );
Delete Message Queue( Description );
```

Figure 2.19 shows the process for receiving messages for a queue. Using the descriptor for an existing message queue enables two processes to communicate by sending and receiving messages through the queue.

Fig 2.19 Opening a Queue and Receiving Messages

```
ID=Open Message Queue ID(Descriptor);
Message=Remove Message from Queue(ID);
...
Close Message Queue(ID);
```

Pipes:

UNIX uses pipes to pass data from one process to another. For example, the output from the command ls, which lists all the files in a directory, could be piped into the wc command, which counts the number of lines, words, and characters in the input. The combination of the two commands would be a count of the number of files in the directory. Named pipes provide a similar mechanism that can be controlled programmatically. Named pipes are file-like objects that are given a specific name that can be shared between processes. Any process can write into the pipe or read from the pipe. There is no concept of a ¯message‖; the data is treated as a stream of bytes. The method for using a named pipe is much like the method for using a file: The pipe is opened, data is written into it or read from it, and then the pipe is closed.

Fig 2.20: Setting Up and Writing into a Pipe

```
Make Pipe( Descriptor );
ID = Open Pipe( Descriptor );
Write Pipe( ID, Message, sizeof(Message) );
...
Close Pipe( ID );
Delete Pipe( Descriptor );
```

Figure 2.20 shows the steps necessary to set up and write data into a pipe, before closing and deleting the pipe. One process needs to actually make the pipe, and once it has been created, it can be opened and used for either reading or writing. Once the process has completed, the pipe can be closed, and one of the processes using it should also be responsible for deleting it.

Figure 2.21 shows the steps necessary to open an existing pipe and read messages from it. Processes using the same descriptor can open and use the same pipe for communication.

Fig 2.21: Opening an Existing Pipe to Receive Messages

```
ID=Open Pipe( Descriptor );
Read Pipe( ID, buffer, sizeof(buffer) );
...
Close Pipe( ID );
```

UNIT-3 SHARED MEMORY PROGRAMMING WITH OpenMP

PART-A

1. List the effect of cancel construct.

The cancel construct depends on its construct-type clause. If a task encounters a cancel construct with a task group construct-type clause, then the task activates cancellation and continues execution at the end of its task region, which implies completion of that task.

2. Define the term thread private memory.

The temporary view of memory allows the thread to cache variables and thereby to avoid going to memory for every reference to a variable. Each thread also has access to another type of memory that must not be accessed by other threads, called thread private memory.

3. Write short notes on private variable.

A private variable in a task region that eventually generates an inner nested parallel region is permitted to be made shared by implicit tasks in the inner parallel region. A private variable in a task region can be shared by an explicit task region generated during its execution.

4. Define the term shared variable in execution context.

A shared variable has the same address in the execution context of every thread. All threads have access to shared variables.

5. What is the need of OpenMP flush operation?

OpenMP flush operation enforces consistency between the temporary view and memory. The flush operation is applied to a set of variables called the flush-set. The flush operation is applied to a set of variables called the flush set. The flush operation restricts reordering of memory applications that an implementation might otherwise do.

6. List the restrictions to worksharing contructs.

- Each worksharing region must be encountered by all threads in a team or by none at all, unless cancellation has been requested for the innermost enclosing parallel region.
- The sequence of worksharing regions and barrier regions encountered must be the same for every thread in a team.

7. List the restrictions to sections constructs.

- Orphaned section directives are prohibited. That is, the section directives must appear within the sections construct and must not be encountered elsewhere in the sections region.
- The code enclosed in a sections construct must be a structured block.
- Only a single no wait clause can appear on a sections directive.

8. Brief about Simple lock routines.

The type omp_lock_t is a data type capable of representing a simple lock. For the following routines, a simple lock variable must be of omp_lock_type. All simple lock routines reqire an argument that is a pointer to a variable of type omp_lock_t. The simple lock routines are as follows:

- The omp_init_lock routine initializes a simple lock.
- The omp_destroy_lock routine uninitializes a simple lock.
- The omp_set_lock routine waits until a simple lock is available, and then sets it.
- The omp_unset_lock routine unsets a simple lock.
- The omp_test_lock routine tests a simple lock, and sets it if it is available.

9. Brief about pragma.

A compiler directive in C or C++ is called a pragma. The word pragma is short for ‗pragmatic information.‖ A pragma is a way to communicate information to the compiler. The information is nonessential in the sense that the compiler may ignore the information and still produce a correct object program. However, the information provided by the pragma can help the compiler optimize the program.

Like other lines that provide information to the preprocessor, a pragma begins with the # character. A pragma in C or C++ has this syntax:

```
#pragma omp <rest of pragma>
```

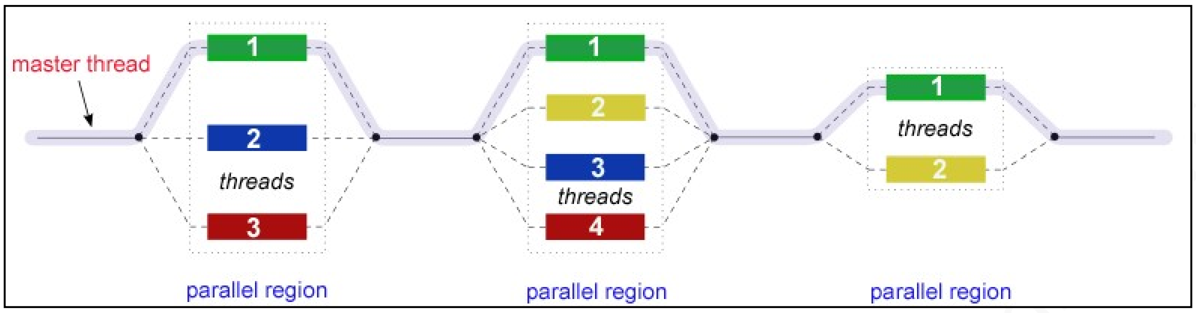
10. What is termed as initial task region?

An initial thread executes sequentially, as if enclosed in an implicit task region, called an initial task region, that is defined by the implicit parallel region surrounding the whole program.

PART-B

OpenMP EXECUTION MODEL

1. Explain briefly about the OpenMP Execution Model. (16)



OpenMP uses the fork-join model of parallel execution. When a thread encounters a parallel construct, the thread creates a team composed of itself and some additional (possibly zero) number of threads. The encountering thread becomes the master of the new team. The other threads of the team are called **slave threads** of the team. All team members execute the code inside the parallel construct. When a thread finishes its work within the parallel construct, it waits at an implicit barrier at the end of the parallel construct. When all team members have arrived at the barrier, the threads can leave the barrier. The master thread continues execution of user code beyond the end of the parallel construct, while the slave threads wait to be summoned to join other teams.

OpenMP parallel regions can be nested inside each other. If nested parallelism is disabled, then the new team created by a thread encountering a parallel construct inside a parallel region consists only of the encountering thread. If nested parallelism is enabled, then the new team may consist of more than one thread.

The OpenMP runtime library maintains a pool of threads that can be used as slave threads in parallel regions. When a thread encounters a parallel construct and needs to create a team of more than one thread, the thread will check the pool and grab idle threads from the pool, making them slave threads of the team. The master thread might get fewer slave threads than it needs if the pool does not contain a sufficient number of idle threads. When the team finishes executing the parallel region, the slave threads return to the pool.

Control of Nested Parallelism

Nested parallelism can be controlled at runtime by setting various environment variables prior to execution of the program.

OMP_NESTED

Nested parallelism can be enabled or disabled by setting the **OMP_NESTED** environment variable or calling **omp_set_nested()**.

The following example has three levels of nested parallel constructs.

Nested Parallelism Example

```
#include <omp.h>
#include <stdio.h>
void report_num_threads(int level)
{
   #pragma omp single
   {
      printf("Level %d: number of threads in the team - %d\n",
              level, omp_get_num_threads());
   }
 }
int main()
{
   omp_set_dynamic(0);
   #pragma omp parallel num_threads(2)
   {
      report_num_threads(1);
      #pragma omp parallel num_threads(2)
      {
         report_num_threads(2);
         #pragma omp parallel num_threads(2)
         {
            report_num_threads(3);
         }
      }
   }
   return(0);
}
```

Compiling and running this program with nested parallelism enabled produces the following (sorted) output:

```
% setenv OMP_NESTED TRUE
% a.out
Level 1: number of threads in the team - 2
Level 2: number of threads in the team - 2
Level 2: number of threads in the team - 2
Level 3: number of threads in the team - 2
Level 3: number of threads in the team - 2
Level 3: number of threads in the team - 2
Level 3: number of threads in the team - 2
```

The following example runs the same program but with nested parallelism disabled:

```
% setenv OMP_NESTED FALSE
% a.out
Level 1: number of threads in the team - 2
Level 2: number of threads in the team - 1
Level 3: number of threads in the team - 1
Level 2: number of threads in the team - 1
Level 3: number of threads in the team - 1
```

OMP_THREAD_LIMIT

The OpenMP runtime library maintains a pool of threads that can be used as slave threads in parallel regions. The setting of the **OMP_THREAD_LIMIT** environment variable controls the number of threads in the pool. By default, the number of threads in the pool is at most 1023.

The thread pool consists of only non-user threads that the runtime library creates. The pool does not include the initial thread or any thread created explicitly by the user's program.

If **OMP_THREAD_LIMIT** is set to 1 (or **SUNW_MP_MAX_POOL_THREADS** is set to 0), then the thread pool will be empty and all parallel regions will be executed by one thread.

The following example shows that a parallel region can get fewer slave threads if the pool does not contain sufficient threads. The code is the same as Example 4-1. The number of threads needed for all the parallel regions to be active at the same time is 8. Therefore, the pool needs to contain at least 7 threads.
If **OMP_THREAD_LIMIT** is set to 6 (or **SUNW_MP_MAX_POOL_THREADS** to 5), then the pool contains at most 5 slave threads. This implies that two of the four innermost parallel regions might not be able to get all the slave threads requested. The following example shows one possible result:

```
% setenv OMP_NESTED TRUE
% OMP_THREAD_LIMIT 6
% a.out
Level 1: number of threads in the team - 2
Level 2: number of threads in the team - 2
Level 2: number of threads in the team - 2
Level 3: number of threads in the team - 2
Level 3: number of threads in the team - 2
Level 3: number of threads in the team - 1
Level 3: number of threads in the team - 1
```

OMP_MAX_ACTIVE_LEVELS

The environment variable **OMP_MAX_ACTIVE_LEVELS** controls the maximum number of nested active parallel regions. A parallel region is active if it is executed by a team consisting of more than one thread. The default maximum number of nested active parallel regions is 4.

Note that setting this environment variable is not sufficient to enable nested parallelism. This environment variable simply controls the the maximum number of nested active parallel regions; it does not enable nested parallelism. To enable nested parallelism, **OMP_NESTED** must be set to **TRUE**, or **omp_set_nested()** must be called with an argument that evaluates to **true**.

The following code will create 4 levels of nested parallel regions. If **OMP_MAX_ACTIVE_LEVELS** is set to 2, then nested parallel regions at nested depth of 3 and 4 are executed single-threaded.

```
#include  <omp.h>
#include <stdio.h>
#define DEPTH 5
void report_num_threads(int level)
{
   #pragma omp single
   {
      printf("Level %d: number of threads in the team - %d\n",
           level, omp_get_num_threads());
   }
}
void nested(int depth)
{
   if (depth == DEPTH)
      return;

   #pragma omp parallel num_threads(2)
   {
      report_num_threads(depth);
      nested(depth+1);
   }
}
int main()
{
   omp_set_dynamic(0);
   omp_set_nested(1);
   nested(1);
   return(0);
}
```

The following example shows the possible results from compiling and running this program with a maximum nesting level of 4. (Actual results would depend on how the OS schedules threads.)

```
% setenv OMP_MAX_ACTIVE_LEVELS 4
% a.out |sort
Level 1: number of threads in the team - 2
Level 2: number of threads in the team - 2
Level 2: number of threads in the team - 2
Level 3: number of threads in the team - 2
Level 3: number of threads in the team - 2
Level 3: number of threads in the team - 2
Level 3: number of threads in the team - 2
Level 4: number of threads in the team - 2
Level 4: number of threads in the team - 2
Level 4: number of threads in the team - 2
Level 4: number of threads in the team - 2
Level 4: number of threads in the team - 2
Level 4: number of threads in the team - 2
Level 4: number of threads in the team - 2
Level 4: number of threads in the team - 2
```

The following example shows a possible result running with the nesting level set at 2:

```
% setenv OMP_MAX_ACTIVE_LEVELS 2
% a.out |sort
Level 1: number of threads in the team - 2
Level 2: number of threads in the team - 2
Level 2: number of threads in the team - 2
Level 3: number of threads in the team - 1
Level 3: number of threads in the team - 1
Level 3: number of threads in the team - 1
Level 3: number of threads in the team - 1
Level 4: number of threads in the team - 1
Level 4: number of threads in the team - 1
Level 4: number of threads in the team - 1
Level 4: number of threads in the team - 1
```

Again, these examples only show some **possible** results. Actual results would depend on how the OS schedules threads.

OpenMP DIRECTIVES

2. Discuss briefly about OpenMP Directives. (16)

Directives

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

In fixed form source files, the sentinel !omp, comp,or *$ompintroduce a directive and must start in column 1. An initial directive line must contain a space or zero in column 6, and continuation lines must have a character other than a space or zero in column 6. In free form source files, the sentinel !$omp introduces a directive and can appear in any column if preceded only by white space.

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

```
!$omp parallel[clause[ [,]clause] ...]
                structured-block
!$omp end parallel
clause:
if(scalar-expression)
num_threads(scalar-integer-expression
default(private|firstprivate|
    shared| none)
private(list)
firstprivate(list)
shared(list)
copyin(list)
reduction({operator|intrinsic_procedure_name}:list)
```

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

```
    !$omp do[clause[[,] clause] ... ]
    do-loops
    [!$omp end do [nowait]]
    clause:
    private(list)
    firstprivate(list)
    lastprivate(list)
    reduction({operator|intrinsic_procedure_name}:list)
    schedule(kind[, chunk_size])
    collapse(n)
    ordered
```

The sections construct contains a set of structured blocks to be distributed among and executed by encountering team of threads.

```
    !$omp sections[clause[[,] clause] ...]
    [!$omp section]
    structured-block
    [!$omp section
    structured-block ]
    ...
    !$omp end sections [nowait]
```

clause:

```
    private(list)
    firstprivate(list)
    lastprivate(list)
    reduction({operator|intrinsic_procedure_name}:list)
```

The single construct specifies that the associated structured block is executed by only one of the threads in the team (not necessarily the master thread), in the context of its implicit task.

```
    !$omp single [clause[[,] clause] ...]
    structured-block
    !$omp end single [end_clause[[,] end_clause] ...]
```

clause:

```
    private(list)
    firstprivate(list)
    end_clause:
    copyprivate(list)
```

```
nowait
```

The workshareconstruct divides the execution of the enclosed structured block into separate units of work.

```
!$omp workshare structured-block
!$omp end workshare [nowait]
```

The combined parallel worksharing constructs are a shortcut for specifying a parallel construct containing one work-sharing construct and no other statements.

```
!$omp parallel do [clause[[,] clause] ...]
do-loop
[!$omp end parallel do]
!$omp parallel sections [clause[ [,]clause] ...]
[!$omp section]
structured-block
[!$omp section
structured-block ]
...
!$omp end parallel sections
clause:
any clause from parallelor sections
statement: one of the following forms:
x = x operator expr
x = expr operator x
x = intrinsic_procedure_name (x, expr_list)
x = intrinsic_procedure_name (expr_list, x)
```

The flush construct executes the OpenMP flush operation, which makes a thread‘s temporary view of memory consistent with memory, and enforces an order on the memory operations of the variables.

```
!$omp flush [(list)]
```

The ordered construct specifies a structured block in a loop region that will be executed in the order of the loop iterations. This sequentializes and orders the code within an ordered region while allowing code outside the region to run in parallel.

```
#pragma !$omp ordered
structured-block
#pragma !$omp ordered
```

The threadprivatedirective specifies that variables are replicated,

with each thread having its own copy.

```
!$omp threadprivate(list)
```

WORK-SHARING CONSTRUCTS

3. Explain in detail about various OpenMP work-sharing constructs. (16)

Work-Sharing Constructs

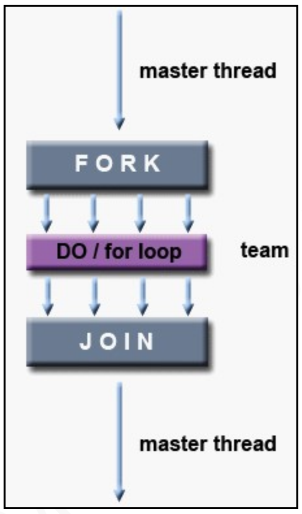
- A work-sharing construct divides the execution of the enclosed code region among the members of the team that encounter it.
- Work-sharing constructs do not launch new threads
- There is no implied barrier upon entry to a work-sharing construct, however there is an implied barrier at the end of a work sharing construct.

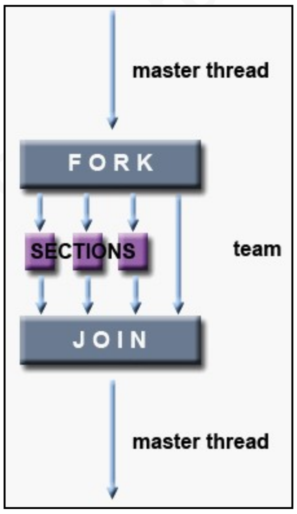
Types of Work-Sharing Constructs:

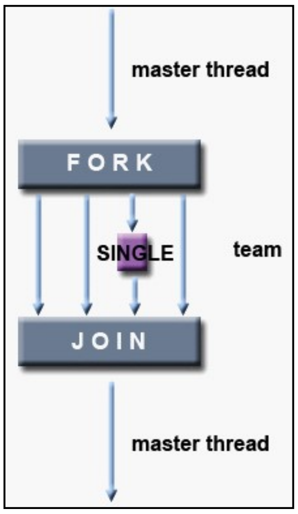
DO / for - shares iterations of a loop across the team. Represents a type of "data parallelism".

SECTIONS - breaks work into separate, discrete sections. Each section is executed by a thread. Can be used to implement a type of "functional parallelism".

SINGLE - serializes a section of code







Restrictions:

- A work-sharing construct must be enclosed dynamically within a parallel region in order for the directive to execute in parallel.
- Work-sharing constructs must be encountered by all members of a team or none at all
- Successive work-sharing constructs must be encountered in the same order by all members of a team

DO / for Directive

Purpose:

- The DO / for directive specifies that the iterations of the loop immediately following it must be executed in parallel by the team. This assumes a parallel region has already been initiated, otherwise it executes in serial on a single processor.

```
#include <omp.h>
 #define N 1000
 #define CHUNKSIZE 100
 main(int argc, char *argv[]) {
 int i, chunk;
 float a[N], b[N], c[N];
 /* Some initializations */
 for (i=0; i < N; i++)
   a[i] = b[i] = i * 1.0;
 chunk = CHUNKSIZE;
 #pragma omp parallel shared(a,b,c,chunk) private(i)
   {
   #pragma omp for schedule(dynamic,chunk) nowait
   for (i=0; i < N; i++)
     c[i] = a[i] + b[i];
   }  /* end of parallel region */
 }
```

SECTIONS Directive

Purpose:

- The SECTIONS directive is a non-iterative work-sharing construct. It specifies that the enclosed section(s) of code are to be divided among the threads in the team.
- Independent SECTION directives are nested within a SECTIONS directive. Each SECTION is executed once by a thread in the team. Different sections may be executed by different threads. It is possible for a thread to execute more than one section if it is quick enough and the implementation permits such.

```
#include <omp.h>
 #define N 1000
 main(int argc, char *argv[]) {
 int i;
 float a[N], b[N], c[N], d[N];
 /* Some initializations */
 for (i=0; i < N; i++) {
   a[i] = i * 1.5;
   b[i] = i + 22.35;
   }
 #pragma omp parallel shared(a,b,c,d) private(i)
   {
   #pragma omp sections nowait
     {
     #pragma omp section
     for (i=0; i < N; i++)
       c[i] = a[i] + b[i];
     #pragma omp section
     for (i=0; i < N; i++)
       d[i] = a[i] * b[i];
     }  /* end of sections */
   }  /* end of parallel region */
 }
```

SINGLE Directive

Purpose:

- The SINGLE directive specifies that the enclosed code is to be executed by only one thread in the team.
- May be useful when dealing with sections of code that are not thread safe (such as I/O)

LIBRARY FUNCTIONS

4. Explain about the OpenMP Library functions. (16)

- The OpenMP API includes an ever-growing number of run-time library routines.
- These routines are used for a variety of purposes as shown in the table below:

Routine	Purpose
OMP_SET_NUM_THREADS	Sets the number of threads that will be used in the next parallel region
OMP_GET_NUM_THREADS	Returns the number of threads that are currently in the team executing the parallel region from which it is called
OMP_GET_MAX_THREADS	Returns the maximum value that can be returned by a call to the OMP_GET_NUM_THREADS

	function
OMP_GET_THREAD_NUM	Returns the thread number of the thread, within the team, making this call.
OMP_GET_THREAD_LIMIT	Returns the maximum number of OpenMP threads available to a program
OMP_GET_NUM_PROCS	Returns the number of processors that are available to the program
OMP_IN_PARALLEL	Used to determine if the section of code which is executing is parallel or not
OMP_SET_DYNAMIC	Enables or disables dynamic adjustment (by the run time system) of the number of threads available for execution of parallel regions
OMP_GET_DYNAMIC	Used to determine if dynamic thread adjustment is enabled or not
OMP_SET_NESTED	Used to enable or disable nested parallelism
OMP_GET_NESTED	Used to determine if nested parallelism is enabled or not
OMP_SET_SCHEDULE	Sets the loop scheduling policy when "runtime" is used as the schedule kind in the OpenMP directive
OMP_GET_SCHEDULE	Returns the loop scheduling policy when "runtime" is used as

	the schedule kind in the OpenMP directive
OMP_SET_MAX_ACTIVE_LEVELS	Sets the maximum number of nested parallel regions
OMP_GET_MAX_ACTIVE_LEVELS	Returns the maximum number of nested parallel regions
OMP_GET_LEVEL	Returns the current level of nested parallel regions
OMP_GET_ANCESTOR_THREAD_NUM	Returns, for a given nested level of the current thread, the thread number of ancestor thread
OMP_GET_TEAM_SIZE	Returns, for a given nested level of the current thread, the size of the thread team
OMP_GET_ACTIVE_LEVEL	Returns the number of nested, active parallel regions enclosing the task that contains the call
OMP_IN_FINAL	Returns true if the routine is executed in the final task region; otherwise it returns false
OMP_INIT_LOCK	Initializes a lock associated with the lock variable
OMP_DESTROY_LOCK	Disassociates the given lock variable from any locks
OMP_SET_LOCK	Acquires ownership of a lock
OMP_UNSET_LOCK	Releases a lock
OMP_TEST_LOCK	Attempts to set a lock, but does not block if the lock is

	unavailable
OMP_INIT_NEST_LOCK	Initializes a nested lock associated with the lock variable
OMP_DESTROY_NEST_LOCK	Disassociates the given nested lock variable from any locks
OMP_SET_NEST_LOCK	Acquires ownership of a nested lock
OMP_UNSET_NEST_LOCK	Releases a nested lock
OMP_TEST_NEST_LOCK	Attempts to set a nested lock, but does not block if the lock is unavailable
OMP_GET_WTIME	Provides a portable wall clock timing routine
OMP_GET_WTICK	Returns a double-precision floating point value equal to the number of seconds between successive clock ticks

---

HANDLING LOOPS

5. Explain briefly about how to handle loops in OpenMP (16)

Virtually all useful programs have some sort of loop in the code, whether it is a for, do, or while loop. This is especially true for all programs which take a significant amount of time to execute. Much of the time, different iterations of these loops have nothing to do with each other, therefore making these loops a prime target for parallelization. OpenMP effectively exploits these common program characteristics, so it is extremely easy to allow an OpenMP program to use multiple processors simply by adding a few lines of compiler directives into your source code.

Parallel for loops

This tutorial will be exploring just some of the ways in which you can use OpenMP to allow your loops in your program to run on multiple processors. For the sake of argument, suppose you‘re writing a ray tracing program. Without going too much into the details of how ray tracing works, it simply goes through each pixel of the screen, and using lighting, texture, and geometry information, the color of that pixel is determined. The program goes on to the next pixel and repeats the process. The important thing to note here is that the calculation for each pixel is completely separate from the calculation of any other pixel, therefore making this program highly suitable for OpenMP. Consider the following pseudo-code:

```
for(int x=0; x < width; x++)
{
  for(int y=0; y < height; y++)
  {
            finalImage[x][y] = RenderPixel(x,y, &sceneData);
  }
}
```

Please take a quick look at the code above. This piece of code simply goes through each pixel of the screen, and calls a function, RenderPixel, to determine the final color of that pixel. Note that the results are simply stored in an array. Simply put, the entire scene that is being rendered is stored in a variable, sceneData, whose address is passed to the RenderPixel function. Because each pixel is independent of all other pixels, and because RenderPixel is expected to take a noticeable amount of time, this small snippet of code is a prime candidate for parallelization with OpenMP. Consider the following modified pseudo-code:

```
#pragma omp parallel for
for(int x=0; x < width; x++)
{
  for(int y=0; y < height; y++)
  {
            finalImage[x][y] = RenderPixel(x,y, &sceneData);
  }
}
```

The only change to the code is the line directly above the outer for loop. This compiler directive tells the compiler to auto-parallelize the for loop with OpenMP. If a user is using a quad-core processor, the performance of your program can be expected to be 300% increased with the addition of just one line of code, which is amazing. In practice, true linear or super linear speedups are rare, while near linear speedups are very common.

There are a few important things you need to keep in mind when parallelizing for loops or any other sections of code with OpenMP. For example, take a look at variable y in the pseudo code above. Because the variable is effectively being declared inside the parallelized region, each processor will have a unique and private value for y. However, take the following buggy code example below:

```
int x,y;
#pragma omp parallel for
for(x=0; x < width; x++)
```

```
{
  for(y=0; y < height; y++)
  {
          finalImage[x][y] = RenderPixel(x,y, &sceneData);
  }
}
```

The above code has a serious bug in it. The only thing that changed is the fact that now, variables x and y are declared outside the parallelized region. When we use the compiler directive to declare the outer for loop to be parallelized with OpenMP, the compiler already knows by common sense that the variable x is going to have different values for different threads. However, the default scope for the other variables, y, finalImage, and sceneData, are all shared by default, meaning that these values will be the same for all threads. All threads have access to read and write to these shared variables. The code above is buggy because variable y should be different for each thread. Declaring y inside of the parallelized region is one way to guarantee that a variable will be private to each thread, but there is another way to accomplish this.

```
int x,y;
#pragma omp parallel for private(y)
for(x=0; x < width; x++)
{
  for(y=0; y < height; y++)
  {
          finalImage[x][y] = RenderPixel(x,y, &sceneData);
  }
}
```

Instead of declaring variable y inside the parallel region, we can declare it outside the parallel region and explicitly declare it a private variable during the OpenMP compiler directive. This effectively makes each thread have an independent variable called y. Each thread will only have access to it‘s own copy of this variable.

A word about shared variables

Forgetting to declare a variable as private is one of the most common bugs associated with writing OpenMP applications. However, if you want the highest performance out of your program, it is best to use private variables only when you have to. In the pseudo code for this tutorial, finalImage and sceneData are shared variables for good reasons. Even though each thread is writing to the finalImage array, these writes will not conflict with each other as long as x and y are private variables. SceneData is also a shared variable, because the threads will only read from this data structure, which will never change. Therefore, there will be no race conditions associated with this variable.

Wrapping up

There are many more interesting things that can be done with parallelizing loops, and this tutorial is just the tip of the iceberg. The main focus of this article was to let you know how you can quickly and easily modify your program to use multiple processors with OpenMP. Aside from using the compiler directive to specify which loop you want to be parallel, it is also extremely important to know which variables should be private, and which should be shared. Failure to properly classify a variable will result in terrible bugs and race conditions which are very hard to debug. If your program is written correctly, it should work great on a computer with one processor, and it should work even better on a serious computer with 24 or more processors. Using OpenMP to parallelize loops for you can be extremely scalable.

UNIT-4 DISTRIBUTED MEMORY PROGRAMMING WITH MPI

PART-A

1. What is the goal of MPI?

The goal of the Message Passing Interface is to establish a portable, efficient, and flexible standard for message passing that will be widely used for writing message passing programs. As such, MPI is the first standardized, vendor independent, message passing library. The advantages of developing message passing software using MPI closely match the design goals of portability, efficiency, and flexibility.

2. Write short notes on collective communication.

Collective communication is defined as communication that involves a group of processes. The functions of this type provided by MPI are the following:

- Barrier synchronization across all group members
- Broadcast from one member to all members of a group
- Gather data from all group members to one member
- Scatter data from one member to all members of a group
- A variation on Gather where all members of the group receive the result
- Scatter/Gather data from all members to all members of a group
- Global reduction operations such as sum, max, min, or user-defined functions, where the result is returned to all group members and a variation where the result is returned to only one member
- A combined reduction and scatter operation
- Scan across all members of a group

3. List the functions of group assessors.

- MPI_GROUP_SIZE(group, size)
- MPI_GROUP_RANK(group,rank)
- MPI_GROUP_TRANSLATE_RANKS(group1,n,ranks1,group2,ranks2)
- MPI_GROUP_COMPARE(group1,group2,result)

4. Define the term broadcast in collective communication.

A collective communication in which data belonging to a single process is sent to all of the processes in the communicator is called as broadcast.

5. Write short notes on communicator in MPI.

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.

6. What is the purpose of wrapper script?

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.

7. Define the term linear speedup.

The ideal value for S(n, p) is p. If S(n, p)= p, then our parallel program with comm_sz = p processes is running p times faster than the serial program. In practice, this speedup, sometimes called linear speedup, is rarely achieved. Our matrix-vector multiplication program got the speedups.

8. Brief about strongly and weakly scalable.

Programs that can maintain a constant efficiency without increasing the problem size are sometimes said to be strongly scalable. Programs that can maintain a constant efficiency if the problem size increases at the same rate as the number of processes are sometimes said to be weakly scalable.

9. What is the use of MPI derived datatype?

In MPI, a derived datatype 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.

10. What are the different datatype constructors?

- MPI_TYPE_CONTIGUOUS
- MPI_TYPE_VECTOR
- MPI_TYPE_CREATE_HVECTOR
- MPI_TYPE_INDEXED

- MPI_TYPE_CREATE_HINDEXED
- MPI_TYPE_HINDEXED_BLOCK
- MPI_TYPE_INDXED_BLOCK
- MPI_TYPE_STRUCT

PART-B

MPI PROGRAM EXECUTION

1. Explain in detail about the execution of MPI programs. (16)

In parallel programming, it‘s common (one might say standard) for the processes to be identified by nonnegative integer ranks. So if there are p processes, the processes will have ranks 0,1,2, : : : , p 1. For our parallel ¯hello, world,‖ let‘s make process 0 the designated process, and the other processes will send it messages. See Program 4.1.

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

const int MAX_STRING = 100;

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

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

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

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

Program 4.1: MPI program that prints greetings from the processes

Compilation and Execution:

The details of compiling and running the program depend on your system, so you may need to check with a local expert. However, recall that when we need to be explicit, we‗ll assume that we‗re using a text editor to write the program source, and the command line to compile and run. Many systems use a command called mpicc for compilation.

```
$ mpicc  g  Wall  o mpi hello mpi hello.c
```

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.

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
```

and to run the program with four processes, we‗d type

```
$ mpiexec  n 4 ./mpi-hello
```

With one process the program‗s output would be

```
Greetings from process 0 of 1!
```

and with four processes the program‗s output would be

```
Greetings from process 0 of 4!
Greetings from process 1 of 4!
Greetings from process 2 of 4!
Greetings from process 3 of 4!
```

How do we get from invoking mpiexec to one or more lines of greetings? The mpiexec command tells the system to start <number of processes> instances of our <mpi hello> program. It may also tell the system which core should run each instance of the program. After the processes are running, the MPI implementation takes care of making sure that the processes can communicate with each other.

Let‗s take a closer look at the program. 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; it contains all the definitions and declarations needed for compiling an MPI program. 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, so there's no question about what is defined by MPI and what's defined by the user program.

MPI_Init and MPI_Finalize:

In Line 12 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 */);
```

The arguments, argc_p and argv_p, are pointers to the arguments to main, argc, and argv. However, when our program doesn't use these arguments, we can just pass NULL for both. Like most MPI functions, MPI_Init returns an int error code, and in most cases we'll ignore these error codes.

In Line 30 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. Thus, a typical MPI program has the following basic outline:

```
. . .
#include <mpi.h>
. . .
int main(int argc, char* argv[]) {
   . . .
   /* No MPI calls before this */
   MPI_Init(&argc, &argv);
   . . .
   MPI_Finalize();
   /* No MPI calls after this */
   . . .
   return 0;
}
```

Communicators-MPI_Comm_size and MPI_Comm_rank:

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 13 and 14 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 */);
```

For both functions, the first argument is a communicator and has the special type defined by MPI for communicators, MPI_Comm. MPI_Comm size returns in its second argument the number of processes in the communicator, and MPI_Comm rank returns in its second argument the calling process' rank in the communicator. We'll often use the variable comm_sz for the number of processes in MPI_COMM_WORLD, and the variable my_rank for the process rank.

MPI SEND AND RECEIVE

2. Explain in detail about MPI_Send and MPI_Receive. (16)

MPI_Send

The sends executed by processes 1,2, : : : , comm, sz-1 are fairly complex, so let's take a closer look at them. Each of the sends is carried out by a call to MPI Send, whose syntax is

```
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 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. In our program, this is just the string containing the message, greeting. (Remember that in C an array, such as a string, is a pointer.) 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.

Since C types (int, char, and so on.) can‘t be passed as arguments to functions, MPI defines a special type, MPI_Datatype, that is used for the msg_type argument. MPI also defines a number of constant values for this type. The ones we‘ll use (and a few others) are listed in Table.

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	

Notice that the size of the string greeting is not the same as the size of the message specified by the arguments msg_size and msg_type. For example, when we run the program with four processes, the length of each of the messages is 31 characters, while we‘ve allocated storage for 100 characters in greetings. Of course, the size of the message sent should be less than or equal to the amount of storage in the buffer—in our case the string greeting.

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. For example, suppose process 1 is sending floats to process 0. Some of the floats should be printed, while others should be used in a computation. Then the first four arguments to MPI_Send provide no information regarding which floats should be printed and which should be used in a computation. So process 1 can use, say, a tag of 0 for the messages that should be printed and a tag of 1 for the messages that should be used in a computation.

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; recall that a communicator is a collection of processes that can send messages to each other. Conversely, a message sent by a process using one communicator cannot be received by a process that‘s using a different communicator. Since MPI provides functions for creating new communicators, this feature can be used in complex programs to insure that messages aren‘t ¯accidentally received‖ in the wrong place.

An example will clarify this. Suppose we‘re studying global climate change, and we‘ve been lucky enough to find two libraries of functions, one for modeling the Earth‘s atmosphere and one for modeling the Earth‘s oceans. Of course, both libraries use MPI. These models were built independently, so they don‘t communicate with each other, but they do communicate internally. It‘s our job to write the interface code. One problem we need to solve is to insure that the messages sent by one library won‘t be accidentally received by the other. We might be able to work out some scheme with tags: the atmosphere library gets tags 0,1, : : : , n-1 and the ocean library gets tags n, n + 1, : : : , n + m. Then each library can use the given range to figure out which tag it should use for which message. However, a much simpler solution is provided by communicators: we simply pass one communicator to the atmosphere library functions and a different communicator to the ocean library functions.

MPI_Recv

The first six arguments to MPI_Recv correspond to the first six arguments of MPI_Send:

```
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 */);
```

Thus, the first three arguments specify the memory available for receiving the message: msg_buf_p points to the block of memory, buf_size determines the number of objects that can be stored in the block, and buf_type indicates the type of the objects. The next three arguments identify the message. The source argument specifies the process from which the message should be received. The tag argument should match the tag argument of the message being sent, and the communicator argument must match the communicator used by the sending process. We‘ll talk about the status_p argument shortly. In many cases it won‘t be used by the calling function, and, as in our ¯greetings‖ program, the special MPI constant MPI_STATUS_IGNORE can be passed.

COLLECTIVE COMMUNICATION

3. Explain in detail about the following collective communication mechanisms.(16)

a) Tree-structured communication

b) Broadcast

a) Tree-structures communication:

In the following figure 4.2, initially students or processes 1, 3, 5, and 7 send their values to processes 0, 2, 4, and 6, respectively. Then processes 0, 2, 4, and 6 add the received values to their original values, and the process is repeated twice:

1. a. Processes 2 and 6 send their new values to processes 0 and 4, respectively.
   b. Processes 0 and 4 add the received values into their new values.
2. a. Process 4 sends its newest value to process 0.
   b. Process 0 adds the received value to its newest value.

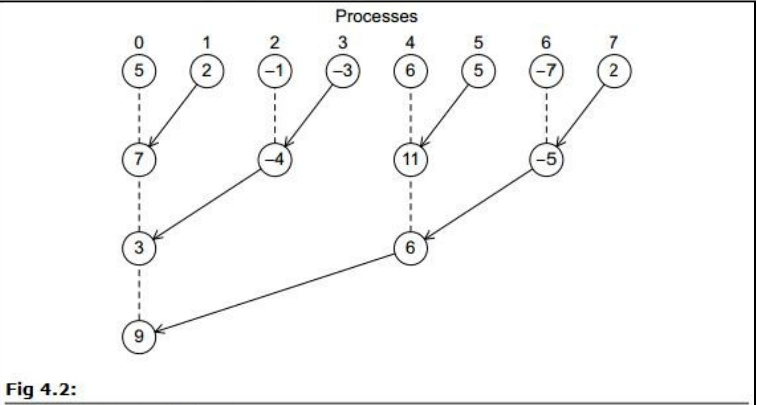


Fig 4.2:
A tree-structured global sum

This solution may not seem ideal, since half the processes (1, 3, 5, and 7) are doing the same amount of work that they did in the original scheme. However, if you think about it, the original scheme required comm-sz - 1 = seven receives and seven adds by process 0, while the new scheme only requires three, and all the other processes do no more than two receives and adds. Furthermore, the new scheme has a property by which a lot of the work is done concurrently by different processes. For example, in the first phase, the receives and adds by processes 0, 2, 4, and 6 can all take place simultaneously. So, if the processes start at roughly the same time, the total time required to compute the global sum will be the time required by process 0, that is, three receives and three additions. We‘ve thus reduced the overall time by more than 50%. Furthermore, if we use more processes, we can do even better. For example, if comm_sz = 1024, then the original scheme requires process 0 to do 1023 receives and additions, while it can be shown that the new scheme requires process 0 to do only 10 receives and additions. This improves the original scheme by more than a factor of 100!

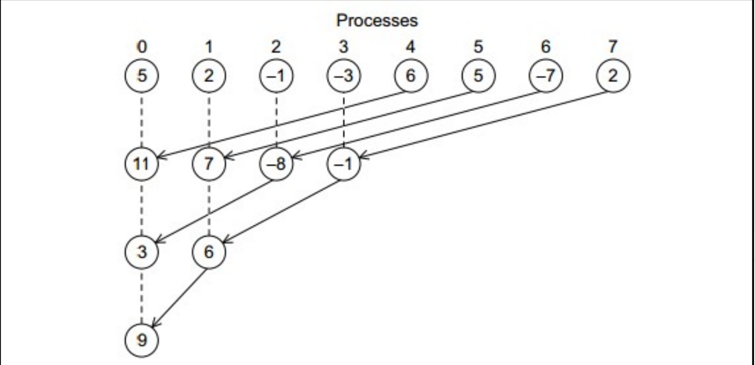


Fig 4.3:
An alternative tree-structured global sum

You may be thinking to yourself, this is all well and good, but coding this treestructured global sum looks like it would take a quite a bit of work, and you‘d be right. See Programming Assignment 3.3. In fact, the problem may be even harder. For example, it‘s perfectly feasible to construct a tree-structured global sum that uses different ―process-pairings.‖ For example, we might pair 0 and 4, 1 and 5, 2 and 6, and 3 and 7 in the first phase. Then we could pair 0 and 2, and 1 and 3 in the second, and 0 and 1 in the final. See Figure 4.3. Of course, there are many other possibilities. How can we decide which is the best? Do we need to code each alternative and evaluate its performance? If we do, is it possible that one method works best for ―small‖ trees, while another works best for ―large‖ trees? Even worse, one approach might work best on system A, while another might work best on system B.

b) Broadcast:

A collective communication in which data belonging to a single process is sent to all of the processes in the communicator is called a broadcast, and you‘ve probably guessed that MPI provides 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     */);
```

The process with rank source_proc sends the contents of the memory referenced by data_p to all the processes in the communicator comm.

```
void Get_input(
      int      my_rank  /* in  */,
      int      comm_sz  /* in  */,
      double*  a_p      /* out */,
      double*  b_p      /* out */,
      int*     n_p      /* out */) {

   if (my_rank == 0) {
      printf("Enter a, b, and n\n");
      scanf("%lf %lf %d", a_p, b_p, n_p);
   }
   MPI_Bcast(a_p, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
   MPI_Bcast(b_p, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
   MPI_Bcast(n_p, 1, MPI_INT, 0, MPI_COMM_WORLD);
}  /* Get_input */
```

Figure 4.4 shows how to modify the Get_input function so that it uses MPI_Bcast instead of MPI_Send and MPI_Recv.

Recall that in serial programs, an in/out argument is one whose value is both used and changed by the function. For MPI_Bcast, however, the data_p argument is an input argument on the process with rank source_proc and an output argument on the other processes. Thus, when an argument to a collective communication is labeled in/out, it‘s possible that it‘s an input argument on some processes and an output argument on other processes.

4. Discuss elaborately about the differences of collective and point-to-point communications. (16)

1. All the processes in the communicator must call the same collective function. For example, a program that attempts to match a call to MPI_Reduce on one process with a call to MPI_Recv on another process is erroneous, and, in all likelihood, the program will hang or crash.

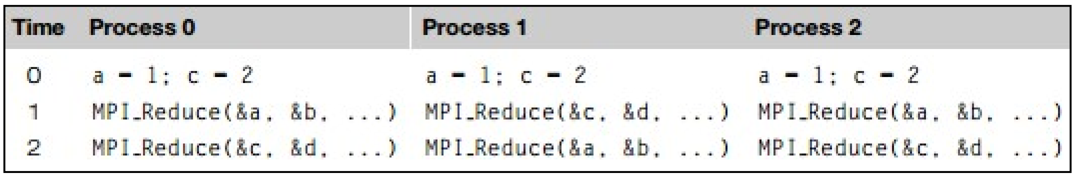
2. The arguments passed by each process to an MPI collective communication must be ¯compatible.‖ For example, if one process passes in 0 as the dest_process and another passes in 1, then the outcome of a call to MPI_Reduce is erroneous, and, once again, the program is likely to hang or crash.

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.

4. Point-to-point communications are matched on the basis of tags and communicators. Collective communications don‘t use tags, so they‘re matched solely on the basis of the communicator and the order in which they‘re called. As an example, consider the calls to MPI Reduce shown in Figure 4.5. Suppose that each process calls MPI_Reduce with operator MPI_SUM, and destination process 0. At first glance, it might seem that after the two calls to MPI_Reduce, the value of b will be three, and the value of d will be six. However, the names of the memory location are irrelevant to the matching, of the calls to MPI_Reduce. The order of the calls will determine the matching, so the value stored in b will be 1 + 2 + 1 = 4, and the value stored in d will be 2 + 1 + 2 = 5.

A final caveat: it might be tempting to call MPI_Reduce using the same buffer for both input and output. For example, if we wanted to form the global sum of x on each process and store the result in x on process 0, we might try calling

MPI_Reduce(&x, &x, 1, MPI_DOUBLE, MPI_SUM, 0, comm);



Time	Process 0	Process 1	Process 2
0	a = 1; c = 2	a = 1; c = 2	a = 1; c = 2
1	MPI_Reduce(&a, &b, ...)	MPI_Reduce(&c, &d, ...)	MPI_Reduce(&a, &b, ...)
2	MPI_Reduce(&c, &d, ...)	MPI_Reduce(&a, &b, ...)	MPI_Reduce(&c, &d, ...)

Fig 4.5: Multiple Calls to MPI_Reduce

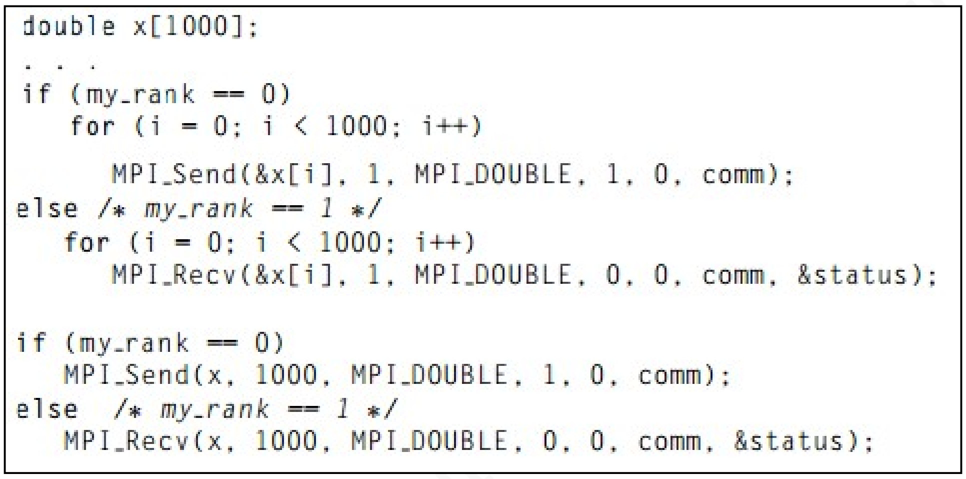
However, this call is illegal in MPI, so its result will be unpredictable: it might produce an incorrect result, it might cause the program to crash, it might even produce a correct result. It‘s illegal because it involves aliasing of an output argument. Two arguments are aliased if they refer to the same block of memory, and MPI prohibits aliasing of arguments if one of them is an output or input/output argument. This is because the MPI Forum wanted to make the Fortran and C versions of MPI as similar as possible, and Fortran prohibits aliasing. In some instances, MPI provides an alternative construction that effectively avoids this restriction.

If the communicator comm contains comm_sz processes, then MPI Scatter divides the data referenced by send buf p into comm_sz pieces–the first piece goes to process 0, the second to process 1, the third to process 2, and so on. For example, suppose we‘re using a block distribution and process 0 has read in all of an n-component vector into send buf_p. Then, process 0 will get the first local n D n=comm_sz components, process 1 will get the next local n components, and so on. Each process should pass its local vector as the recv_buf_p argument and the recv count argument should be local n. Both send type and recv type should be MPI DOUBLE and src_proc should be 0. Perhaps surprisingly, send count should also be local n–send count is the amount of data going to each process; it‘s not the amount of data in the memory referred to by send buf_p. If we use a block distribution and MPI Scatter, we can read in a vector using the function Read vector. One point to note here is that MPI Scatter sends the first block of send count objects to process 0, the next block of send count objects to process 1, and so on, so this approach to reading and distributing the input vectors will only be suitable if we‘re using a block distribution and n, the number of components in the vectors, is evenly divisible by comm_sz. We‘ll discuss a partial solution to dealing with a cyclic or block-cyclic distribution. We‘ll look at dealing with the case in which n is not evenly divisible by comm_sz. The data stored in the memory referred to by send buf_p on process 0 is stored in the first block in recv_buf_p, the data stored in the memory referred to by send_buf_p on process 1 is stored in the second block referred to by recv_buf_p, and so on.

MPI DERIVED DATATYPES

5. Explain in detail about various MPI Derived Datatypes. (16)

In virtually all distributed-memory systems, communication can be much more expensive than local computation. For example, sending a double from one node to another will take far longer than adding two doubles stored in the local memory of a node. Furthermore, the cost of sending a fixed amount of data in multiple messages is usually much greater than the cost of sending a single message with the same amount of data. For example, we would expect the following pair of for loops to be much slower than the single send/receive pair:



```
double x[1000];
. . .
if (my_rank == 0)
   for (i = 0; i < 1000; i++)
      MPI_Send(&x[i], 1, MPI_DOUBLE, 1, 0, comm);
else /* my_rank == 1 */
   for (i = 0; i < 1000; i++)
      MPI_Recv(&x[i], 1, MPI_DOUBLE, 0, 0, comm, &status);

if (my_rank == 0)
   MPI_Send(x, 1000, MPI_DOUBLE, 1, 0, comm);
else  /* my_rank == 1 */
   MPI_Recv(x, 1000, MPI_DOUBLE, 0, 0, comm, &status);
```

In fact, on one of our systems, the code with the loops of sends and receives takes nearly 50 times longer. On another system, the code with the loops takes more than 100 times longer. Thus, if we can reduce the total number of messages we send, we‘re likely to improve the performance of our programs.

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. In this section we‘ll discuss one method for building derived datatypes. In the exercises, we‘ll take a look at some other methods for building derived datatypes and MPI_Pack/Unpack

In MPI, a derived datatype 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. As an example, in our trapezoidal rule program we needed to call MPI_Bcast three times: once for the left endpoint a, once for the right endpoint b, and once for the number of trapezoids n. As an alternative, we could build a single derived datatype that consists of two doubles and one int. If we do this, we'll only need one call to MPI_Bcast. On process 0, a, b, and n will be sent with the one call, while on the other processes, the values will be received with the call.

Formally, a derived datatype consists of a sequence of basic MPI datatypes together with a displacement for each of the datatypes. In our trapezoidal rule example, suppose that on process 0 the variables a, b, and n are stored in memory locations with the following addresses:

Variable	Address
a	24
b	40
n	48

Then the following derived datatype could represent these data items:

```
{(MPI_DOUBLE,0),(MPI_DOUBLE,16),(MPI_INT,24)}.
```

The first element of each pair corresponds to the type of the data, and the second element of each pair is the displacement of the data element from the beginning of the type. We've assumed that the type begins with a, so it has displacement 0, and the other elements have displacements measured, in bytes, from a: b is 40 - 24 = 16 bytes beyond the start of a, and n is 48 - 24 = 24 bytes beyond the start of a.

We can use MPI_Type_create_struct to build a derived datatype that consists of individual elements that have different basic types:

```
int MPI_Type_create_struct(
      int            count                    /* in   */,
      int            array_of_blocklengths[]  /* in   */,
      MPI_Aint       array_of_displacements[] /* in   */,
      MPI_Datatype   array_of_types[]         /* in   */,
      MPI_Datatype*  new_type_p               /* out  */);
```

The argument count is the number of elements in the datatype, so for our example, it should be three. Each of the array arguments should have count elements. The first array, array_of_block_lengths, allows for the possibility that the individual data items might be arrays or subarrays. If, for example, the first element were an array containing five elements, we would have

```
array_of_blocklengths[0] = 5;
```

However, in our case, none of the elements is an array, so we can simply define

```
int array_of_blocklengths[3] = {1, 1, 1};
```

The third argument to MPI_Type_create_struct, array_of_displacements, specifies the displacements, in bytes, from the start of the message. So we want

```
array_of_displacements[] = {0, 16, 24};
```

To find these values, we can use the function MPI Get address:

```
int MPI_Get_address(
      void*      location_p  /* in  */,
      MPI_Aint*  address_p   /* out */);
```

It returns the address of the memory location referenced by location_p. The special type MPI_Aint is an integer type that is big enough to store an address on the system. Thus, in order to get the values in array_of_displacements, we can use the following code:

```
MPI_Aint a_addr, b_addr, n_addr;

MPI_Get_address(&a, &a_addr);
array_of_displacements[0] = 0;
MPI_Get_address(&b, &b_addr);
array_of_displacements[1] = b_addr - a_addr;
MPI_Get_address(&n, &n_addr);
array_of_displacements[2] = n_addr - a_addr;
```

The array_of_datatypes should store the MPI datatypes of the elements, so we can just define

```
MPI_Datatype array_of_types[3] = {MPI_DOUBLE, MPI_DOUBLE, MPI_INT};
```

With these initializations, we can build the new datatype with the call

```
MPI_Datatype input_mpi_t;
. . .
MPI_Type_create_struct(3, array_of_blocklengths,
      array_of_displacements, array_of_types,
      &input_mpi_t);
```

Before we can use input_mpi_t in a communication function, we must first commit it with a call to

```
int MPI_Type_commit(MPI_Datatype*  new_mpi_t_p  /* in/out */);
```

This allows the MPI implementation to optimize its internal representation of the datatype for use in communication functions.

Now, in order to use new_mpi_t, we make the following call to MPI_Bcast on each process:

```
MPI_Bcast(&a, 1, input_mpi_t, 0, comm);
```

So we can use input_mpi_t just as we would use one of the basic MPI datatypes.

In constructing the new datatype, it‘s likely that the MPI implementation had to allocate additional storage internally. Therefore, when we‘re through using the new type, we can free any additional storage used with a call to

```
int MPI_Type_free(MPI_Datatype*  old_mpi_t_p  /* in/out */);
```

UNIT-5 PARALLEL PROGRAM DEVELOPMENT

PART-A

1. What are the input and output parameters of n-body problem?

The input to the problem is the mass, position, and velocity of each particle at the start of the simulation, and the output is typically the position and velocity of each particle at a sequence of user-specified times, or simply the position and velocity of each particle at the end of a user-specified time period.

2. What is the function of Pthread's Loop_schedule in parallel processing?

Loop schedule determines

- the initial value of the loop variable,
- the final value of the loop variable, and
- the increment for the loop variable.

3. What are the modes of message passing interfaces for send and its functions?

MPI provides four modes for sends:

- standard(MPI_Send)
- synchronous(MPI_Ssend)
- ready(MPI_Rsend)
- buffered(MPI_Bsend)

4. Brief about My_avail_tour_count functions.

The function My avail tour count can simply return the size of the process' stack. It can also make use of a ―cutoff length.‖ When a partial tour has already visited most of the cities, there will be very little work associated with the subtree rooted at the partial tour.

5. Distinguish between MPI_Pack and MPI_Unpack.

MPI_Pack	MPI_Unpack
Packing data into a buffer of contiguous memory	Unpacking data from a buffer of contiguous memory
Takes the data in data_to_be_packed and packs it into contig_buf	Takes the data in contig_buf and unpacks it into unpacked_data

6. What are the global variables for recursive depth first search?

- n: the total number of cities in the problem
- digraph: a data structure representing the input digraph

- hometown: a data structure representing vertex or city 0, the salesperson‘s hometown
- best tour: a data structure representing the best tour so far

7. Write about Fulfill_request functions.

If a process has enough work so that it can usefully split its stack, it calls Fulfill request. Fulfill request uses MPI Iprobe to check for a request for work from another process. If there is a request, it receives it, splits its stack, and sends work to the requesting process.

8. Brief about pthread_mutex_trylock.

Pthreads provides a nonblocking alternative to pthread_mutex_lock called pthread_mutex_trylock:

```
int pthread_mutex_trylock(
          pthread_mutex_t*   mutex_p   /* in/out */);
```

This function attempts to acquire mutex_p. However, if it‘s locked, instead of waiting, it returns immediately. The return value will be zero if the calling thread has successfully acquired the mutex, and nonzero if it hasn‘t. As an alternative to waiting on the mutex before splitting its stack, a thread can call pthread_mutex_trylock. If it acquires term_mutex, it can proceed as before. If not, it can just return. Presumably on a subsequent call it can successfully acquire the mutex.

9. What are the two phases for computation of forces?

First phase, each thread carries out exacltly the same calculations it carried out in the erroneous parallelization. Second phase, the thread that has been assigned particle q will add the contributions that have been computed by the different threads.

10. What is graph?

A graph is a pictorial representation of a set of objects where some pairs of objects are connected by links. The interconnected objects are represented by points termed as vertices, and the links that connect the vertices are called edges.

PART-B

nBODY SOLVERS

1. Explain in detail about how to parallelize the solvers using Pthreads and MPI? (16)

Parallelizing the solvers using pthreads:

Parallelizing the two n-body solvers using Pthreads is very similar to parallelizing them using OpenMP. The differences are only in implementation details, so rather than repeating the discussion, we will point out some of the principal differences between the Pthreads and the OpenMP implementations. We will also note some of the more important similarities.

- 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. To facilitate this, we‘ve written a function Loop_schedule, which determines
    - the initial value of the loop variable,
    - the final value of the loop variable, and
    - the increment for the loop variable.

    The input to the function is
    - the calling thread‘s rank,
    - the number of threads,
    - the total number of iterations, and
    - an argument indicating whether the partitioning should be block or cyclic.

Another difference between the Pthreads and the OpenMP versions has to do with barriers. Recall that the end of a parallel for directive in OpenMP has an implied barrier. As we‘ve seen, this is important. For example, we don‘t want a thread to start updating its positions until all the forces have been calculated, because it could use an out-of-date force and another thread could use an out of date position. If we simply partition the loop iterations among the threads in the Pthreads version, there won‘t be a barrier at the end of an inner for loop and we‘ll have a race condition.

Thus, we need to add explicit barriers after the inner loops when a race condition can arise. The Pthreads standard includes a barrier. However, some systems don‘t implement it, so we‘ve defined a function that uses a Pthreads condition variable to implement a barrier.

Parallelizing the basic solver using MPI:

With our composite tasks corresponding to the individual particles, it‘s fairly straightforward to parallelize the basic algorithm using 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. In the shared-memory implementations, we collected most of the data associated with a single particle (mass, position, and velocity) into a single struct. However, if we use this data structure in the MPI implementation, we‘ll need to use a derived datatype in the call to MPI_Allgather, and communications with derived datatypes
tend to be slower than communications with basic MPI types. Thus, it will make more sense to use individual arrays for the masses, positions, and velocities. We‘ll also need an array for storing the positions of all the particles. If each process has sufficient memory, then each of these can be a separate array. In fact, if memory isn‘t a problem, each process can store the entire array of masses, since these will never be updated and their values only need to be communicated during the initial setup. On the other hand, if memory is short, there is an ¯in-place‖ option that can be used with some MPI collective communications. For our situation, suppose that the array pos can store the positions of all n particles. Further suppose that vect_mpi_t is an MPI datatype that stores two contiguous doubles. Also suppose that n is evenly divisible by comm_sz and loc_n = n/comm_sz. Then, if we store the local positions in a separate array, loc_pos, we can use the following call to collect all of the positions on each process:

```
MPI_Allgather(loc_pos, loc_n, vect_mpi_t,
      pos, loc_n, vect_mpi_t, comm);
```

If we can‘t afford the extra storage for loc_pos, then we can have each process q store its local positions in the qth block of pos. That is, the local positions of each process should be stored in the appropriate block of each process‘ pos array:

```
Process 0:   pos[0], pos[1], . . . , pos[loc_n−1]
Process 1:   pos[loc_n], pos[loc_n+1], . . . , pos[loc_n + loc_n−1]
   . . .
Process q:   pos[q*loc_n], pos[q*loc_n+1], . . . , pos[q*loc_n +
     loc_n−1]
   . . .
```

With the pos array initialized this way on each process, we can use the following call to MPI_Allgather:

```
MPI_Allgather(MPI_IN_PLACE, loc_n, vect_mpi_t,
  pos, loc_n, vect_mpi_t, comm);
```

In this call, the first loc n and vect mpi t arguments are ignored. However, it‘s not a bad idea to use arguments whose values correspond to the values that will be used, just to increase the readability of the program.

```
Get input data;
for each timestep {
   if (timestep output)
      Print positions and velocities of particles;
   for each local particle loc_q
      Compute total force on loc_q;
   for each local particle loc_q
      Compute position and velocity of loc_q;
   Allgather local positions into global pos array;
}
Print positions and velocities of particles;
```

Pseudocode for the MPI version of the basic n-body solver

2. Explain in detail about performance of the MPI Solvers. (16)

Table 5.1 shows the run-times of the two n-body solvers when they‘re run with 800 particles for 1000 timesteps on Infiniband-connected cluster.

Table 5.1: 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

All the timings were taken with one process per cluster node. 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.

For example, the efficiency of the basic solver on 16 nodes is about 0.95, while the efficiency of the reduced solver on 16 nodes is only about 0.70.

A point to stress here is that the reduced MPI solver makes much more efficient use of memory than the basic MPI solver; the basic solver must provide storage for all n positions on each process, while the reduced solver only needs extra storage for n/comm_sz positions and n/comm_sz forces. Thus, the extra storage needed on each process for the basic solver is nearly comm_sz/2 times greater than the storage needed for the reduced solver. When n and comm_sz are very large, this factor can easily make the difference between being able to run a simulation only using the process' main memory and having to use secondary storage.

The nodes of the cluster on which we took the timings have four cores, so we can compare the performance of the OpenMP implementations with the performance of the MPI implementations.

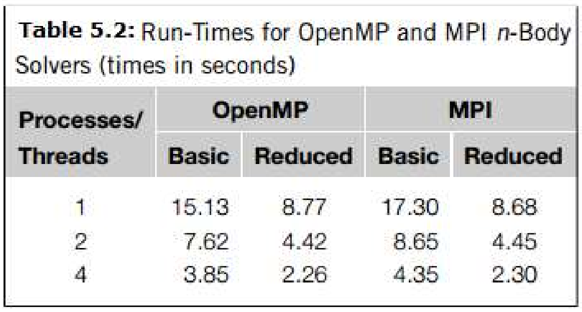


Table 5.2: Run-Times for OpenMP and MPI *n*-Body Solvers (times in seconds)

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

We see that the basic OpenMP solver is a good deal faster than the basicMPI solver. This isn't surprising since MPI Allgather is such an expensive operation. Perhaps surprisingly, though, the reduced MPI solver is quite competitive with the reduced OpenMP solver.

Let's take a brief look at the amount of memory required by the MPI and OpenMP reduced solvers. Say that there are n particles and p threads or processes. Then each solver will allocate the same amount of storage for the local velocities and the local positions. The MPI solver allocates n doubles per process for the masses. It also allocates 4n/p doubles for the tmp_pos and tmp_forces arrays, so in addition to the local velocities and positions, the MPI solver stores

$$n+4n/p$$

doubles per process. The OpenMP solver allocates a total of 2pn + 2n doubles for the forces and n doubles for the masses, so in addition to the local velocities and positions, the OpenMP solver stores

$$3n/p+2n$$

doubles per thread. Thus, the difference in the local storage required for the OpenMP version and the MPI version is

$$n-n/p$$

doubles. In other words, if n is large, the local storage required for the MPI version is substantially less than the local storage required for the OpenMP version. So, for a fixed number of processes or threads, we should be able to run much larger simulations with the MPI version than the OpenMP version. Of course, because of hardware considerations, we‘re likely to be able to use many more MPI processes than OpenMP threads, so the size of the largest possible MPI simulations should be much greater than the size of the largest possible OpenMP simulations. The MPI version of the reduced solver is much more scalable than any of the other versions, and the ―ring pass‖ algorithm provides a genuine breakthrough in the design of n-body solvers.

TREE SEARCH

3. Explain the following tree search mechanisms: (16)

a) Recursive depth-first search
b) Non-recursive depth-first search
a) Recursive depth-first search:

Using depth-first search we can systematically visit each node of the tree that could possibly lead to a least-cost solution. The simplest formulation of depth-first search uses recursion (see Figure 5.1). Later on it will be useful to have a definite order in which the cities are visited in the for loop in Lines 8 to 13, so we‘ll assume that the cities are visited in order of increasing index, from city 1 to city n - 1.

The algorithm makes use of several global variables:

- n: the total number of cities in the problem
- digraph: a data structure representing the input digraph
- hometown: a data structure representing vertex or city 0, the salesperson‘s hometown
- best_tour: a data structure representing the best tour so far

The function City_count examines the partial tour tour to see if there are n cities on the partial tour. If there are, we know that we simply need to return to the hometown to complete the tour, and we can check to see if the complete tour has a lower cost than the current ―best tour‖ by calling Best_tour. If it does, we can replace the current best tour with this tour by calling the function Update_best_tour. Note that before the first call to Depth_first_search, the best_tour variable should be initialized so that its cost is greater than the cost of any possible least-cost tour.

If the partial tour tour hasn‘t visited n cities, we can continue branching down in the tree by ―expanding the current node,‖ in other words, by trying to visit other cities from the city last visited in the partial tour. To do this we simply loop through the cities. The function Feasible checks to see if the city or vertex has already been visited, and, if not, whether it can possibly lead to a least-cost tour. If the city is feasible, we add it to the tour, and recursively call Depth_first_search.

```
void Depth_first_search(tour_t tour) {
   city_t city;

   if (City_count(tour) == n) {
      if (Best_tour(tour))
         Update_best_tour(tour);
   } else {
      for each neighboring city
         if (Feasible(tour, city)) {
            Add_city(tour, city);
            Depth_first_search(tour);
            Remove_last_city(tour, city);
         }
   }
}  /* Depth_first_search */
```

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

When we return from Depth_first_search, we remove the city from the tour, since it shouldn‘t be included in the tour used in subsequent recursive calls.

b) Non-recursive depth-first search:

Since function calls are expensive, recursion can be slow. It also has the disadvantage that at any given instant of time only the current tree node is accessible. This could be a problem when we try to parallelize tree search by dividing tree nodes among the threads or processes.

It is possible to write a nonrecursive depth-first search. The basic idea is modeled on recursive implementation. Recall that recursive function calls can be implemented by pushing the current state of the recursive function onto the run-time stack. Thus, we can try to eliminate recursion by pushing necessary data on our own stack before branching deeper into the tree, and when we need to go back up the tree–either because we‘ve reached a leaf or because we‘ve found a node that can‘t lead to a better solution–we can pop the stack.

This outline leads to the implementation of iterative depth-first search shown in Program. In this version, a stack record consists of a single city, the city that will be added to the tour when its record is popped. In the recursive version we continue to make recursive calls until we‘ve visited every node of the tree that corresponds to a feasible partial tour. At this point, the stack won‘t have any more activation records for calls to Depth_first_search, and we‘ll return to the function that made the original call to 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 */
```

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

The main control structure in our iterative version is the while loop extending from Line 3 to Line 20, and the loop termination condition is that our stack is empty. As long as the search needs to continue, we need to make sure the stack is nonempty, and, in the first two lines, we add each of the non-hometown cities. Note that this loop visits the cities in decreasing order, from n - 1 down to 1. This is because of the order created by the stack, whereby the stack pops the top cities first. By reversing the order, we can insure that the cities are visited in the same order as the recursive function. Also notice that in Line 5 we check whether the city we‘ve popped is the constant NO_CITY. This constant is used so that we can tell when we‘ve visited all of the children of a tree node; if we didn‘t use it, we wouldn‘t be able to tell when to back up in the tree. Thus, before pushing all of the children of a node (Lines 15-17), we push the NO_CITY marker.

4. Discuss briefly about the data structures and performance of the serial implementations. (16)

Data structures for the serial implementations:

Our principal data structures are the tour, the digraph, and, in the iterative implementations, the stack. The tour and the stack are essentially list structures. In problems that we‘re likely to be able to tackle, the number of cities is going to be small– certainly less than 100–so there‘s no great advantage to using a linked list to represent the tours and we‘ve used an array that can store n C 1 cities. We repeatedly need both the number of cities in the partial tour and the cost of the partial tour. Therefore, rather than just using an array for the tour data structure and recomputing these values, we use a struct with three members: the array storing the cities, the number of cities, and the cost of the partial tour.

To improve the readability and the performance of the code, we can use preprocessor macros to access the members of the struct. However, since macros can be a nightmare to debug, it‘s a good idea to write ―accessor‖ functions for use during initial development. When the program with accessor functions is working, they can be replaced with macros. As an example, we might start with the function

```
/* Find the ith city on the partial tour */
int Tour_city(tour_t tour, int i) {
   return tour->cities[i];
}  /* Tour_city */
```

When the program is working, we could replace this with the macro

```
/* Find the ith city on the partial tour */
#define Tour_city(tour, i) (tour->cities[i])
```

The stack in the original iterative version is just a list of cities or ints. Furthermore, since there can‘t be more than $n^2/2$ records on the stack at any one time, and n is likely to be small, we can just use an array, and like the tour data structure, we can store the number of elements on the stack. Thus, for example, Push can be implemented with

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

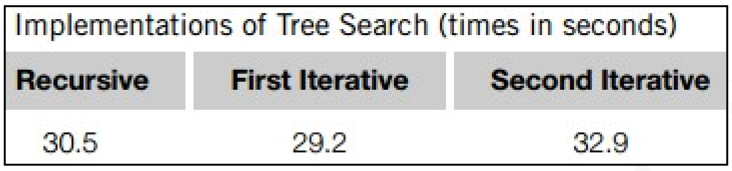
In the second iterative version, the version that stores entire tours in the stack, we can probably still use an array to store the tours on the stack. Now the push function will look something like this:

```
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 */
```

Once again, element access for the stack can be implemented with macros. There are many possible representations for digraphs. When the digraph has relatively few edges, list representations are preferred. However, in our setting, if vertex i is different from vertex j, there are directed, weighted edges from i to j and from j to i, so we need to store a weight for each ordered pair of distinct vertices i and j. Thus, in our setting, an adjacency matrix is almost certainly preferable to a list structure. This is an n x n matrix, in which the weight of the edge from vertex i to vertex j can be the entry in the ith row and jth column of the matrix. We can access this weight directly, without having to traverse a list. The diagonal elements (row i and column i) aren‘t used, and we‘ll set them to 0.

Performance of the serial implementations:

The run-times of the three serial implementations are shown in Table.



Implementations of Tree Search (times in seconds)

Recursive	First Iterative	Second Iterative
30.5	29.2	32.9

The input digraph contained 15 vertices (including the hometown), and all three algorithms visited approximately 95,000,000 tree nodes. The first iterative version is less than 5% faster than the recursive version, and the second iterative version is about 8% slower than the recursive version. As expected, the first iterative solution eliminates some of the overhead due to repeated function calls, while the second iterative solution is slower because of the repeated copying of tour data structures. However, as we‘ll see, the second iterative solution is relatively easy to parallelize, so we‘ll be using it as the basis for the parallel versions of tree search.

OpenMP AND MPI IMPLEMENTATIONS AND COMPARISONS

5. Explain briefly about the implementation of tree search using MPI and static partitioning. (16)

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.

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. For example, even if we have 100 cities, it‘s unlikely that the matrix will require more than 80,000 bytes of storage, so it makes sense to simply read in the matrix on process 0 and broadcast it to all the processes.

Once the processes have copies of the adjacency matrix, the bulk of the tree search can proceed as it did in the Pthreads and OpenMP implementations. The principal differences lie in

- partitioning the tree,
- checking and updating the best tour, and
- after the search has terminated, making sure that process 0 has a copy of the best tour for output.

Partitioning the tree

In the Pthreads and OpenMP implementations, thread 0 uses breadth-first search to search the tree until there are at least thread_count partial tours. Each thread then determines which of these initial partial tours it should get and pushes its tours onto its local stack. Certainly MPI process 0 can also generate a list of comm_sz partial tours. However, since memory isn‘t shared, it will need to send the initial partial tours to the appropriate process. We could do this using a loop of sends, but distributing the initial partial tours looks an awful lot like a call to MPI_Scatter. In fact, the only reason we can‘t use MPI_Scatter is that the number of initial partial tours may not be evenly divisible by comm_sz. When this happens, process 0 won‘t be sending the same number of tours to each process, and MPI_Scatter requires that the source of the scatter send the same number of objects to each process in the communicator.

Fortunately, there is a variant of MPI_Scatter, MPI_Scatterv, which can be used to send different numbers of objects to different processes. First recall the syntax of MPI_Scatter:

```
int MPI_Scatter(
        void            sendbuf     /* in  */,
        int             sendcount   /* in  */,
        MPI_Datatype    sendtype    /* in  */,
        void*           recvbuf     /* out */,
        int             recvcount   /* in  */,
        MPI_Datatype    recvtype    /* in  */,
        int             root        /* in  */,
        MPI_Comm        comm        /* in  */);
```

Process root sends sendcount objects of type sendtype from sendbuf to each process in comm. Each process in comm receives recvcount objects of type recvtype into recvbuf. Most of the time, sendtype and recvtype are the same and sendcount and recvcount are also the same. In any case, it‘s clear that the root process must send the same number of objects to each process. MPI_Scatterv, on the other hand, has syntax

```
int MPI_Scatterv(
      void*         sendbuf         /* in  */,
      int*          sendcounts      /* in  */,
      int*          displacements   /* in  */,
      MPI_Datatype  sendtype        /* in  */,
      void*         recvbuf         /* out */,
      int           recvcount       /* in  */,
      MPI_Datatype  recvtype        /* in  */,
      int           root            /* in  */,
      MPI_Comm      comm            /* in  */);
```

The single sendcount argument in a call to MPI_Scatter is replaced by two array arguments: sendcounts and displacements. Both of these arrays contain comm_sz elements: sendcounts[q] is the number of objects of type sendtype being sent to process q. Furthermore, displacements[q] specifies the start of the block that is being sent to process q. The displacement is calculated in units of type sendtype. So, for example, if sendtype is MPI_INT, and sendbuf has type int*, then the data that is sent to process q will begin in location

```
sendbuf + displacements[q]
```

In general, displacements[q] specifies the offset into sendbuf of the data that will go to process q. The ―units‖ are measured in blocks with extent equal to the extent of sendtype.

Similarly, MPI_Gatherv generalizes MPI_Gather:

```
int MPI_Gatherv(
      void*         sendbuf         /* in  */,
      int           sendcount       /* in  */,
      MPI_Datatype  sendtype        /* in  */,
      void*         recvbuf         /* out */,
      int*          recvcounts      /* in  */,
      int*          displacements   /* in  */,
      MPI_Datatype  recvtype        /* in  */,
      int           root            /* in  */,
      MPI_Comm      comm            /* in  */);
```

```
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 */
```

MPI code to check for new best tour costs

If msg_avail is true, then we can receive the new cost with a call to MPI_Recv:

```
MPI_Recv(&received_cost, 1, MPI_INT, status.MPI_SOURCE,
      NEW_COST_TAG, comm, MPI_STATUS_IGNORE);
```

This code will continue to receive messages with new costs as long as they‘re available. Each time a new cost is received that‘s better than the current best cost, the variable best tour cost will be updated.

Printing the best tour

When the program finishes, we‘ll want to print out the actual tour as well as its cost, so we do need to get the tour to process 0. It might at first seem that we could arrange this by having each process store its local best tour—the best tour that it finds—and when the tree search has completed, each process can check its local best tour cost and compare it to the global best tour cost. If they‘re the same, the process could send its local best tour to process 0. There are, however, several problems with this. First, it‘s entirely possible that there are multiple ―best‖ tours in the TSP digraph, tours that all have the same cost, and different processes may find these different tours. If this happens, multiple processes will try to send their best tours to process 0, and all but one of the threads could hang in a call to MPI Send. A second problem is that it‘s possible that one or more processes never received the best tour cost, and they may try to send a tour that isn‘t optimal.

We can avoid these problems by having each process store its local best tour, but after all the processes have completed their searches, they can all call MPI_Allreduce and the process with the global best tour can then send it to process 0 for output. The following pseudocode provides some details:

```
struct {
   int cost;
   int rank;
} loc_data, global_data;

loc_data.cost = Tour_cost(loc_best_tour);
loc_data.rank = my_rank;
MPI_Allreduce(&loc_data, &global_data, 1, MPI_2INT, MPI_MINLOC,
     comm);
if (global_data.rank == 0) return;
     /* 0 already has the best tour */
if (my_rank == 0)
   Receive best tour from process global_data.rank;
else if (my_rank == global_data.rank)
   Send best tour to process 0;
```

The key here is the operation we use in the call to MPI_Allreduce. If we just used MPI_MIN, we would know what the cost of the global best tour was, but we wouldn‘t know who owned it. However, MPI provides a predefined operator, MPI_MINLOC, which operates on pairs of values. The first value is the value to be minimized—in our setting, the cost of the tour—and the second value is the location of the minimum—in our setting, the rank of the process that actually owns the best tour. If more than one process owns a tour with minimum cost, the location will be the lowest of the ranks of the processes that own a minimum cost tour. The input and the output buffers in the call to MPI_Allreduce are two-member structs. Since both the cost and the rank are ints, both members are ints. Note that MPI also provides a predefined type MPI_2INT for this type. When the call to MPI_Allreduce returns, we have two alternatives:

*If process 0 already has the best tour, we simply return.

*Otherwise, the process owning the best tour sends it to process 0.