CS6801 MULTI-CORE ARCHITECTURE AND PROGRAMMING

Unit 1 : MULTI-CORE PROCESSORS

Part A

1. Define multi-core processor.

   A multiprocessor system is one where there are multiple microprocessors plugged into the system board. When each processor can run only a single thread, there is a relatively simple relationship between the number of processors, CPUs, chips, and cores in a system—they are all equal

2. Define SIMD systems?

   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.

3. Define MIMD systems?

   Multiple instruction, multiple data, or MIMD, systems support multiple simultaneous instruction streams operating on multiple data streams. Thus, MIMD systems typically consist of a collection of fully independent processing units or cores, each of which has its own control unit and its own ALU.

4. What is shared memory system?

   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.

5. What is distributed memory system?

   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 multi-core processors.

6. Define Bisection bandwidth?

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.

7. What is cache coherence?

single processor systems provide no mechanism for insuring that when the caches of multiple processors store the same variable, an update by one processor to the cached variable is “seen” by the other processors. That is, that the cached value stored by the other processors is also updated. This is called the cache coherence problem.

8. What is 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.

9. What is directory based cache coherence?

Directory-based cache coherence uses the 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.

10. What is false sharing?

False sharing is a term which applies when threads unwittingly impact the performance of each other while modifying independent variables sharing the same cache line. Write contention on cache lines is the single most limiting factor on achieving scalability for parallel threads of execution in an SMP system.

11. What is linear speedup?

If 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.

12. Define efficiency?

It can be defined as ratio between speedup and no of cores in parallel system,

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

13. Define Amdahl’s law?

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.

14. What is scalability in multi-core architecture?

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.

15. What are the steps involved in parallel programming design?

- Partitioning
- Communication
- Aggregation
- Mapping

16. Define centralized multicore architecture?

The centralized multi core is the direct extension of the unicore architecture. The unicore uses the bus to connect a core with primary memory and input output core. The cache memory makes CPU always busy. In this, the addition cores are added to the bus and the entire core share the same primary memory.

17. What is Ring Interconnect?

The Ring Interconnect is superior to bus and it allows multiple simultaneous communications. In Ring Interconnect, it is simple to formulate the communication methodology in which source of the processors has to wait for other processors to complete their communication process.

18. What are the different Interconnects in Direct Interconnect network?

    In direct interconnect, each switch is directly connected to the processor memory pairs and also the switches are connected to each other. It has following types.

    - Ring Interconnect
    - Toroidal Mesh Interconnect
    - Hybercube Interconnect
    - Fully connected Interconnect

19. Define fully connected Interconnect?

    The complete direct fully interconnect ntwork is the direct interconnect. In this, each switch in the network is directly connected to every other switch .

20. Define Omega indirect interconnect?

    In omega indirect interconnect, there are some communication path which cannot occur and process simultaneously. The construction of Omega indirect interconnect is less expensive than crossbar interconnect. The omega interconnect make “2plog2(p)” switches for connections with p number of processors.

Part B

1. Explain the difference between SIMD and MIMD systems?

SIMD systems

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 *i*th ALU, have the *i*th 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* =4 and *n* =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.

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.*
- *Vectorized and pipelined functional units.*
- *Vector instructions.*
- *Interleaved memory.*
- *Strided memory access and hardware scatter/gather.*

MIMD systems

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

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.

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.

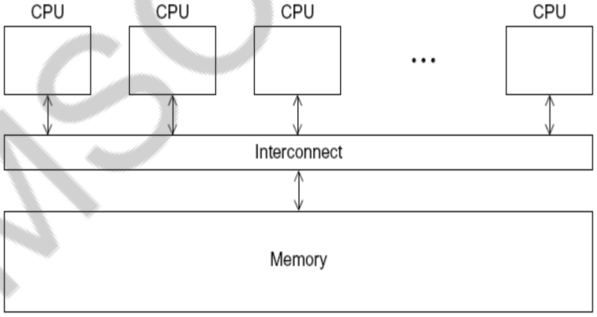


Figure 1.1: A shared-memory system

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 multi core 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.

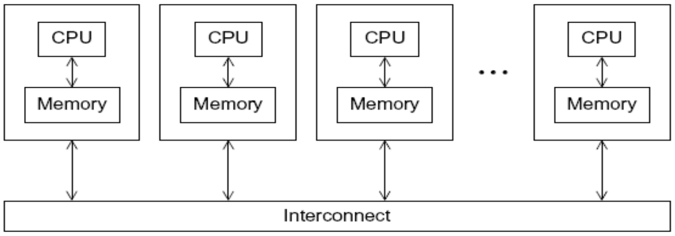


Figure 1.2: A Distributed memory system

2. Explain about various types of interconnection networks?

Interconnection networks

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. As the name suggests, **switched** interconnects use switches to control the routing of data among the connected devices.

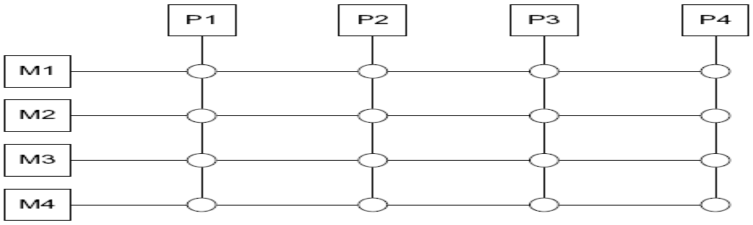


Figure 1.3: A crossbar switch connecting four processors (*Pi*) and four memory modules (*Mj*);

Distributed-memory interconnects

Distributed-memory interconnects are often divided into two groups: direct interconnects and indirect interconnects.

Direct Interconnect

In a **direct interconnect** each switch is directly connected to a processor-memory pair, and the switches are connected to each other. Figure 1.4 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 3*p* in a toroidal mesh, while it's only 2*p* 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.

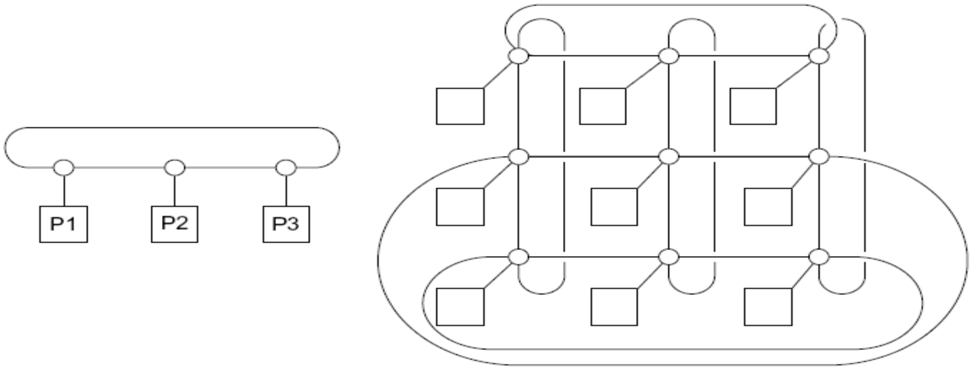


Figure 1.4: (a) A ring and (b) a toroidal mesh

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.

The **hypercube** is a highly connected direct interconnect that has been used in actual systems. Hypercubes are built inductively: A one-dimensional hypercube is a fully-connected system with two processors. A two-dimensional hypercube is built from two one-dimensional hypercubes by joining “corresponding” switches.

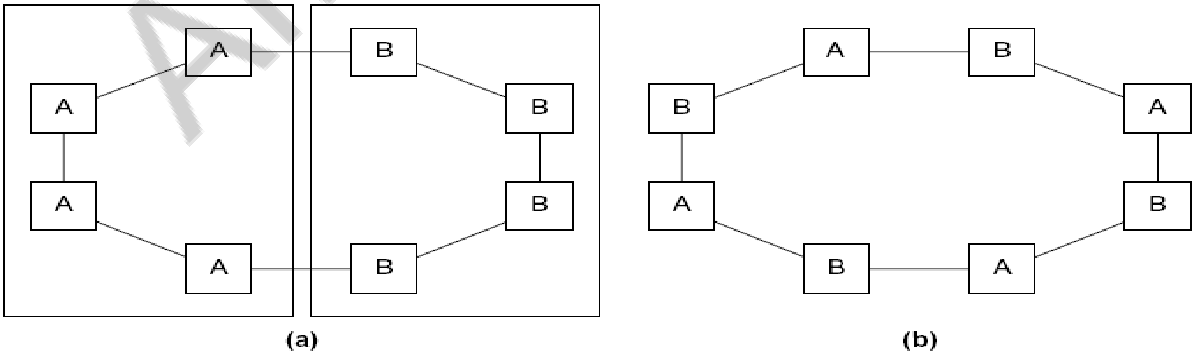


Figure 1.5 : Two bisections of a ring: (a) only two communications can take place between the halves and (b) four simultaneous connections can take place

Indirect Interconnects

Indirect interconnects provide an alternative to direct interconnects. In an indirect interconnect, the switches may not be directly connected to a processor. They're often shown with unidirectional links and a collection of processors, each of which has an outgoing and an incoming link, and a switching network.

An omega network is shown in Figure 1.8. The switches are two-by-two crossbars . Observe that unlike the crossbar, there are communication that cannot occur simultaneously. For example, if processor 0 sends a message to processor 6, then processor 1 cannot simultaneously send a message to processor 7. On the other hand, the omega network is less expensive than the crossbar. The omega network uses ½ $p\log_2(p)$ of the 2*2 crossbar switches, so it uses a total of $2p\log_2(p)$ switches, while the crossbar uses *p*2.

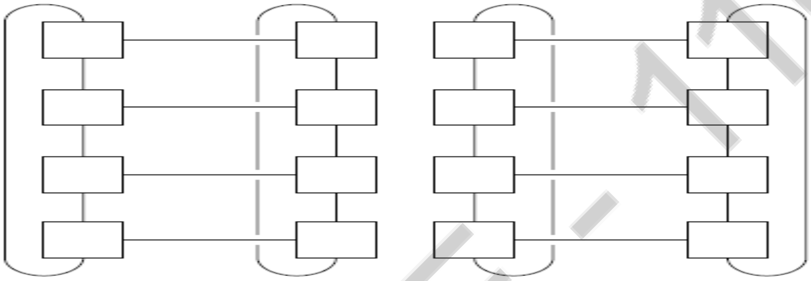


Figure1.6 :A bisection of a toroidal mesh

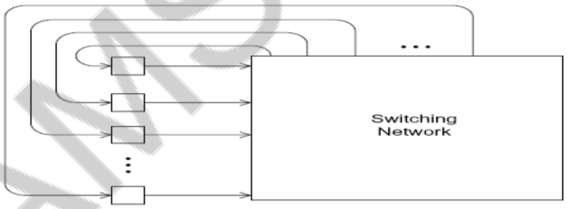


Figure 1.7 : Generic indirect network

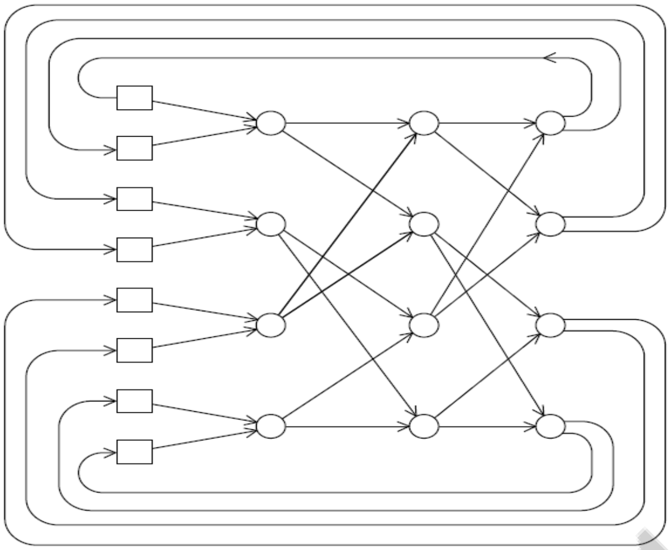


Figure 1.8: Omega network

3. Explain various the performance issues in multi-core architecture?

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 D *T*serial=*p*. When this happens, we say that our parallel program has **linear speedup.** So if we define the **speedup** of a parallel program to be

$$S =\mathrm{l}\frac{T\ serial}{T\ parallel}$$

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

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

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

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

and the speedup will be

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

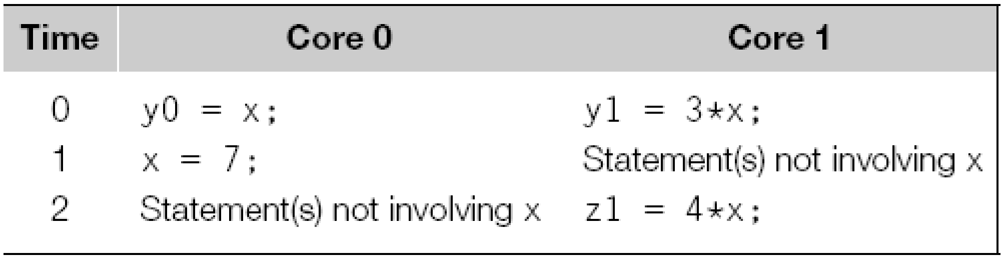
Scalability

The word “scalable” has a wide variety of informal uses. Indeed, we’ve used it several times already. Roughly speaking, a technology is scalable if it can handle ever-increasing problem sizes. However, in discussions of parallel program performance, scalability has a somewhat more formal definition. 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**.

4. **Explain various techniques used in cache coherence?**

Cache coherence

Recall that CPU caches are managed by system hardware: programmers don’t have direct control over them. This has several important consequences for shared-memory systems. To understand these issues, suppose we have a shared memory system with two cores, each of which has its own private data cache. As long as the two cores only read shared data, there is no problem. For example, suppose that x is a shared variable that has been initialized to 2, y0 is private and owned by core 0, and y1 and z1 are private and owned by core 1. Now suppose the following statements are executed at the indicated times:



Time	Core 0	Core 1
0	y0 = x;	y1 = 3*x;
1	x = 7;	Statement(s) not involving x
2	Statement(s) not involving x	z1 = 4*x;

Then the memory location for y0 will eventually get the value 2, and the memory location for y1 will eventually get the value 6. However, it's not so clear what value z1 will get. It might at first appear that since core 0 updates x to 7 before the assignment to z1, z1 will get the value 4_7 D 28. However, at time 0, x is in the cache of core 1. So unless for some reason x is evicted from core 0's cache and then reloaded into core 1's cache, it actually appears that the original value x = 2 may be used, and z1 will get the value 4_2 D 8.

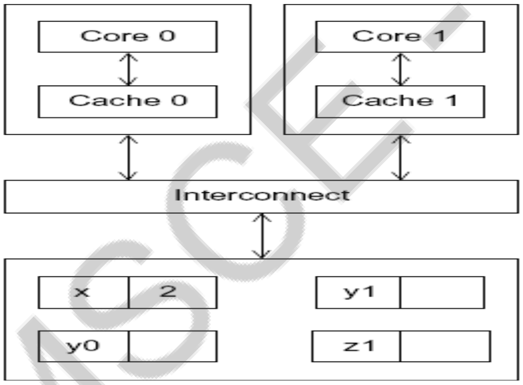


Figure 1.8 :A shared-memory system with two cores and two caches

Snooping cache coherence

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

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.

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.

False sharing

False sharing is a term which applies when threads unwittingly impact the performance of each other while modifying independent variables sharing the same cache line. Write contention on cache lines is the single most limiting factor on achieving scalability for parallel threads of execution in an SMP system.

5. **Explain about single core and multi-core architecture in detail?**

<u>SINGLE-CORE PROCESSORS</u>

A single-core processor is a microprocessor with a single core on a chip, running a single thread at any one time. The term became common after the emergence of multi-core processors (which have several independent processors on a single chip) to distinguish non-multi-core designs. For example, Intel released a Core 2 Solo and Core 2 Duo, and one would refer to the former as the 'single-core' variant. Most microprocessors prior to the multi-core era are single-core. The class of many-core processors follows on from multi-core, in a progression showing increasing parallelism over time.

Processors remained single-core until it was impossible to achieve performance gains from the increased clock speed and transistor count allowed by Moore's law (there were diminishing returns to increasing the depth of a pipeline, increasing CPU cache sizes, or adding execution units).

Problems of Single Core Processors:

As we try to increase the clock speed of this processor, the amount of heat produced by the chip also increases. It is a big hindrance in the way of single core processors to continue evolving.

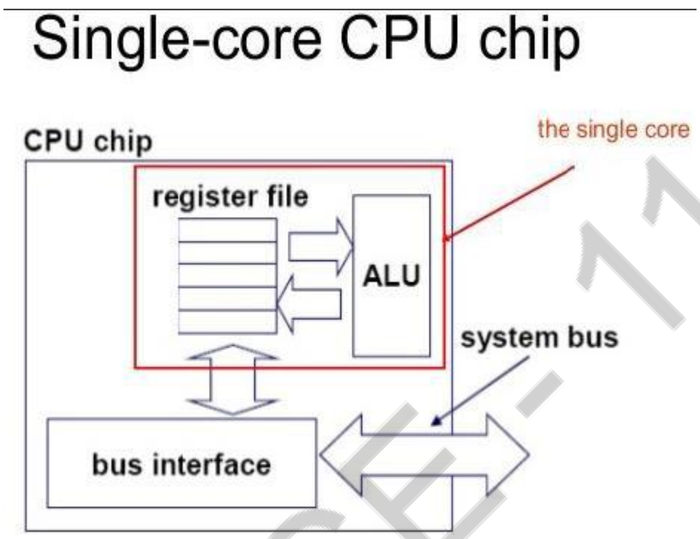


Figure 1.9: single-core architecture

MULTI-CORE PROCESSORS

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. 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.

A multi-core processor implements multiprocessing in a single physical package. Designers may couple cores in a multi-core device tightly or loosely. For example, cores may or may not share caches, and they may implement message passing or shared-memory inter-core communication methods. Common network topologies to interconnect cores include bus, ring, two-dimensional mesh, and crossbar.

Homogeneous multi-core systems include only identical cores; heterogeneous multi-core systems have cores that are not identical (e.g. big.LITTLE have heterogeneous cores that share the same instruction set, while AMD Accelerated Processing Units have cores that don't even share the same instruction set). Just as with single-processor systems, cores in multi-core systems may implement architectures such as VLIW, superscalar, vector, or multithreading.

Multi-core processors are widely used across many application domains, including general-purpose, embedded, network, digital signal processing (DSP), and graphics (GPU).

The improvement in performance gained by the use of a multi-core processor depends very much on the software algorithms used and their implementation. In particular, possible gains are limited by the fraction of the software that can run in parallel simultaneously on multiple cores; this effect is described by Amdahl's law. In the best case, so-called embarrassingly parallel problems may realize speedup factors near the number of cores, or even more if the problem is split up enough to fit within each core's cache(s), avoiding use of much slower main-system memory.

Most applications, however, are not accelerated so much unless programmers invest a prohibitive amount of effort in re-factoring the whole problem. The parallelization of software is a significant ongoing topic of research.

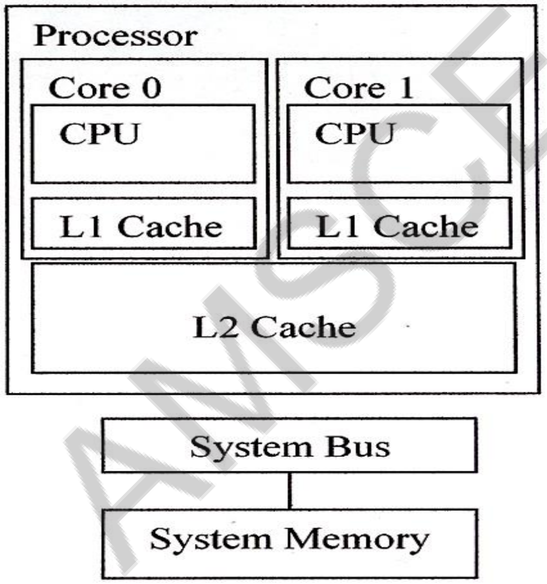


Figure 1.9: multi-core architecture

Problems with multicore processors:

According to Amdahl’s law, the performance of parallel computing is limited by its serial components. So, increasing the number of cores may not be the best solution. There is need to increase the clock speed of individual cores.

COMPARISON OF SINGLE-CORE PROCESSOR AND MULTI-CORE PROCESSOR

Parameter	Single-Core Processor	Multi-Core Processor
Number of cores on a die	Single	Multiple
Instruction Execution	Can execute Single instruction at a time	Can execute multiple instructions by using multiple cores
Gain	Speed up every program or software being executed	Speed up the programs which are designed for multi-core processors
Performance	Dependent on the clock frequency of the core	Dependent on the frequency, number of cores and program to be executed
Examples	Processor launched before 2005 like 80386,486, AMD 29000, AMD K6, Pentium I,II,III etc.	Processor launched after 2005 like Core-2-Duo,Athlon 64 X2, I3,I5 and I7 etc.

Unit 2 : PARALLEL PROGRAM CHALLENGES

Part A

1. Define data races problem?

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.

2. What is mutex lock in synchronization?

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. Following coding shows how a mutex lock could be used to protect access to a variable ‘counter’.

```
int counter;
mutex_lock mutex;
void Increment()
{
acquire( &mutex );
counter++;
release( &mutex );
}
void Decrement()
{
acquire( &mutex );
counter--;
release( &mutex );
}
```

3. What is critical region? give an example

   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.

4. Define spinlock in multicore architecture?

   A spinlock is a mutual exclusion device that can have only two values: "locked" and "unlocked." It is usually implemented as a single bit in an integer value. Code wishing to take out a particular lock tests the relevant bit. If the lock is available, the "locked" bit is set and the code continues into the critical section. If, instead, the lock has been taken by somebody else, the code goes into a tight loop where it repeatedly checks the lock until it becomes available. This loop is the "spin" part of a spinlock.

5. What are the difference between mutex lock and spinlock?

   Spin locks are essentially mutex locks. The difference between a mutex lock and a spinlock 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.

6. What is semaphore?

   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.

7. What is reader-writer lock?

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.

8. What is 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.

9. What is deadlock and livelock?

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*.

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

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

10. What are the different communication methods for threads?

Following are the communication methods used for threads,

- Condition variables.
- Signals
- Message queues
- Named pipes

11. What is condition variable communication method?

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.

12. What is producer –consumer problem?

The producer–consumer problem is a classic example of a multi process synchronization problem. The problem describes two processes, the producer and the consumer, who share a common, fixed-size buffer used as a queue. The producer's job is to generate data, put it into the buffer, and start again. At the same time, the consumer is consuming the data (i.e., removing it from the buffer), one piece at a time. The problem is to make sure that the producer won't try to add data into the buffer if it's full and that the consumer won't try to remove data from an empty buffer.

13. What is signal based communication?

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.

14. What is queue based communication?

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.

15. What is pipes based communication?

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.

16. What is POSIX thread?

POSIX Threads, usually referred to as Pthreads, is an execution model that exists independently from a language, as well as a parallel execution model. It allows a program to control multiple different flows of work that overlap in time. Each flow of work is referred to as a thread, and creation and control over these flows is achieved by making calls to the POSIX Threads API.

17. Give the steps to create POSIX thread?

```
#include <pthread.h>
#include <stdio.h>
void* thread_code( void * param )
{
printf( "In thread code\n" );
}
int main()
{
pthread_t thread;
pthread_create( &thread, 0, &thread_code, 0 );
printf( "In main thread\n" );
}
```

An application initially starts with a single thread, which is often referred to as the *main thread* or the *master thread*. Calling pthread_create() creates a new thread. It takes the following parameters:

- A pointer to a pthread_t structure. The call will return the handle to the thread in this structure.
- A pointer to a pthread attributes structure, which can be a null pointer if the default attributes are to be used. The details of this structure will be discussed later.
- The address of the routine to be executed.
- A value or pointer to be passed into the new thread as a parameter.

18. Define safe Dead Lock?

The safe Deadlock occurs when a thread T1 need to acquire a lock on Resouce R1 which is already owned by the save thread T1.

19. What is recursive Deadlock?

    When the self deadlock condition occur between two threads T1 and T2, then it is said to be recursive deadlock.

20. Mansion the four necessary condition to occur deadlock?
    - Mutual exclusive.
    - Hold and wait
    - Circular wait
    - No preemption

Part B

1. **Explain about various types of synchronization primitives in detail?**

Synchronization Primitives

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.

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.

```
int counter;
mutex_lock mutex;
void Increment()
{
acquire( &mutex );
counter++;
release( &mutex );
}
void Decrement()
{
acquire( &mutex );
counter--;
release( &mutex );
}
```

In this 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.

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.

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.

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.

```
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 );
}
```

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. Following statements shows a situation using a barrier to separate the calculation of a variable from its use.

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

DeadLock and LiveLock

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*.

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

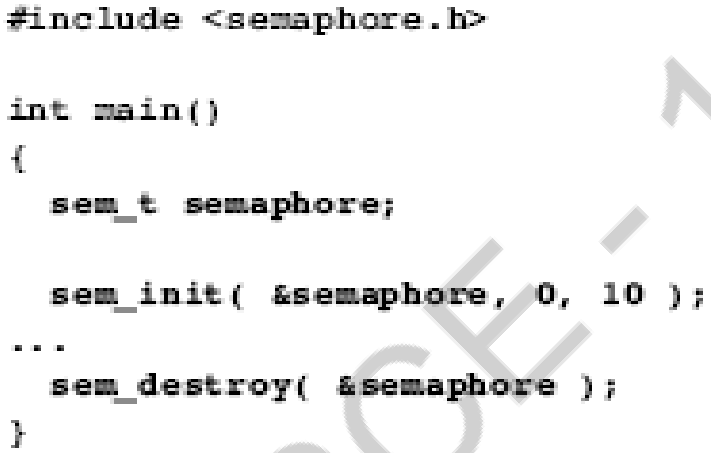
```
Thread 1                          Thread 2
void update1()                   void update2()
{                                {
acquire(A);                       acquire(B);
acquire(B); <<< Thread 1         acquire(A); <<< Thread 2
waits here                        waits here
variable1++;         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

2. **Explain about semaphore with example?**

Semaphores

A *semaphore* is a counting and signaling mechanism. One use for it is to allow threads access to a specified number of items. If there is a single item, then a semaphore is essentially the same as a mutex, but it is more commonly useful in a situation where there are multiple items to be managed. Semaphores can also be used to signal between threads or processes, for example, to tell another thread that there is data present in a queue. There are two types of semaphores: named and unnamed semaphores. An *unnamed* semaphore is initialized with a call to sem_init(). This function takes three parameters. The first parameter is a pointer to the semaphore. The next is an integer to indicate whether the semaphore is shared between multiple processes or private to a single process. The final parameter is the value with which to initialize the semaphore.

A semaphore created by a call to sem_init() is destroyed with a call to sem_destroy(). The code shown in following initializes a semaphore with a count of 10. The middle parameter of the call to sem_init() is zero, and this makes the semaphore private to the thread; passing the value one rather than zero would enable the semaphore to be shared between multiple processes.

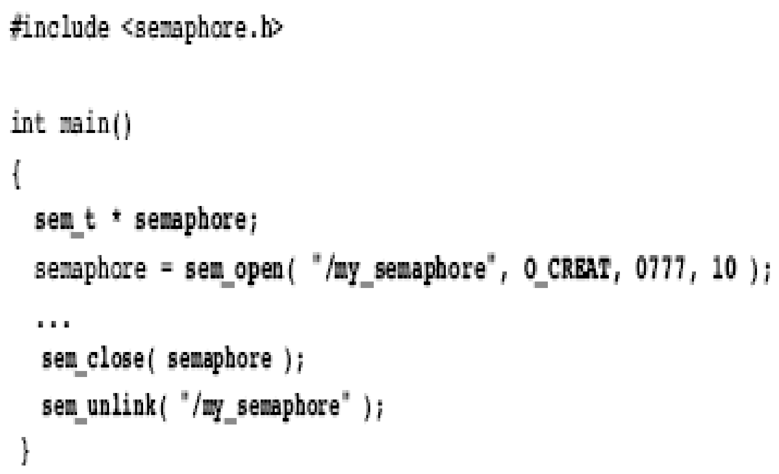


```
#include <semaphore.h>

int main()
{
  sem_t semaphore;

  sem_init( &semaphore, 0, 10 );
...
  sem_destroy( &semaphore );
}
```

A *named* semaphore is opened rather than initialized. The process for doing this is similar to opening a file. The call to sem_open() returns a pointer to a semaphore. The first parameter to the call is the name of the semaphore. The name must conform to the naming conventions for files on the operating system and must start with a single slash sign and contain no further slash signs. The next parameter is the set of flags. There are three combinations of flags that can be passed to the sem_open() call. If no flags are passed, the function will return a pointer to the existing named semaphore if it exists and if the semaphore has the appropriate permissions to be shared with the calling process. If the O_CREAT flag is passed, the semaphore will be created; if it does not exist or if it does exist, a pointer will be returned to the existing version. The flag O_EXCL can be passed with the O_CREAT flag. This will successfully return a semaphore only if that semaphore does not already exist. Creating a semaphore requires two additional parameters: the permissions that the semaphore should be created with and the initial value for the semaphore. Following piece of code shows an example of opening a semaphore with an initial value of 10, with read and write permissions for the user, the group, and all users.

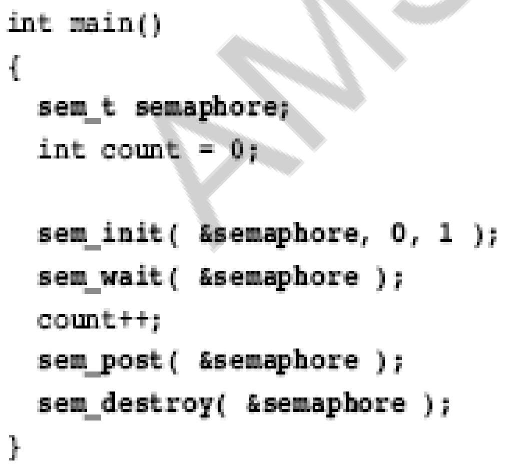


```
#include <semaphore.h>

int main()
{
  sem_t * semaphore;
  semaphore = sem_open( "/my_semaphore", O_CREAT, 0777, 10 );
  ...
   sem_close( semaphore );
   sem_unlink( "/my_semaphore" );
 }
```

A named semaphore is closed by a call to sem_close(). This closes the connection to the semaphore but leaves the semaphore present on the machine. A call to sem_unlink() with the name of the semaphore will free the resources consumed by it but only once all the processes that have the semaphore open have closed their connection to it. The code shown in Listing 5.34 will close and unlink the previously opened semaphore.

The semaphore is used through a combination of three methods. The function sem_wait() will attempt to decrement the semaphore. If the semaphore is already zero, the calling thread will wait until the semaphore becomes nonzero and then return, having decremented the semaphore. The call sem_trywait() will return immediately either having decremented the semaphore or if the semaphore is already zero. The call to sem_post() will increment the semaphore. One more call, sem_getvalue(), will write the current value of the semaphore into an integer variable. The following code shows a semaphore used in the same way as a mutex might be, to protect the increment of the variable count.



```
int main()
{
  sem_t semaphore;
  int count = 0;

  sem_init( &semaphore, 0, 1 );
  sem_wait( &semaphore );
  count++;
  sem_post( &semaphore );
  sem_destroy( &semaphore );
}
```

3. Explain about mutex lock and barriers with example?

Barriers

There are situations where a program needs to wait until an entire group of threads has completed its work before further progress can be made. This is a *barrier*. A barrier is created by a call to pthread_barrier_init(). The call to initialize the barrier takes the following:

- A pointer to the barrier to be initialized.
- An optional attributes structure, this structure determines whether the barrier is private to a process or shared across processes.
- The number of threads that need to reach the barrier before any threads are released.

The resources consumed by a barrier can be released by a call to pthread_barrier_ destroy().
Each thread calls pthread_barrier_wait() when it reaches the barrier. This call will return when the appropriate number of threads has reached the barrier. The following code demonstrates using a barrier to cause the threads in an application to wait until all the threads have been created.

```
#include <pthread.h>
#include <stdio.h>
pthread_barrier_t barrier;
void * work( void* param)
{
int id=(int)param;
printf( "Thread arrived %i\n", id );
pthread_barrier_wait( &barrier );
printf( "Thread departed %i\n", id );
}
int main()
{
pthread_t threads[10];
pthread_barrier_init( &barrier, 0, 10 );
for ( int i=0; i<10; i++ )
{
pthread_create( &threads[i], 0, work, (void*)i );
}
for ( int i=0; i<10; i++ )
{
pthread_join( threads[i], 0 );
}
pthread_barrier_destroy( &barrier );
}
```

Mutex Locks

A *mutex lock* is a mechanism supported by the POSIX standard that can be acquired by only one thread at a time. Other threads that attempt to acquire the same mutex must wait until it is released by the thread that currently has it.

Mutex Attributes

Mutexes can be shared between multiple processes. By default, mutexes are private to a process. To create a mutex that can be shared between processes, it is necessary to set up the attributes for pthread_mutex_init(), as shown in following code.

```
#include <pthread.h>
int main()
{
pthread_mutexattr_t attributes;
pthread_mutex_t mutex;
pthread_mutexattr_init( &attributes );
pthread_mutexattr_setpshared( &attributes, PTHREAD_PROCESS_SHARED );
pthread_mutex_init( &mutex, &attributes );
pthread_mutexattr_destroy( &attributes );
...
}
```

The attributes structure pthread_mutexattr_t is initialized with default values by a call to pthread_mutexattr_init(). A call to pthread_mutex_setpshared() with a pointer to the attribute structure and the value PTHREAD_PROCESS_SHARED sets the attributes to cause a shared mutex to be created. By default, mutexes are not shared between processes; calling pthread_mutex_setpshared() with the value PTHREAD_ PROCESS_PRIVATE restores the attribute to the default. These attributes are passed into the call to pthread_mutex_init() to set the attributes of the initialized mutex. Once the attributes have been used, they can be disposed of by a call to pthread_mutex_attr_destroy().

A mutex can have other attributes set using the same mechanism:

- The type of mutex. This can be a normal mutex, a mutex that detects errors such as multiple attempts to lock the mutex, or a recursive mutex that can be locked multiple times and then needs to be unlocked the same number of times.
- The protocol to follow when another thread is waiting for the mutex. This can be the default of no change to thread priority, that the thread holding the mutex inherits the priority of any higher-priority thread waiting for the mutex, or that the thread gets the highest priority associated with the mutexes held by it.
- The priority ceiling of the mutex. This is the priority that any lower-priority thread will be elevated to while it holds the mutex. The attributes governing the priority of any thread holding the mutex are designed to avoid problems of priority inversion where a higher-priority thread is waiting for a lower-priority thread to release the mutex.

4. **Explain about deadlock and live lock with proper example**?

Dead lock is 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*. Listing in following shows this situation.

```
Thread 1                                   Thread 2
void update1()                            void update2()
{                                         {
acquire(A);                                acquire(B);
acquire(B); <<< Thread 1                  acquire(A); <<< Thread 2
waits here                                 waits here
variable1++;                               variable1++;
release(B);                                release(B);
release(A);                               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 B and then eventually release both locks, allowing thread 2 to make progress. A *livelock* traps threads in an unending loop releasing and acquiring locks. Livelocks can be caused by code to back out of deadlocks. In following code, 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. 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.

```
Thread 1                      Thread 2
void update1()                 void update2()
{                              {
int done=0;                    int done=0;
while (!done)                 while (!done)
{                              {
acquire(A);                   acquire(B);
if ( canAcquire(B) )           if ( canAcquire(A) )
{                             {
variable1++;                  variable2++;
release(B);                    release(A);
release(A);                   release(B);
done=1;                       done=1;
}                             }
else                          else
{                              {
release(A);                   release(B);
}                              }
}                              }
}                             }
```

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.

5. Explain various methods for thread communication in UNIX based systems?

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. Listing in following shows the pseudocode for a producer thread adding an item onto the queue.

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.

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 and Events

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. Listing in following 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.

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. Listing in following shows code for creating and placing messages into a queue. This code is also responsible for removing the queue after use.

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 );
```

Opening a Queue and Receiving Messages

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

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. Listing in following 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.

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 );
```

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. Define openMP programming ?
   Like Pthreads, OpenMP is an API for shared-memory parallel programming. The "MP" in OpenMP stands for "multiprocessing," a term that is synonymous with shared-memory parallel computing. Thus, OpenMP is designed for systems in which each thread or process can potentially have access to all available memory, and, when we're programming with OpenMP, we view our system as a collection of cores or CPUs, all of which have access to main memory.

2. Write a procedure to run openMP program?

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

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

To run the program, we specify the number of threads on the command line.
For example, we might run the program with four threads and type

```
$ ./omp hello 4
```

3. What is SECTIONS Directive?

   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.

4. What is SINGLE Directive?

   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)

5. What is the purpose of TASK construct in openMP?

   The TASK construct defines an explicit task, which may be executed by the encountering thread, or deferred for execution by any other thread in the team.The data environment of the task is determined by the data sharing attribute clauses.

6. What is the purpose of FLUSH operation in openMP?

   Even when variables used by threads are supposed to be shared, the compiler may take liberties and optimize them as register variables. This can skew concurrent observations of the variable. The flush directive can be used to ensure that the value observed in one thread is also the value observed by other threads.

7. Write the steps to move data between threads?
   To move the value of a shared var from thread a to thread b, do the following in exactly this order:

- Write var on thread a
- Flush var on thread a
- Flush var on thread b
- Read var on thread b

8. Define task parallelism?

   Task parallelism (also known as function parallelism and control parallelism) is a form of parallelization of computer code across multiple processors in parallel computingenvironments. Task parallelism focuses on distributing tasks—concretely performed by processes or threads—across different processors. It contrasts to data parallelism as another form of parallelism.

9. What is the thread-level parallelism?

   Thread-level parallelism (TLP) is the parallelism inherent in an application that runs multiple threads at once. This type of parallelism is found largely in applications written for commercial servers such as databases. By running many threads at once, these applications are able to tolerate the high amounts of I/O and memory system latency their workloads can incur - while one thread is delayed waiting for a memory or disk access, other threads can do useful work.

10. What is the need for CRITICAL construct?

    The critical construct restricts the execution of the associated statement / block to a single thread at time. The critical construct may optionally contain a global name that identifies the type of the critical construct. No two threads can execute a critical construct of the same name at the same time.

11. Define thread safe? Give an example.

    A block of code is **thread-safe** if it can be simultaneously executed by multiple threads without causing problems. As an example, suppose we want to use multiple threads to “tokenize” a file. Let’s suppose that the file consists of ordinary English text, and that the tokens are just contiguous sequences of characters separated from the rest of the text by white space—spaces, tabs, or newlines. A simple approach to this problem is to divide the input file into lines of text and assign the lines to the threads in a round-robin fashion: the first line goes to thread 0, the second goes to thread 1, . . . , the *t*th goes to thread *t*, the *t*+1st goes to thread 0, and so on.

12. How to reduce false sharing in multi core architecture?

    - changing the structure and use of shared data into private data,

- increasing the problem size (iteration length),
- changing the mapping of iterations to processors to give each processor more work per iteration (*chunk size*),
- utilizing the compiler's optimization features to eliminate memory loads and stores

13. what is BARRIER Directive?

The BARRIER directive synchronizes all threads in the team.When a BARRIER directive is reached, a thread will wait at that point until all other threads have reached that barrier. All threads then resume executing in parallel the code that follows the barrier.

14. What are the different work sharing constructs in opemMP?

- Do-For Directive
- Section Directive
- Single Directive

15. What are the main components of openMP API?

The OpenMP API is comprised of three distinct components. They are

- Compiler Directives
- Runtime Library Routines
- Environment Variables

16. Mention the difference between parallel and parallel for directive?

The difference between parallel, parallel for and for is as follows:

- A team is the group of threads that execute currently.
    - At the program beginning, the team consists of a single thread.
    - A parallel construct splits the current thread into *a new team* of threads for the duration of the next block/statement, after which the team merges back into one.

- for divides the work of the for-loop among the threads of the *current team*. It does not create threads, it only divides the work amongst the threads of the currently executing team.
- parallel for is a shorthand for two commands at once: parallel and for. Parallel creates a new team, and for splits that team to handle different portions of the loop.

17. What are the unique features of openMP?
    - Open MP consists of set of compiler directions and library support functions.
    - It work along with programming language like Fortran, C and C++
    - It provides shared memory parallel programming
    - Special pre-processor instructions are used in C,C++ to implement open mp parallel program known as "pragma".

18. List the different directives used in OpenMP?
    - Parallel
    - Parallel For
    - Critical
    - For
    - Single
    - Sections
    - Master
    - No wait
    - Barrier
    - Atomic
    - Flush
    - Task

19. What is parallel for Directive?

    The parallel for directive instructs the system to parallelize the 'for' loop by dividing the loop iterations among the threads. It is different from parallel directive and make block partitioning of the for loop in which if there are n iteration in the for loop, it divides as 'n/threadcount' that is assigned to thread '0'and the next 'n/threadcount' that is assigned to thread 1 and so on.

20. What is parallel section directive?

    The parallel section directive is open MP make concurrent execution of the same code by multiple threads, and the syntax is as follows,

```
# pragma omp parallel sections
{
//block of code
}
```

Part B

1. **Explain about Openmp work sharing construct in detail.**

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:

NOTE: The Fortran **workshare** construct is not shown here.

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

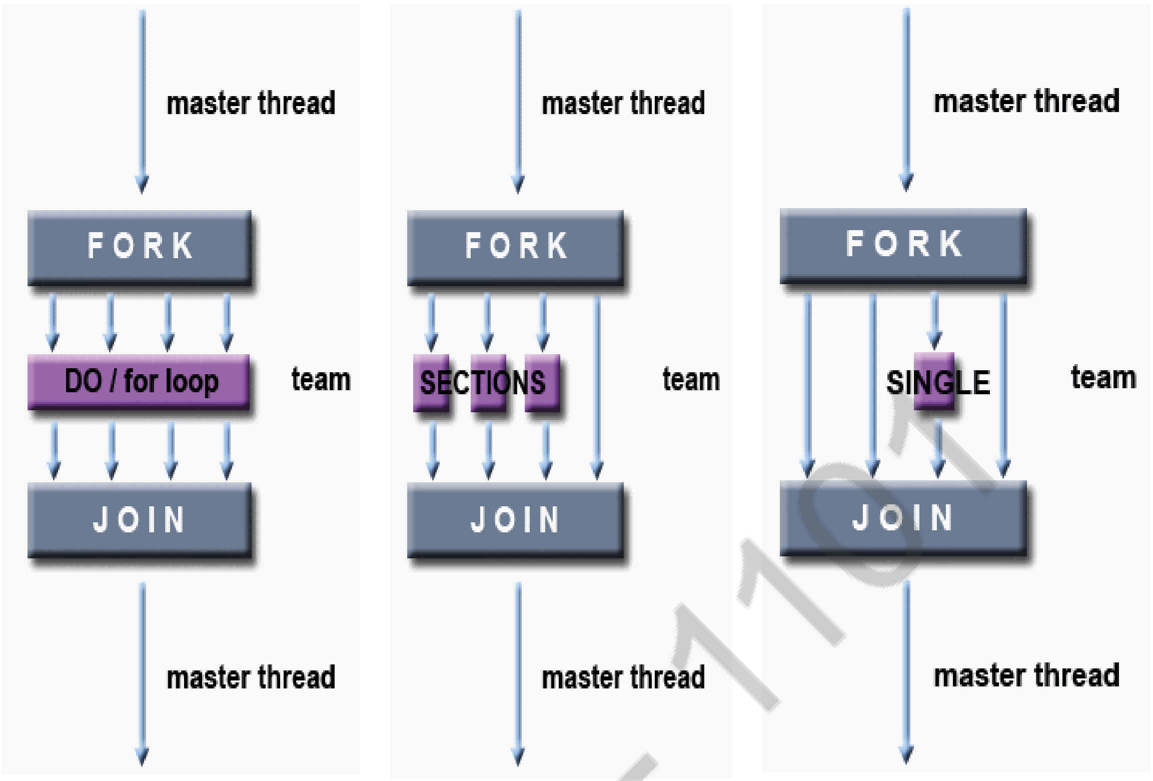


Figure 3.1 different workflow construct of openMP

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.

Format:

```
#pragma omp for [clause ...]  newline
             schedule (type [,chunk])
```

ordered
private ***(list)***
firstprivate ***(list)***
lastprivate ***(list)***
shared ***(list)***
reduction ***(operator: list)***
collapse ***(n)***
nowait

for_loop

Clauses:

- **SCHEDULE**: Describes how iterations of the loop are divided among the threads in the team. The default schedule is implementation dependent.

  STATIC

  Loop iterations are divided into pieces of size *chunk* and then statically assigned to threads. If chunk is not specified, the iterations are evenly (if possible) divided contiguously among the threads.

  DYNAMIC

  Loop iterations are divided into pieces of size *chunk*, and dynamically scheduled among the threads; when a thread finishes one chunk, it is dynamically assigned another. The default chunk size is 1.

  GUIDED

  Iterations are dynamically assigned to threads in blocks as threads request them until no blocks remain to be assigned. Similar to DYNAMIC except that the block size decreases each time a parcel of work is given to a thread. The size of the initial block is proportional to:

  number_of_iterations / number_of_threads
  Subsequent blocks are proportional to
  number_of_iterations_remaining / number_of_threads
  The chunk parameter defines the minimum block size. The default chunk size is 1.

  RUNTIME
  The scheduling decision is deferred until runtime by the environment variable OMP_SCHEDULE. It is illegal to specify a chunk size for this clause.
  AUTO

The scheduling decision is delegated to the compiler and/or runtime system.

- **NO WAIT / nowait**: If specified, then threads do not synchronize at the end of the parallel loop.
- **ORDERED**: Specifies that the iterations of the loop must be executed as they would be in a serial program.
- **COLLAPSE**: Specifies how many loops in a nested loop should be collapsed into one large iteration space and divided according to the **schedule** clause. The sequential execution of the iterations in all associated loops determines the order of the iterations in the collapsed iteration space.

Restrictions

- The DO loop can not be a DO WHILE loop, or a loop without loop control. Also, the loop iteration variable must be an integer and the loop control parameters must be the same for all threads.
- Program correctness must not depend upon which thread executes a particular iteration.
- It is illegal to branch (goto) out of a loop associated with a DO/for directive.
- The chunk size must be specified as a loop invarient integer expression, as there is no synchronization during its evaluation by different threads.
- ORDERED, COLLAPSE and SCHEDULE clauses may appear once each.
- See the OpenMP specification document for additional restrictions.

Example

```
#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.

Clauses:

- There is an implied barrier at the end of a SECTIONS directive, unless the **NOWAIT/nowait** clause is used.

Format:

```
#pragma omp sections [clause ...]  newline
                     private (list)
                     firstprivate (list)
                     lastprivate (list)
                     reduction (operator: list)
                     nowait
  {

  #pragma omp section   newline

     structured_block
  #pragma omp section   newline

     structured_block

  }
```

Restrictions:

- It is illegal to branch (goto) into or out of section blocks.
- SECTION directives must occur within the lexical extent of an enclosing SECTIONS directive (no orphan SECTIONs).

Example:

```
#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)

Format:

```
#pragma omp single [clause ...]  newline
               private (list)
               firstprivate (list)
               nowait

    structured_block
```

Clauses:

- Threads in the team that do not execute the SINGLE directive, wait at the end of the enclosed code block, unless a **NOWAIT/nowait** clause is specified.

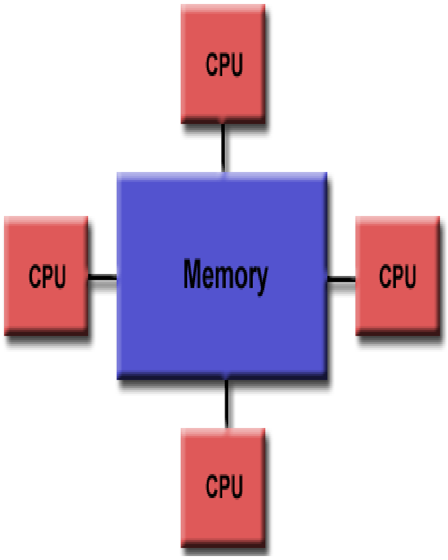
Restrictions:

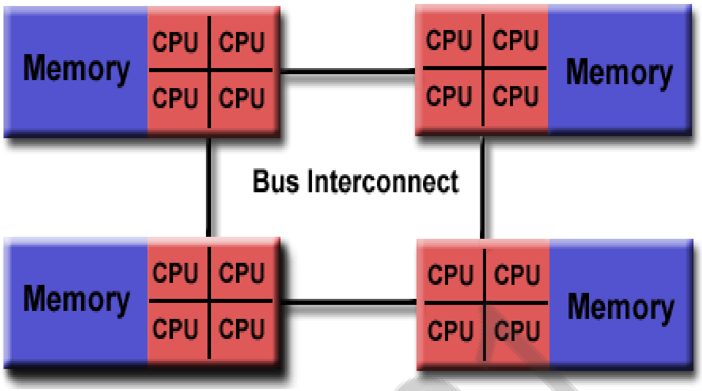
- It is illegal to branch into or out of a SINGLE block.

2. **Explain about execution model of openMP in detail**

Shared Memory Model:

OpenMP is designed for multi-processor/core, shared memory machines. The underlying architecture can be shared memory UMA or NUMA.





Uniform Memory Access **Non-Uniform Memory Access**

Thread Based Parallelism:

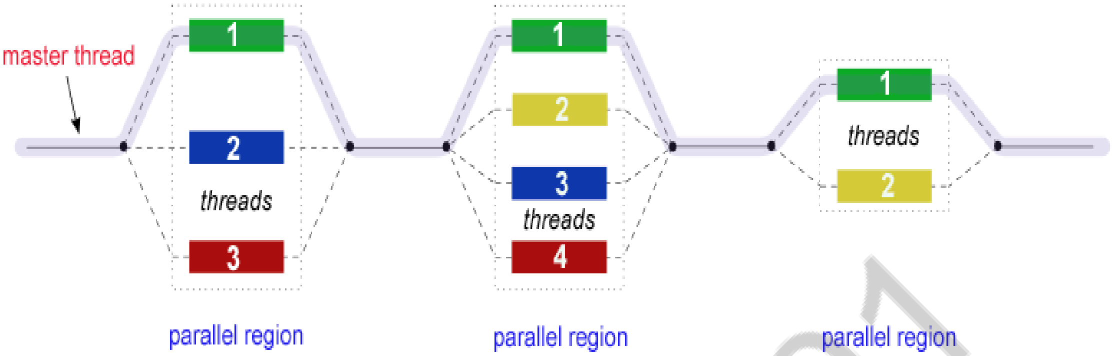
- OpenMP programs accomplish parallelism exclusively through the use of threads.
- A thread of execution is the smallest unit of processing that can be scheduled by an operating system. The idea of a subroutine that can be scheduled to run autonomously might help explain what a thread is.
- Threads exist within the resources of a single process. Without the process, they cease to exist.
- Typically, the number of threads match the number of machine processors/cores. However, the actual use of threads is up to the application.

Explicit Parallelism:

- OpenMP is an explicit (not automatic) programming model, offering the programmer full control over parallelization.
- Parallelization can be as simple as taking a serial program and inserting compiler directives....
- Or as complex as inserting subroutines to set multiple levels of parallelism, locks and even nested locks.

Fork - Join Model:

- OpenMP uses the fork-join model of parallel execution:



- All OpenMP programs begin as a single process: the **master thread**. The master thread executes sequentially until the first **parallel region** construct is encountered.
- **FORK:** the master thread then creates a team of parallel ***threads***.
- The statements in the program that are enclosed by the parallel region construct are then executed in parallel among the various team threads.
- **JOIN:** When the team threads complete the statements in the parallel region construct, they synchronize and terminate, leaving only the master thread.
- The number of parallel regions and the threads that comprise them are arbitrary.

Compiler Directive Based:

- Most OpenMP parallelism is specified through the use of compiler directives which are imbedded in C/C++ or Fortran source code.

Nested Parallelism:

- The API provides for the placement of parallel regions inside other parallel regions.
- Implementations may or may not support this feature.

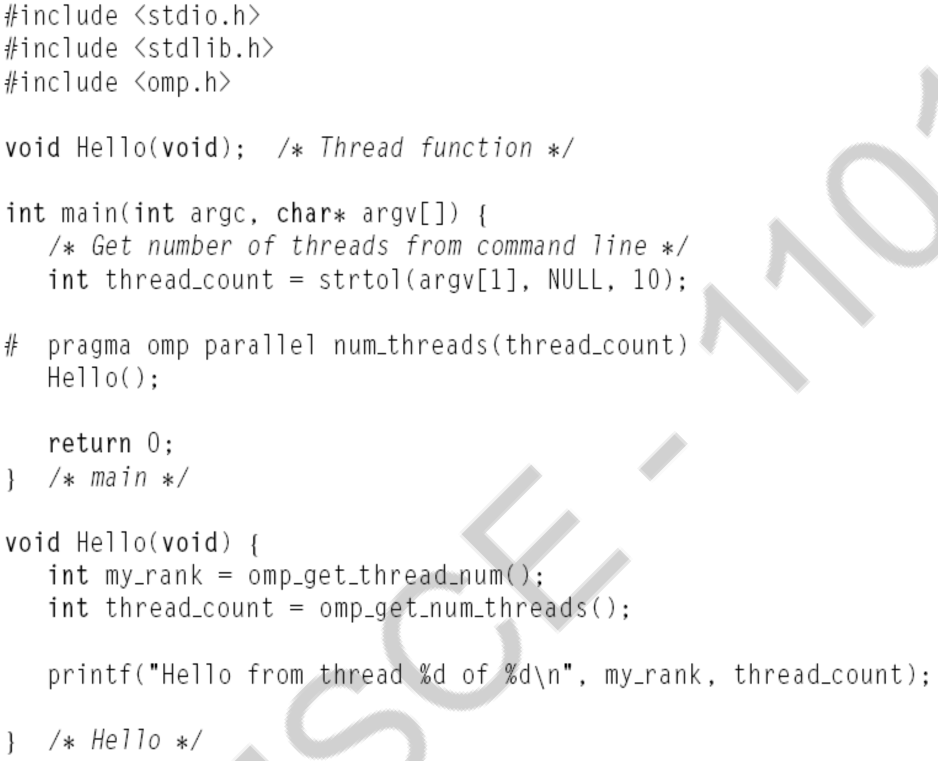
Dynamic Threads:

- The API provides for the runtime environment to dynamically alter the number of threads used to execute parallel regions. Intended to promote more efficient use of resources, if possible.
- Implementations may or may not support this feature.

I/O:

- OpenMP specifies nothing about parallel I/O. This is particularly important if multiple threads attempt to write/read from the same file.
- If every thread conducts I/O to a different file, the issues are not as significant.
- It is entirely up to the programmer to ensure that I/O is conducted correctly within the context of a multi-threaded program.

Example Program:



```
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>

void Hello(void);  /* Thread function */

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

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

   return 0;
}  /* main */

void Hello(void) {
   int my_rank = omp_get_thread_num();
   int thread_count = omp_get_num_threads();

   printf("Hello from thread %d of %d\n", my_rank, thread_count);

}  /* Hello */
```

Compiling and running OpenMP programs

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

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

To run the program, we specify the number of threads on the command line. For example, we might run the program with four threads and type

```
$ ./omp hello 4
```

3. Explain about how loop is handled in open mp with example?

The parallel for directive

OpenMP provides the parallel **for** directive for parallizing the the looping construct. Using it, we can parallelize the serial trapezoidal rule

```
h = (b-a)/n;
approx = (f(a) + f(b))/2.0;
for (i = 1; i <= n□1; i++)
approx += f(a + i_h);
approx = h_approx;
```

by simply placing a directive immediately before the **for** loop:

```
h = (b-a)/n;
approx = (f(a) + f(b))/2.0;
# pragma omp parallel for num_ threads(thread count)
reduction(+: approx)
for (i = 1; i <= n-1; i++)
approx += f(a + i*h);
approx = h*approx;
```

Like the parallel directive, the parallel **for** directive forks a team of threads to execute the following structured block. However, the structured block following the parallel **for** directive must be a **for** loop. Furthermore, with the parallel **for** directive the system parallelizes the **for** loop by dividing the iterations of the loop among the threads. The parallel **for** directive is therefore very different from the parallel directive, because in a block that is preceded by a parallel directive, in general, the work must be divided among the threads by the threads themselves.

In a **for** loop that has been parallelized with a parallel **for** directive, the default partitioning, that is, of the iterations among the threads is up to the system. However, most systems use roughly a block partitioning, that is, if there are *m* iterations, then roughly the first *m*=thread count are assigned to thread 0, the next *m*=thread count are assigned to thread 1, and so on. Note that it was essential that we made approx a reduction variable.

If we hadn’t, it would have been an ordinary shared variable, and the body of the loop approx += f(a + i*h); would be an unprotected critical section. However, speaking of scope, the default scope for all variables in a parallel directive is shared, but in our parallel **for** if the loop variable i were shared, the variable update, i++, would also be an unprotected critical section. Hence, in a loop that is parallelized with a parallel **for** directive, the default scope of the loop variable is *private*; in our code, each thread in the team has its own copy of i.

Caveats

This is truly wonderful: It may be possible to parallelize a serial program that consists of one large **for** loop by just adding a single parallel **for** directive. It may be possible to incrementally parallelize a serial program that has many **for** loops by successively placing parallel **for** directives before each loop. However, things may not be quite as rosy as they seem. There are several caveats associated with the use of the parallel **for** directive. First, OpenMP will only parallelize **for** loops. It won't parallelize **while** loops or **do-while** loops. This may not seem to be too much of a limitation, since any code that uses a **while** loop or a **do-while** loop can be converted to equivalent code that uses a **for** loop instead.

However, OpenMP will only parallelize **for** loops for which the number of iterations
can be determined

- from the **for** statement itself (that is, the code **for (. . . ; . . . ; . . .)**),and
- prior to execution of the loop.

For example, the "infinite loop"

```
for ( ; ; )
{ . . .
}
```

cannot be parallelized. Similarly, the loop

```
for (i = 0; i < n; i++) {
 if ( . . . ) break;
. . .
}
```

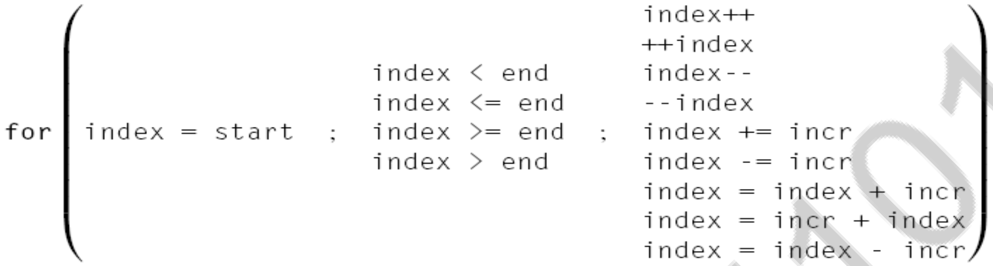
cannot be parallelized, since the number of iterations can't be determined from the **for** statement alone. This **for** loop is also not a structured block, since the **break** adds another point of exit from the loop.

In fact, OpenMP will only parallelize **for** loops that are in **canonical form**. Loops in canonical form take one of the forms shown in figure. The variables and expressions in this template are subject to some fairly obvious restrictions:

- The variable index must have integer or pointer type (e.g., it can't be a **float**).

- The expressions start, end, and incr must have a compatible type. For example,if index is a pointer, then incr must have integer type.
- The expressions start, end, and incr must not change during execution of the loop.
- During execution of the loop, the variable index can only be modified by the

"increment expression" in the **for** statement.



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

Data dependences

A more insidious problem occurs in loops in which the computation in one iteration depends on the results of one or more previous iterations. As an example, consider the following code, which computes the first *n* fibonacci numbers:

```
fibo[0] = fibo[1] = 1;
for (i = 2; i < n; i++)
fibo[i] = fibo[i-1] + fibo[i-2];
```

let's try parallelizing the **for** loop with a parallel **for** directive:

```
fibo[0] = fibo[1] = 1;
# pragma omp parallel for num threads(thread count)
for (i = 2; i < n; i++)
fibo[i] = fibo[i-1] + fibo[i-2];
```

The compiler will create an executable without complaint. However, if we try running it with more than one thread, we may find that the results are, at best, unpredictable. For example, on one of our systems if we try using two threads to compute the first 10 Fibonacci numbers, we sometimes get

1 1 2 3 5 8 13 21 34 55,

which is correct. However, we also occasionally get

1 1 2 3 5 8 0 0 0 0.

Finding loop-carried dependences

Perhaps the first thing to observe is that when we’re attempting to use a parallel **for** directive, we only need to worry about loop-carried dependences. We don’t need to worry about more general data dependences.

For example, in the loop

```
for (i = 0; i < n; i++) {
  x[i] = a + i_h;
 y[i] = exp(x[i]);
}
```

there is a data dependence between Lines 2 and 3. However, there is no problem with the parallelization

```
 # pragma omp parallel for num threads(thread count)
 for (i = 0; i < n; i++)  {
 x[i] = a + i_h;
 y[i] = exp(x[i]);
  }
```

since the computation of x[i] and its subsequent use will always be assigned to the same thread. Also observe that at least one of the statements must write or update the variable in order for the statements to represent a dependence, so in order to detect a loop carried dependence, we should only concern ourselves with variables that are updated by the loop body. That is, we should look for variables that are read or written in one iteration, and written in another.

4. **Explain about various Open MP Directives?**

   The open MP parallel shared memory programming has number of directives to support parallel programming. The open MP compiler directive in C or C++ is called “PRAGMA”. The pragma is used to communicate the information to the compiler. The information that are provided by the pragma help the compiler to optimize the program. The pragma begins with “#” character and the syntax is as follows.

   #pargma omp <pragma directive of openMP>

ATOMIC Directive

Purpose:

The ATOMIC directive specifies that a specific memory location must be updated atomically, rather than letting multiple threads attempt to write to it. In essence, this directive provides a mini-CRITICAL section.

Format:

#pragma omp atomic ***newline***

statement_expression

Restrictions:

- The directive applies only to a single, immediately following statement
- An atomic statement must follow a specific syntax. See the most recent OpenMP specs for this.

BARRIER Directive

Purpose:

- The BARRIER directive synchronizes all threads in the team.
- When a BARRIER directive is reached, a thread will wait at that point until all other threads have reached that barrier. All threads then resume executing in parallel the code that follows the barrier.

Format:

#pragma omp barrier ***newline***

Restrictions:

- All threads in a team (or none) must execute the BARRIER region.
- The sequence of work-sharing regions and barrier regions encountered must be the same for every thread in a team.

CRITICAL Directive

Purpose:

- The CRITICAL directive specifies a region of code that must be executed by only one thread at a time.

Format:

#pragma omp critical *[name] newline*

  structured_block

Restrictions:

- It is illegal to branch into or out of a CRITICAL block.

MASTER Directive

Purpose:

- The MASTER directive specifies a region that is to be executed only by the master thread of the team. All other threads on the team skip this section of code
- There is no implied barrier associated with this directive

Format:

#pragma omp master *newline*

  structured_block

Restrictions:

- It is illegal to branch into or out of MASTER block.

TASK Construct

Purpose:

- The TASK construct defines an explicit task, which may be executed by the encountering thread, or deferred for execution by any other thread in the team.
- The data environment of the task is determined by the data sharing attribute clauses.
- Task execution is subject to task scheduling

Format:

```
#pragma omp task [clause ...]  newline
                 if (scalar expression)
                 final (scalar expression)
                 untied
                 default (shared | none)
                 mergeable
                 private (list)
                 firstprivate (list)
                 shared (list)

    structured_block
```

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)

Format:

```
#pragma omp single [clause ...]  newline
                 private (list)
                 firstprivate (list)
                 nowait

    structured_block
```

Clauses:

- Threads in the team that do not execute the SINGLE directive, wait at the end of the enclosed code block, unless a **NOWAIT/nowait** clause is specified.

Restrictions:

- It is illegal to branch into or out of a SINGLE block.

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.

Format:

```
#pragma omp sections [clause ...]  newline
                 private (list)
                 firstprivate (list)
                 lastprivate (list)
                 reduction (operator: list)
                 nowait
 {

 #pragma omp section   newline

   structured_block

 #pragma omp section   newline

   structured_block

 }
```

Clauses:

- There is an implied barrier at the end of a SECTIONS directive, unless the **NOWAIT/nowait** clause is us

Restrictions:

- It is illegal to branch (goto) into or out of section blocks.
- SECTION directives must occur within the lexical extent of an enclosing SECTIONS directive (no orphan SECTIONs).

Parallel Directive

The parallel open MP directive is one which start multiple threads. It instructs the runtime system to execute the structured block of code in parallel

Parallel directive may fork or several threads to execute the structured block.

The structured block of code is one which has a single entry the exit point.

The number of threads that are generated for structured code block is system dependent.

The group of threads executing the code block is called as team.

The thread before executed before parallel thread is called master thread.

The threads that are executed by the parallel directive is called slave threads.

Format :

```
#pargma omp parallel

{

//block of code

}
```

For Directive

The for directive is used to instruct the loop scheduling and partitioning information to be instructed to the compiler.

The scheduling type can be as follows

Static

Loop iterations are divided into pieces of size *chunk* and then statically assigned to threads. If chunk is not specified, the iterations are evenly (if possible) divided contiguously among the threads.

Dynamic

Loop iterations are divided into pieces of size *chunk*, and dynamically scheduled among the threads; when a thread finishes one chunk, it is dynamically assigned another. The default chunk size is 1.

Guided

Iterations are dynamically assigned to threads in blocks as threads request them until no blocks remain to be assigned. Similar to DYNAMIC except that the block size decreases each time a parcel of work is given to a thread.

Runtime

The scheduling decision is deferred until runtime by the environment variable OMP_SCHEDULE. It is illegal to specify a chunk size for this clause.

Auto

The scheduling decision is delegated to the compiler and/or runtime system.

Format:

#pragma omp for *[clause ...] newline*

- **schedule *(type [,chunk])***
- **ordered**
- **private *(list)***
- **firstprivate *(list)***
- **lastprivate *(list)***
- **shared *(list)***
- **reduction *(operator: list)***
- **collapse *(n)***
- **nowait**

for_loop

Parallel For

The parallel for directive instructs the system to parallelize the 'for' loop by dividing the loop iterations among the threads. It is different from parallel directive and make block partitioning of the for loop in which if there are n iteration in the for loop, it divides as 'n/threadcount' that is assigned to thread '0'and the next 'n/threadcount' that is assigned to thread 1 and so on.

Format

#pragma omp Parallel For

{

//block of code

}

Flush Directive

Purpose:

- The FLUSH directive identifies a synchronization point at which the implementation must provide a consistent view of memory. Thread-visible variables are written back to memory at this point.
- Open mp explicitly make a developer to indicate the code where the variable needs to be saved to memory or loaded from memory using Flush directive.

Format

- **#pragma omp flush *(list)* *newline***

5. **Explain the various Library functions in OpenMP in detail.**

OMP_SET_NUM_THREADS

Purpose:

- Sets the number of threads that will be used in the next parallel region. Must be a postive integer.

Format:

#include <omp.h>

void omp_set_num_threads(int num_threads)

Notes & Restrictions:

- The dynamic threads mechanism modifies the effect of this routine.
    - Enabled: specifies the maximum number of threads that can be used for any parallel region by the dynamic threads mechanism.
    - Disabled: specifies exact number of threads to use until next call to this routine.
- This routine can only be called from the serial portions of the code
- This call has precedence over the OMP_NUM_THREADS environment variable

OMP_GET_NUM_THREADS

Purpose:

- Returns the number of threads that are currently in the team executing the parallel region from which it is called.

Format:

#include <omp.h>

int omp_get_num_threads(void)

Notes & Restrictions:

- If this call is made from a serial portion of the program, or a nested parallel region that is serialized, it will return 1.
- The default number of threads is implementation dependent.

OMP_GET_MAX_THREADS

Purpose:

- Returns the maximum value that can be returned by a call to the OMP_GET_NUM_THREADS function.

Format

#include <omp.h>

int omp_get_max_threads(void)

Notes & Restrictions:

- Generally reflects the number of threads as set by the OMP_NUM_THREADS environment variable or the OMP_SET_NUM_THREADS() library routine.
- May be called from both serial and parallel regions of code.

OMP_GET_THREAD_NUM

Purpose:

- Returns the thread number of the thread, within the team, making this call. This number will be between 0 and OMP_GET_NUM_THREADS-1. The master thread of the team is thread 0

Format:

```
#include <omp.h>

int omp_get_thread_num(void)
```

Notes & Restrictions:

- If called from a nested parallel region, or a serial region, this function will return 0.

OMP_GET_THREAD_LIMIT

Purpose:

- Returns the maximum number of OpenMP threads available to a program.

Format:

```
#include <omp.h>

int omp_get_thread_limit (void)
```

Notes:

- Also see the **OMP_THREAD_LIMIT** environment variable.

OMP_GET_NUM_PROCS

Purpose:

- Returns the number of processors that are available to the program.

Format:

```
#include <omp.h>

int omp_get_num_procs(void)
```

OMP_IN_PARALLEL

Purpose:

- May be called to determine if the section of code which is executing is parallel or not.

Format:

```
#include <omp.h>

int omp_in_parallel(void)
```

Notes & Restrictions:

- For C/C++, it will return a non-zero integer if parallel, and zero otherwise.

OMP_SET_DYNAMIC

Purpose:

- Enables or disables dynamic adjustment (by the run time system) of the number of threads available for execution of parallel regions.

Format:

```
#include <omp.h>
```

```
void omp_set_dynamic(int dynamic_threads)
```

Notes & Restrictions:

- For C/C++, if dynamic_threads evaluates to non-zero, then the mechanism is enabled, otherwise it is disabled.
- The OMP_SET_DYNAMIC subroutine has precedence over the OMP_DYNAMIC environment variable.
- The default setting is implementation dependent.
- Must be called from a serial section of the program.

OMP_GET_DYNAMIC

Purpose:

- Used to determine if dynamic thread adjustment is enabled or not.

Format:

```
#include <omp.h>

int omp_get_dynamic(void)
```

Notes & Restrictions:

- For C/C++, non-zero will be returned if dynamic thread adjustment is enabled, and zero otherwise.

OMP_SET_NESTED

Purpose:

- Used to enable or disable nested parallelism.

Format:

```
#include <omp.h>

void omp_set_nested(int nested)
```

Notes & Restrictions:

- For C/C++, if nested evaluates to non-zero, nested parallelism is enabled; otherwise it is disabled.
- The default is for nested parallelism to be disabled.
- This call has precedence over the OMP_NESTED environment variable

Unit 4: DISTRIBUTED MEMORY PROGRAMMING WITH MPI

Part A

1. What is the difference between distributed and shared memory architecture?
   In computer science, distributed shared memory (DSM) is a form of memory architecture where the (physically separate) memories can be addressed as one (logically shared) address space. Here, the term "shared" does not mean that there is a single centralized memory but "shared" essentially means that the address space is shared (same physical address on two processors refers to the same location in memory).

2. What is difference between MPI_INIT and MPI_FINALIZE method?

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.
MPI Finalize tells the MPI system that we’re done using MPI, and that any resources allocated for MPI can be freed.

3. What is 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.

4. What is the purpose of MPI_SEND and MPI_Recv method?
   In MPI one process can send message to other process through the MPI_send method. It has the following syntax:

```
int MPI Send(void *msg buf _p , int msg size _ in, MPI Datatype  msg type
int dest , int tag , MPI Comm communicator );
```

   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.

In MPI MPI_Recv method used by the destination process to receive the message form source . it has the following format.

int MPI Recv(**void** *msg buf p , **int** buf size , MPI Datatype buf type , **int** source , **int** tag , MPI Comm communicator , MPI _Status *status _p);

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.

5. Define trapezoidal rule?

   we can use the trapezoidal rule to approximate the area between the graph of a function, *y* = *f* (*x*), two vertical lines, and the *x*-axis. The basic idea is to divide the interval on the *x*-axis into *n* equal subintervals. Then we approximate the area lying between the graph and each subinterval by a trapezoid whose base is the subinterval, whose vertical sides are the vertical lines through the endpoints of the subinterval, and whose fourth side is the secant line joining the points where the vertical lines cross the graph.

6. Why MPI_Reduce is used in MPI communication?
   Similar to MPI_Gather, MPI_Reduce takes an array of input elements on each process and returns an array of output elements to the root process. The output elements contain the reduced result. The prototype for MPI_Reduce looks like this:

```
MPI_Reduce( void* send_data,  void* recv_data, int count, MPI_Datatype datatype,
    MPI_Op op,  int root,  MPI_Comm communicator)
```

7. Define collective communications?
   Unlike the MPI Send-MPI Recv pair, the global-sum function may involve more than
   two processes. In fact, in trapezoidal rule program it will involve all the processes
   in MPI COMM WORLD. In MPI parlance, communication functions that involve all the processes in a communicator are called **collective communications**.

8. What is point to point communication?

   Communication between two process using MPI _Send ( ) and MPI _Recv( ) methods is called as point –point communication.

9. Difference between collective and point to point communication?
   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.

10. Define broadcast in MPI?
    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

11. What is cyclic partition?
    An alternative to a block partition is a cyclic partition. In a cyclic partition, we assign the components in a round robin fashion. For an example when *n* =12 and comm._sz =3. Process 0 gets component 0, process 1 gets component 1, process 2 gets component 2, process 0 gets component 3, and so on

12. What is block-cyclic partition?
    The idea here is that instead of using a cyclic distribution of individual components, we use a cyclic distribution of *blocks* of components, so a block-cyclic distribution isn’t fully specified until we decide how large the blocks are.

13. What is MPI_Scatter function?
    MPI_Scatter is a collective routine that is very similar to MPI_Bcast. MPI_Scatterinvolves a designated root process sending data to all processes in a communicator. The primary difference between MPI_Bcast and MPI_Scatteris small but important. MPI_Bcast sends the *same* piece of data to all processes while MPI_Scatter sends *chunks of an array* to different processes.

14. Difference between MPI_Gather and MPI_AllGather?
    MPI_Gather is the inverse of MPI_Scatter. Instead of spreading elements from one process to many processes, MPI_Gather takes elements from many processes and gathers them to one single process. This routine is highly useful to many parallel algorithms, such as parallel sorting and searching.

Given a set of elements distributed across all processes, MPI_Allgather will gather all of the elements to all the processes. In the most basic sense, MPI_Allgather is an MPI_Gather followed by an MPI_Bcast. It follows many to many communication pattern.

15. How to create derived data type in MPI?

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.

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, int array of blocklengths[] , MPI __Aint array of displacements[] , MPI_ Datatype array of types[] , MPI _Datatype  *new type_ p );
```

16. What is MPI?

Programming model for distributed memory system is done through message passing interface(MPI). Message passing model is the communication system followed by the component of the distributed memory system. The two processes in the different memory core communicate with each other through message passing model.

17. Write the procedure to compile and run the MPI program?

Comilation can be done by following command

```
$mpicc  -g  -o  hellompi     hellompi.c
```

Execution of MPI program done by following command

```
$mpiexec –n  1  ./hellompi
```

18. What is MPI_Comm_Size construct in MPI?

When two process need to send and receive the message, then we need to use communicators. In MPI program, the communicator is a collection of process that can be send message to each other.

Syntax of the “MPI_Comm_Size” as follows,

Int MPI_Comm_Size(MPI_Comm comm, int* commpointer);

19. What is MPI_Comm_Rank Construct in MPI?

    Generally all the processes within the communicators are ordered. The rank of a process is the position in the communicator list in the overall order. The process can know its rank within its communicator by calling this function. The syntax is given below,

    Int MPI_Comm_Rank(MPI_Comm comm, int *myRankPointer);

20. What is MPI_Abort construct in MPI?

    This function is used to abort all process in the specified communicator and it returns error code to the calling function and its syntax as follows

    Int MPI_Abort(MPI_Comm communicator, int errorcode);

21. What is MPI_Barrier?

    The function "MPI_Barrier" is called to perform barrier synchronization among all the processes in the specified communicator and it is a collective communication function and syntax is as follows,

    Int MPI_Barrier(MPI_Comm Comm);

Part B

1. **Explain about collective or Tree based communication among process using MPI with an example?**

Collective communication

Each process with rank greater than 0 is "telling process 0 what to do" and then quitting. That is, each process with rank greater than 0 is, in effect, saying "add this number into the total." Process 0 is doing nearly all the work in computing the global sum, while the other processes are doing almost nothing. Sometimes it does happen that this is the best we can do in a parallel program, but if we imagine that we have eight students, each of whom has a number, and we want to find the sum of all eight numbers, we can certainly come up with a more equitable distribution of the work than having seven of the eight give their numbers to one of the students and having the first do the addition.

Tree-structured communication

Binary tree structured can be used for process communication which is shown(Figure 4.1) . In this diagram, 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.

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!

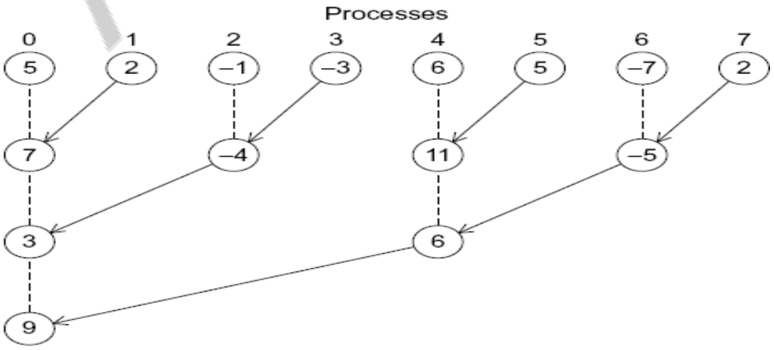
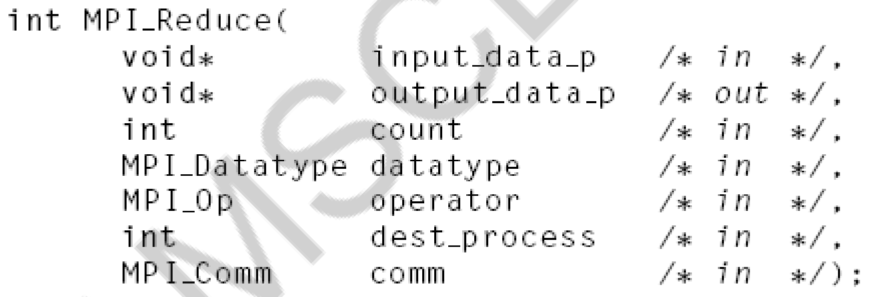


Figure 4.1: A tree structured global sum

MPI Reduce

With virtually limitless possibilities, it’s unreasonable to expect each MPI programmer to write an optimal global-sum function, so MPI specifically protects programmers against this trap of endless optimization by requiring that MPI implementations include implementations of global sums. This places the burden of optimization on the developer of the MPI implementation, rather than the application developer. The assumption here is that the developer of the MPI implementation should know enough about both the hardware and the system software so that she can make better decisions about implementation details.

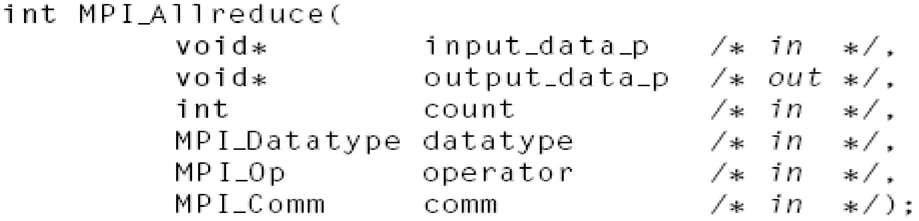
Now, a “global-sum function” will obviously require communication. However, unlike the MPI Send-MPI Recv pair, the global-sum function may involve more than two processes. To distinguish between collective communications and functions such as MPI Send and MPI Recv, MPI Send and MPI Recv are often called **point-to-point** communications. In fact, global sum is just a special case of an entire class of collective communications. For example, it might happen that instead of finding the sum of a collection of comm sz numbers distributed among the processes, we want to find the maximum or the minimum or the product or any one of many other possibilities. MPI generalized the global-sum function so that any one of these possibilities can be implemented with a single function:



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

MPI Allreduce

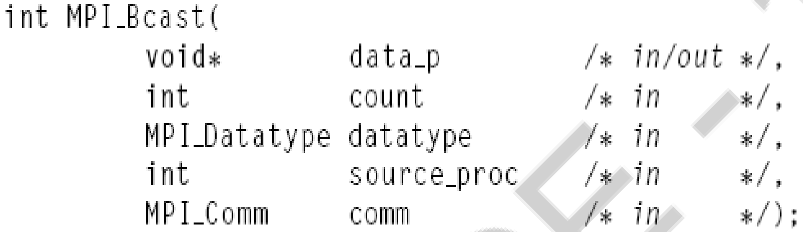
However, it’s not difficult to imagine a situation in which *all* of the processes need the result of a global sum in order to complete some larger computation. In this situation, we encounter some of the same problems we encountered with our original global sum. For example, if we use a tree to compute a global sum, we might “reverse” the branches to distribute the global sum . Alternatively, we might have the processes *exchange* partial results instead of using one-way communications. Such a communication pattern is sometimes called a **butterfly** . Once again, we don’t want to have to decide on which structure to use, or how to code it for optimal performance. Fortunately, MPI provides a variant of MPI Reduce that will store the result on all the processes in the communicator:



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

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:

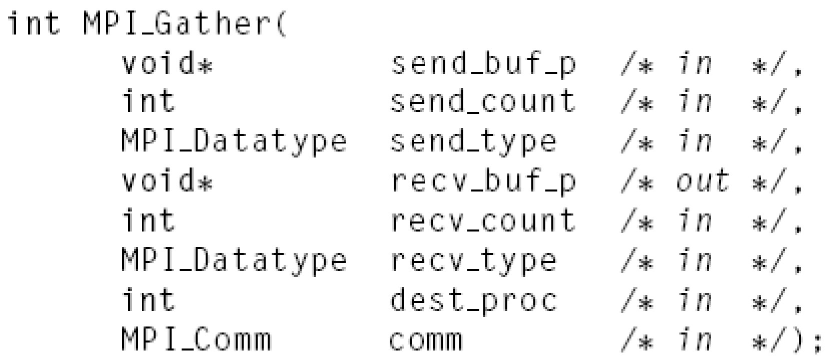


```
int MPI_Bcast(
        void*         data_p       /* in/out */,
        int           count        /* in     */,
        MPI_Datatype  datatype     /* in     */,
        int           source_proc  /* in     */,
        MPI_Comm      comm         /* in     */);
```

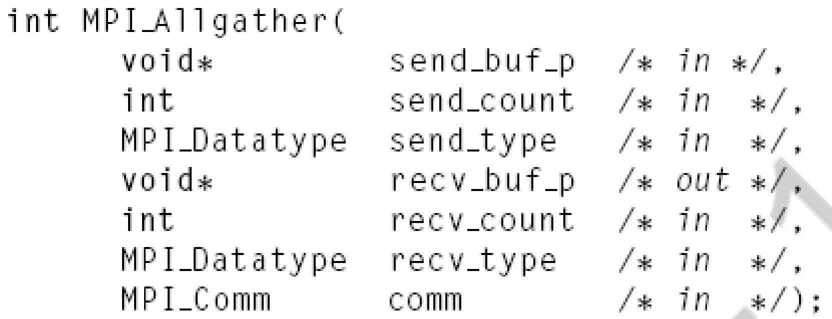
Gather and AllGeather

MPI_Gather is the inverse of MPI_Scatter. Instead of spreading elements from one process to many processes, MPI_Gather takes elements from many processes and gathers them to one single process. This routine is highly useful to many parallel algorithms, such as parallel sorting and searching.

Given a set of elements distributed across all processes, MPI_Allgather will gather all of the elements to all the processes. In the most basic sense, MPI_Allgather is an MPI_Gather followed by an MPI_Bcast. It follows many to many communication pattern.



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



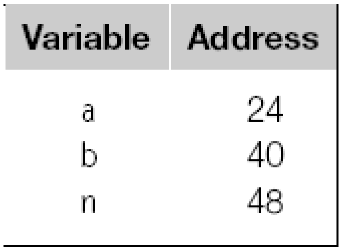
```
int MPI_Allgather(
      void*          send_buf_p  /* in */,
      int            send_count  /* in  */,
      MPI_Datatype   send_type   /* in  */,
      void*          recv_buf_p  /* out */,
      int            recv_count  /* in  */,
      MPI_Datatype   recv_type   /* in  */,
      MPI_Comm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

2. **Explain the derived data types creation in MPI programs?**

MPI DERIVED DATATYPES

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, 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*.

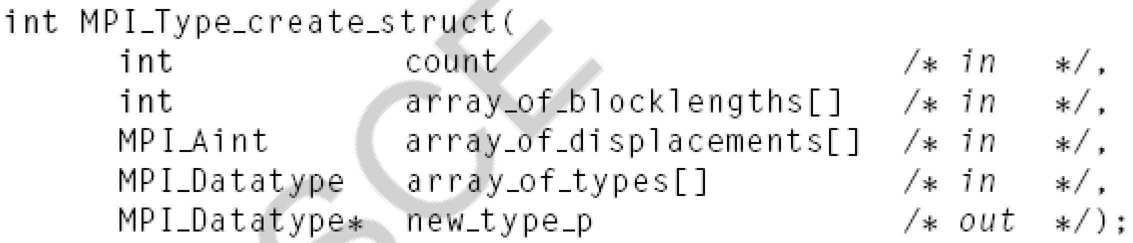
As an alternative, we could build a single derived datatype that consists of two **double**s and one **int**. If we do this, we'll only need one call to MPI Bcast. On process 0, *a*,*b*, and *n* will be sent with the one call, while on the other processes, the values will be received with the call. Formally, a derived datatype consists of a sequence of basic MPI datatypes together with a *displacement* for each of the datatypes. In trapezoidal rule, 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:

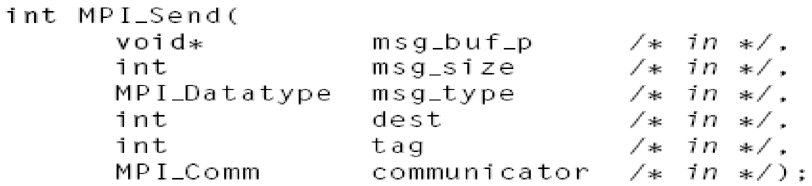


```
int MPI_Type_create_struct(
      int             count                    /* in   */,
      int             array_of_blocklengths[]  /* in   */,
      MPI_Aint        array_of_displacements[] /* in   */,
      MPI_Datatype    array_of_types[]         /* in   */,
      MPI_Datatype*   new_type_p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

3. Explain about MPI send and receive process with an example?

MPI Send

Each process can send message to other by calling MPI_Send method. The syntax of this method is given below,

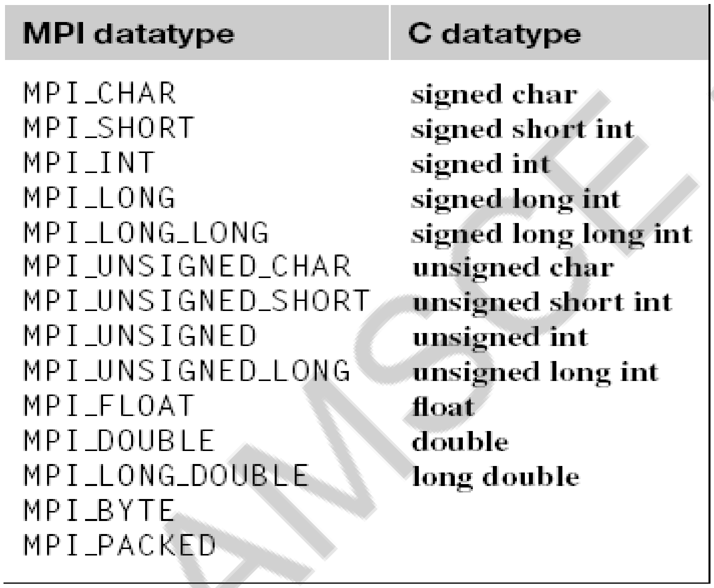


```
int MPI_Send(
      void*          msg_buf_p     /* in */,
      int            msg_size      /* in */,
      MPI_Datatype   msg_type      /* in */,
      int            dest          /* in */,
      int            tag           /* in */,
      MPI_Comm       communicator  /* in */);
```

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. 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 '/0' 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 **char**s. 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.

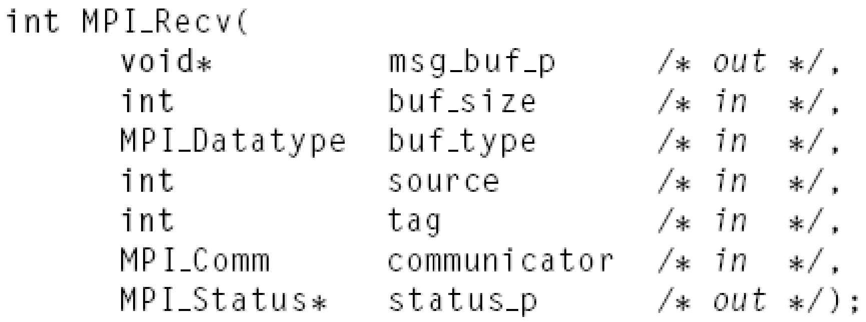
MPI also defines a number of constant values for this type. Which is shown in following figure.



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	

MPI Recv

A process can receive message from source by calling MPI_Recv method. The syntrax of this method is given below,



```
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.

Message matching

Suppose process *q* calls MPI Send with
MPI_Send(send _buf _p, send _buf _sz, send _type, dest, send _tag, Send_ comm);
Also suppose that process *r* calls MPI Recv with
MPI _Recv(recv_buf _p, recv _buf _sz, recv _type, src, recv_ tag,
recv _comm, &status);
Then the message sent by *q* with the above call to MPI Send can be received by *r* with the call to MPI_Recv if

- recv _comm = send _comm,
- recv _tag = send _tag,
- dest = r,
- src = q.

These conditions aren't quite enough for the message to be *successfully* received, however. The parameters specified by the first three pairs of arguments, send _buf _p/recv _buf _p, send_ buf _sz/recv _buf _sz, and send_ type/recv _type, must specify compatible buffers.

4. Explain about performance evaluation of MPI programs with example?

Speedup and efficiency

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**. In practice, we're unlikely to get linear speedup because the use of multiple processes/threads almost invariably introduces some overhead. 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

$$S = T\text{serial} / T\text{parallel}$$

then linear speedup has *S* D *p*, which is unusual. Furthermore, as *p* increases, we expect *S* to become a smaller and smaller fraction of the ideal, linear speedup *p*. *S*=*p*, is sometimes called the **efficiency** of the parallel program. If we substitute the formula for *S*, we see that the efficiency is

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

Amdahl's law

Back in the 1960s, Gene Amdahl made an observation that's become known as **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

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

and the speedup will be

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

Scalability

A technology is scalable if it can handle ever-increasing problem sizes. However, in discussions of parallel program performance, scalability has a somewhat more formal definition. Suppose we run a parallel program with a fixed number of processes/threads and a fixed input size, and we obtain 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

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

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*:

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

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

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

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.

5. Explain about various MPI Library functions in detail.

MPI_Comm_Size

When two process need to send and receive the message, then we need to use communicators. In MPI program, the communicator is a collection of process that can be send message to each other.

Syntax of the "MPI_Comm_Size" as follows,

Int MPI_Comm_Size(MPI_Comm comm, int* commpointer);

MPI_Comm_Rank

Generally all the processes within the communicators are ordered. The rank of a process is the position in the communicator list in the overall order. The process can know its rank within its communicator by calling this function. The syntax is given below,

Int MPI_Comm_Rank(MPI_Comm comm, int *myRankPointer);

MPI_Abort

This function is used to abort all process in the specified communicator and it returns error code to the calling function and its syntax as follows

Int MPI_Abort(MPI_Comm communicator, int errorcode);

MPI_Barrier

The function "MPI_Barrier" is called to perform barrier synchronization among all the processes in the specified communicator and it is a collective communication function and syntax is as follows,

Int MPI_Barrier(MPI_Comm Comm);

MPI_Reduce

Similar to MPI_Gather, MPI_Reduce takes an array of input elements on each process and returns an array of output elements to the root process. The output elements contain the reduced result. The prototype for MPI_Reduce looks like this:

```
MPI_Reduce( void* send_data,  void* recv_data, int count, MPI_Datatype datatype,
 MPI_Op op,  int root,  MPI_Comm communicator)
```

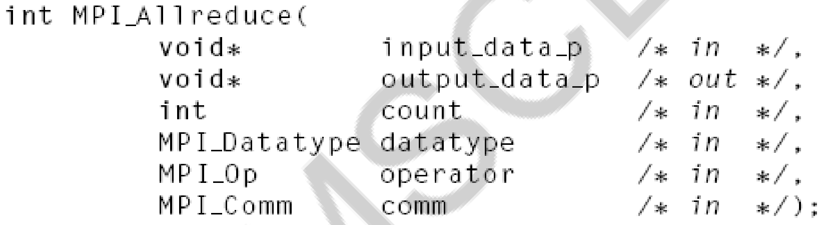
MPI_Address

This function returns the byte address of location in array offset and its syntax is as follows

Int MPI_Address(void * location, MPI_A int * offset);

MPI_Allgether

MPI_Allgather will gather all of the elements to all the processes. In the most basic sense, MPI_Allgather is an MPI_Gather followed by an MPI_Bcast. It follows many to many communication pattern. The format of this method is as follows,

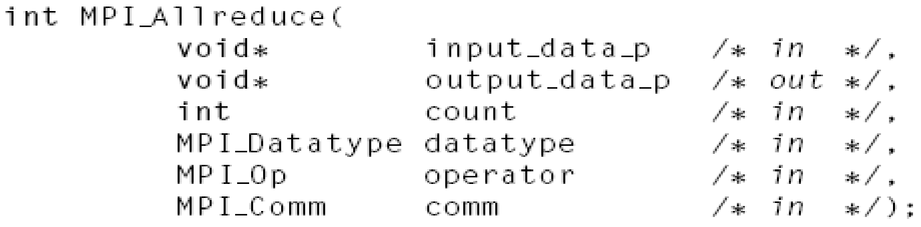


```
int MPI_Allreduce(
        void*         input_data_p   /* in  */,
        void*         output_data_p  /* out */,
        int           count          /* in  */,
        MPI_Datatype  datatype       /* in  */,
        MPI_Op        operator       /* in  */,
        MPI_Comm      comm           /*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

MPI_Allreduce

This function is called to perform count reduction and it is collective communication function.

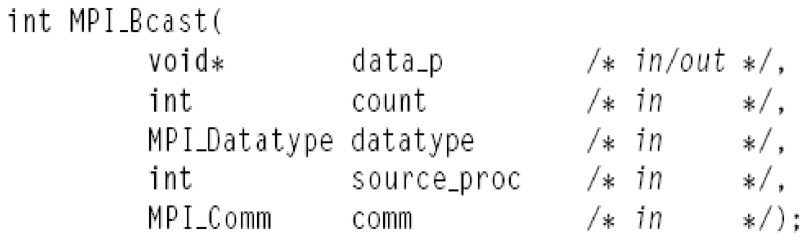
When this function returns, all the process have the results of the reductions and syntax is as follows,



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

MPI_Bcast

This function is called to allow one process to broadcast a message to all other process in the communicator and its syntax is as follows,



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

MPI_Get_processor_name

Returns the processor name. Also returns the length of the name. The buffer for "name" must be at least MPI_MAX_PROCESSOR_NAME characters in size. What is returned into "name" is implementation dependent - may not be the same as the output of the "hostname" or "host" shell commands. And syntax is given as follow,

MPI_Get_processor_name (&name,&resultlength)

MPI_Initialized

Indicates whether MPI_Init has been called - returns flag as either logical true (1) or false(0). MPI requires that MPI_Init be called once and only once by each process. This may pose a problem for modules that want to use MPI and are prepared to call MPI_Init if necessary. MPI_Initialized solves this problem. And syntax is given as follow,

MPI_Initialized (&flag)

MPI_Finalize

Terminates the MPI execution environment. This function should be the last MPI routine called in every MPI program - no other MPI routines may be called after it. And syntax is given as follow,

MPI_Finalize ()

MPI_Scatter

Data movement operation. Distributes distinct messages from a single source task to each task in the group. And syntax is given as follow,

**MPI_Scatter(&sendbuf,sendcnt,sendtype,&recvbuf,
recvcnt,recvtype,root,comm)**

MPI_Gather

Data movement operation. Gathers distinct messages from each task in the group to a single destination task. This routine is the reverse operation of MPI_Scatter.

**MPI_Gather(&sendbuf,sendcnt,sendtype,&recvbuf,
recvcount,recvtype,root,comm)**

Unit 5 : PARALLEL PROGRAM DEVELOPMENT

Part A

1. Define n-Body solver?
   An *n*-body solver is a program that finds the solution to an *n*-body problem by simulating the behavior of the particles. 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 ring pass?

   In a ring pass, we imagine the processes as being interconnected in a ring shown in figure. Process 0 communicates directly with processes 1 and comm._sz-1, process 1 communicates with processes 0 and 2, and so on. The communication in a ring pass takes place in phases, and during each phase each process sends data to its "lower-ranked" neighbor, and receives data from its "higher-ranked" neighbor. Thus, 0 will send to comm._sz-1 and receive from 1. 1 will send to 0 and receive from 2, and so on.

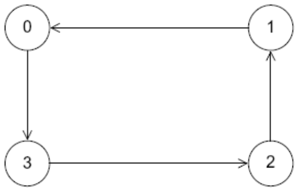


FIG 1: process in ring structure

3. What is the problem in 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.

4. Define non-recursive 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.

5. What is dynamic mapping of tasks?
   In a dynamic scheme, if one thread/process runs out of useful work, it can obtain additional work from another thread/process.

6. What is static parallelization in pthread?
   In static parallelization, a single thread uses breadth-first search to generate enough partial tours so that each thread gets at least one partial tour. Then each thread takes its partial tours and runs iterative tree search on them.

7. What is dynamic parallelization in pthread?
   when a thread runs out of work instead of immediately exiting the while loop, the thread waits to see if another thread can provide more work. On the other hand, if a thread that still has work in its stack finds that there is at least one thread without work, and its stack has at least two tours, it can "split" its stack and provide work for one of the threads.

8. What is travelling sales man problem?
   Given a set of cities and distance between every pair of cities, the problem is to find the shortest possible route that visits every city exactly once and returns to the starting point.

9. What are the different modes for send operations in MPI?
   MPI provides four modes for sends:
   - standard
   - synchronous
   - ready
   - buffered.

10. What is non-blocking send in MPI?

A non-blocking send command really means a send start. The function call returns as soon as possible, i.e., as soon as other parts of the operating system and/or hardware can take over. But the send process itself may not be complete still for a long time. A separate send complete call is necessary to verify that the data has been copied out of the send buffer and that the buffer can now be used for another communication.

11. Define Traffic Circle system?

Traffic circle system is a method of handling traffic at an intersection without using signal lights. Many countries use the provision of concurrent movement of multiple cars in the same direction. The traffic feeds from four roads on four direction .

- Every vehicle moves around the circle in a counter clockwise direction.
- Totally there are 16 section in the traffic circle system.
- During the single step, all vehicles inside the circle move to the next section in the counter clockwise direction

12. Define Neutron Transport system?

The source medium emits neutrons against homogenous medium with high plate thickness with infinite height. The neutrons that are emitted in the homogenous medium may be of following type,

Reflected Neutrons Type
Absorbed Neutrons Type
Transmitted Neutrons Type

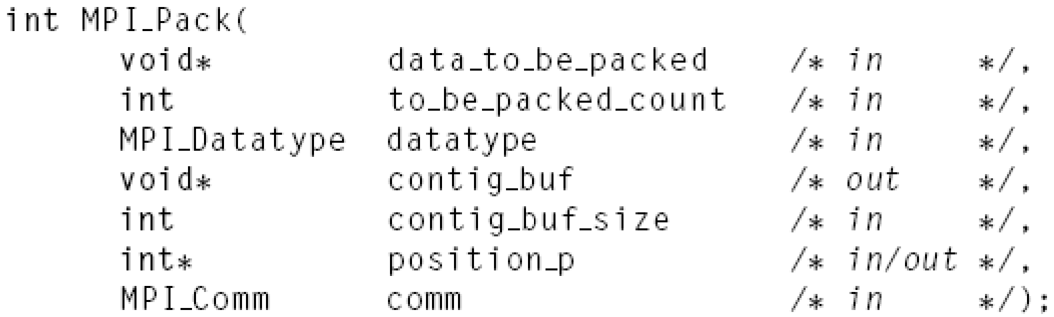
We need to compute the frequency at which each of these events occurs as a function of plate thickness.

13. Define Room Assignment Problem?

With the given n value as even number of students, our aim is to assign them to "n/2" rooms in the conference hall residence to minimize the assignment time. A complete survey has been made and the table is created in which (i,j) of the table denotes the extent to which student "i" and "j" are likely to get on each order. The entry [i,j] in the table is equal to the entry [j,i] and this problem is solved using simulated annealing method.

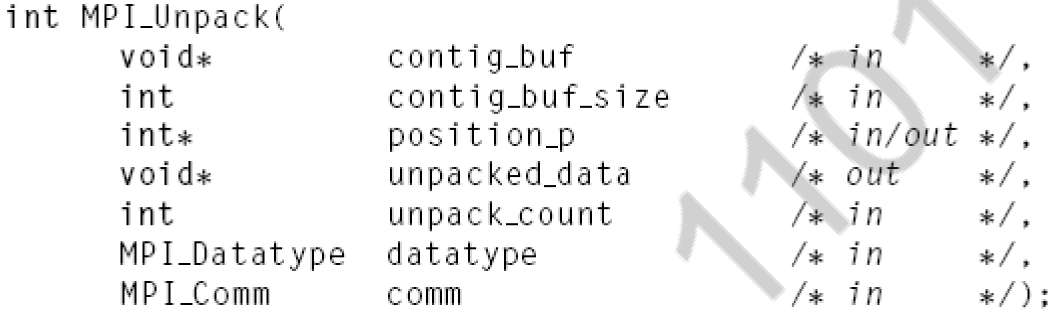
14. Give the format of MPI_Pack method in MPI communication?

The format of MPI_Pack method is given below,



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

15. Give the format of MPI_Pack method in MPI communication?
The format of MPI_Pack method is given below,



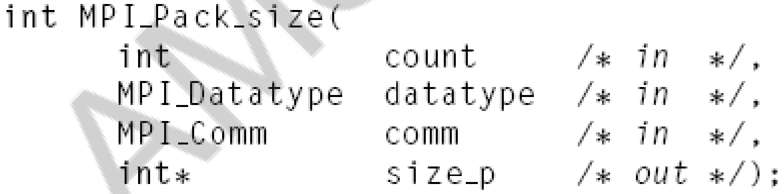
```
int MPI_Unpack(
      void*          contig_buf           /* in     */,
      int            contig_buf_size      /* in     */,
      int*           position_p           /* in/out */,
      void*          unpacked_data        /* out    */,
      int            unpack_count         /* in     */,
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

16. What is the need of split_stack in MPI?

The MPI version of Split stack *packs* the contents of the new stack into contiguous memory and sends the block of contiguous memory, which is *unpacked* by the receiver into a new stack.

17. Give the format of MPI_Pack_Size construct in MPI?

The amount of storage that's needed for the *data* that's transmitted can be determined with a call to MPI_Pack_size:



```
int MPI_Pack_size(
      int            count      /* in  */,
      MPI_Datatype   datatype   /* in  */,
      MPI_Comm       comm       /* in  */,
      int*           size_p     /* out */);
```

18. Define NP-complete problem?

They are a subset of NP problems with the property that all other NP problems can be reduced to any of them in polynomial time. So, they are the hardest problems in NP, in terms of running time. If it can be showed that any NPC Problem is in P, then all problems in NP will be in P , and hence P=NP=NPC.

19. What is NP-hard (NPH) problem?

These problems need not have any bound on their running time. If any NPC Problem is polynomial time reducible to a problem XX, that problem XX belongs to NP Hard class. Hence, all NP Complete problems are also NPH. In other words if a NPH problem is non-deterministic polynomial time solvable, it is a NPC problem. Example of a NP problem that is not NPC is Halting Problem.

20. Give the pseudo code of n-body solver problem?

```
Get input data;
 for each timestep {
 if (timestep output)
Print positions and velocities of particles;
 for each particle q
Compute total force on q;
 for each particle q
Compute position and velocity of q;
}
Print positions and velocities of particles;
```

Part B

1. **Explain briefly about n-Body solver problem with example?**

TWO *n*-BODY SOLVERS

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

The problem

For the sake of explicitness, let's write an *n*-body solver that simulates the motions of planets or stars.We'll use Newton's second law of motion and his law of universal gravitation to determine the positions and velocities. Thus, if particle *q* has position **s***q*(t) at time *t*, and particle *k* has position **s***k*(t), then the force on particle *q* exerted by particle *k* is given by

$$\mathbf{f}_{qk}(t) = -\frac{Gm_q m_k}{\left|\mathbf{s}_q(t) - \mathbf{s}_k(t)\right|^3}\left[\mathbf{s}_q(t) - \mathbf{s}_k(t)\right].$$

Here, *G* is the gravitational constant and *mq* and *mk* are the masses of particles *q* and *k*, respectively. Also, the notation |**s***q*(*t*)-**s***k*(*t)*| represents the distance from particle *k* to particle *q*. If our *n* particles are numbered 0, 1, 2, : : : ,*n*-1, then the total force on particle *q* is given by

$$\mathbf{F}_q(t) = \sum_{\substack{k=0\\k\neq q}}^{n-1} \mathbf{f}_{qk} = -Gm_q \sum_{\substack{k=0\\k\neq q}}^{n-1} \frac{m_k}{\left|\mathbf{s}_q(t) - \mathbf{s}_k(t)\right|^3}\left[\mathbf{s}_q(t) - \mathbf{s}_k(t)\right].$$

the acceleration of an object is given by the second derivative of its position and that Newton’s second law of motion states that the force on an object is given by its mass multiplied by its acceleration, so if the acceleration of particle *q* is **a***q*(t), then **F***q*(t) =*mq***a***q*(*t*) = *mq***s**”*q*(*t*), where **s**”*q*(*t*) is the second derivative of the position **s***q*(*t*). Thus, we can use above Formula to find the acceleration of particle *q*:

$$\mathbf{s}_q''(t) = -G\sum_{\substack{j=0\\j\neq q}}^{n-1} \frac{m_j}{\left|\mathbf{s}_q(t) - \mathbf{s}_j(t)\right|^3}\left[\mathbf{s}_q(t) - \mathbf{s}_j(t)\right].$$

a serial *n*-body solver can be based on the following pseudocode:

```
Get input data;
for each timestep {
   if (timestep output) Print positions and velocities of particles;
   for each particle q
      Compute total force on q;
   for each particle q
      Compute position and velocity of q;
}
Print positions and velocities of particles;
```

2. Explain about recursive and non recursive depth first search in detail with example?

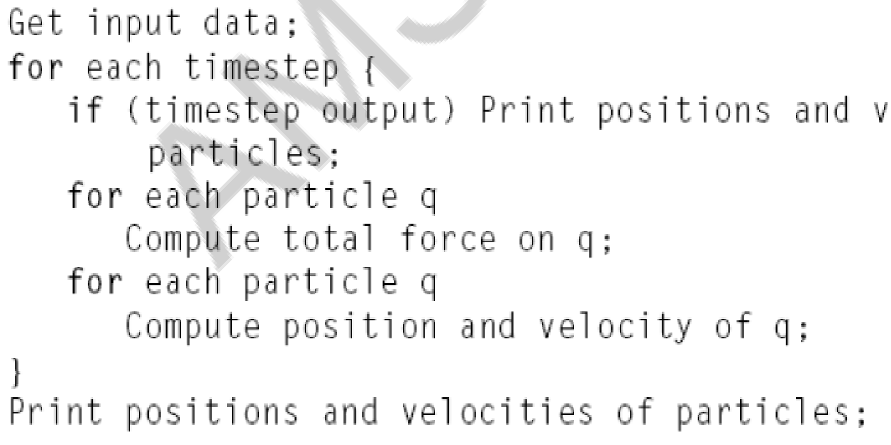
Recursive depth-first search

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.

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

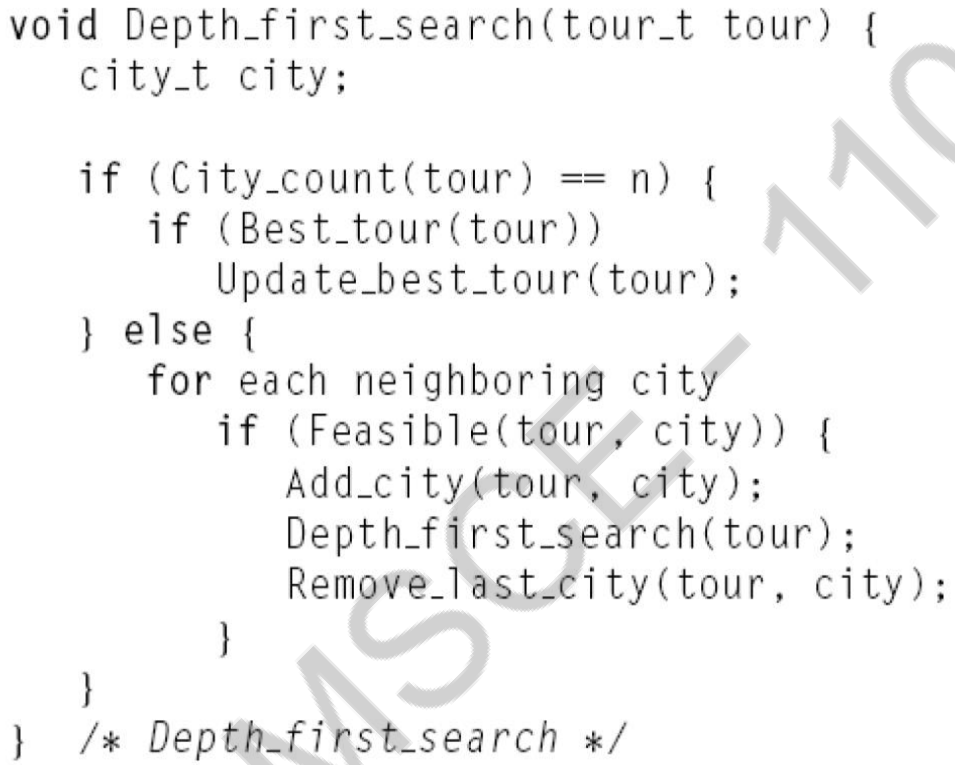
The function City_count examines the partial tour tour to see if there are *n* cities on the partial tour. If there are, we know that we simply need to return to the hometown to complete the tour, and we can check to see if the complete tour has a lower cost than the current "best tour" by calling Best _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.



```
Get input data;
for each timestep {
   if (timestep output) Print positions and velocities of
         particles;
   for each particle q
      Compute total force on q;
   for each particle q
      Compute position and velocity of q;
}
Print positions and velocities of particles;
```

Nonrecursive 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.



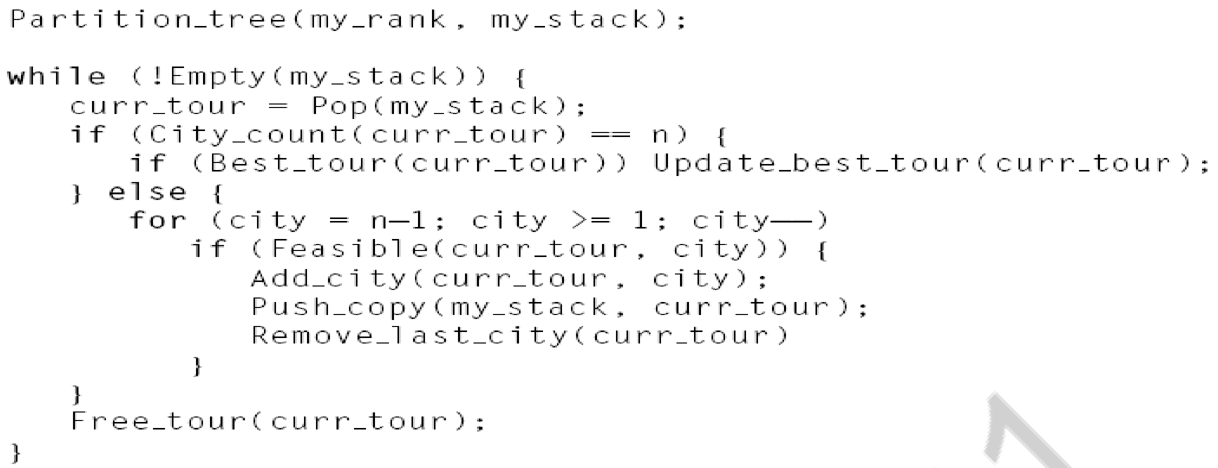
```
void Depth_first_search(tour_t tour) {
   city_t city;

   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 */
```

3. **Explain about different methods for parallelizing the tree search using pthread?**

A static parallelization of tree search using pthreads

In our static parallelization, a single thread uses breadth-first search to generate enough partial tours so that each thread gets at least one partial tour. Then each thread takes its partial tours and runs iterative tree search on them. We can use the pseudocode shown in following Program on each thread.

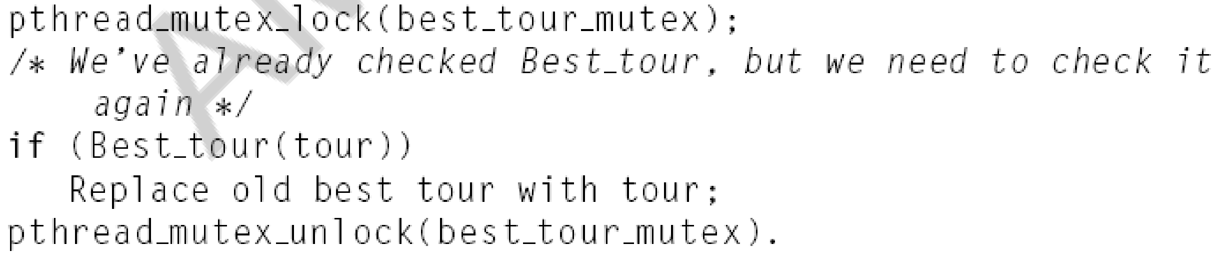


```
Partition_tree(my_rank, my_stack);

while (!Empty(my_stack)) {
   curr_tour = Pop(my_stack);
   if (City_count(curr_tour) == n) {
      if (Best_tour(curr_tour)) Update_best_tour(curr_tour);
   } else {
      for (city = n−1; city >= 1; city−−)
         if (Feasible(curr_tour, city)) {
            Add_city(curr_tour, city);
            Push_copy(my_stack, curr_tour);
            Remove_last_city(curr_tour)
         }
   }
   Free_tour(curr_tour);
}
```

To implement the Best _tour function, a thread should compare the cost of its current tour with the cost of the global best tour. Since multiple threads may be simultaneously accessing the global best cost, it might at first seem that there will be a race condition. However, the Best _tour function only *reads* the global best cost, so there won't be any conflict with threads that are also checking the best cost. If a thread is updating the global best cost, then a thread that is just checking it will either read the old value or the new, updated value. While we would prefer that it get the new value, we can't insure this without using some very costly locking strategy. For example, threads wanting to execute Best _tour or Update _best _tour could wait on a single mutex.

On the other hand, we call Update _best _tour with the intention of *writing* to the best tour structure, and this clearly can cause a race condition if two threads call it simultaneously. To avoid this problem, we can protect the body of the Update _best _tour function with a mutex. Thus, correct pseudocode for Update_ best_ tour should look something like this:



```
pthread_mutex_lock(best_tour_mutex);
/* We've already checked Best_tour, but we need to check it
     again */
if (Best_tour(tour))
   Replace old best tour with tour;
pthread_mutex_unlock(best_tour_mutex).
```

A dynamic parallelization of tree search using pthreads

If the initial distribution of subtrees doesn't do a good job of distributing the work among the threads, the static parallelization provides no means of redistributing work. The threads with "small" subtrees will finish early, while the threads with large subtrees will continue to work. It's not difficult to imagine that one thread gets the lion's share of the work because the edges in its initial tours are very cheap, while the edges in the other threads' initial tours are very expensive.

To address this issue, we can try to dynamically redistribute the work as the computation proceeds. To do this, we can replace the test !Empty(my_ stack) controlling execution of the **while** loop with more complex code. The basic idea is that when a thread runs out of work—that is, !Empty(my stack) becomes false—instead of immediately exiting the **while** loop, the thread waits to see if another thread can provide more work.

On the other hand, if a thread that still has work in its stack finds that there is at least one thread without work, and its stack has at least two tours, it can "split" its stack and provide work for one of the threads. Pthreads condition variables provide a natural way to implement this. When a thread runs out of work it can call pthread _cond_ wait and go to sleep. When a thread with work finds that there is at least one thread waiting for work, after splitting its stack, it can call pthread _cond _signal.

When a thread is awakened it can take one of the halves of the split stack and return to work. This idea can be extended to handle termination. If we maintain a count of the number of threads that are in pthread cond wait, then when a thread whose stack is empty finds that thread_ count-1 threads are already waiting, it can call pthread _cond _broadcast and as the threads awaken, they'll see that all the threads have run out of work and quit.

4. **Explain about parallelizing the tree search using open MP?**

Parallelizing the tree-search programs using OpenMP

The issues involved in implementing the static and dynamic parallel tree-search programs using OpenMP are the same as the issues involved in implementing the programs using Pthreads. There are almost no substantive differences between a static implementation that uses OpenMP and one that uses Pthreads. However, a couple of points should be mentioned:

1. When a single thread executes some code in the Pthreads version, the test **if** (my rank == whatever) can be replaced by the OpenMP directive

# pragma omp single

This will insure that the following structured block of code will be executed by one thread in the team, and the other threads in the team will wait in an implicit barrier at the end of the block until the executing thread is finished. When whatever is 0 (as it is in each test in the Pthreads program), the test can also be replaced by the OpenMP directive

# pragma omp master

This will insure that thread 0 executes the following structured block of code. However, the master directive doesn't put an implicit barrier at the end of the block, so it may be necessary to also add a barrier directive after a structured block that has been modified by a master directive.

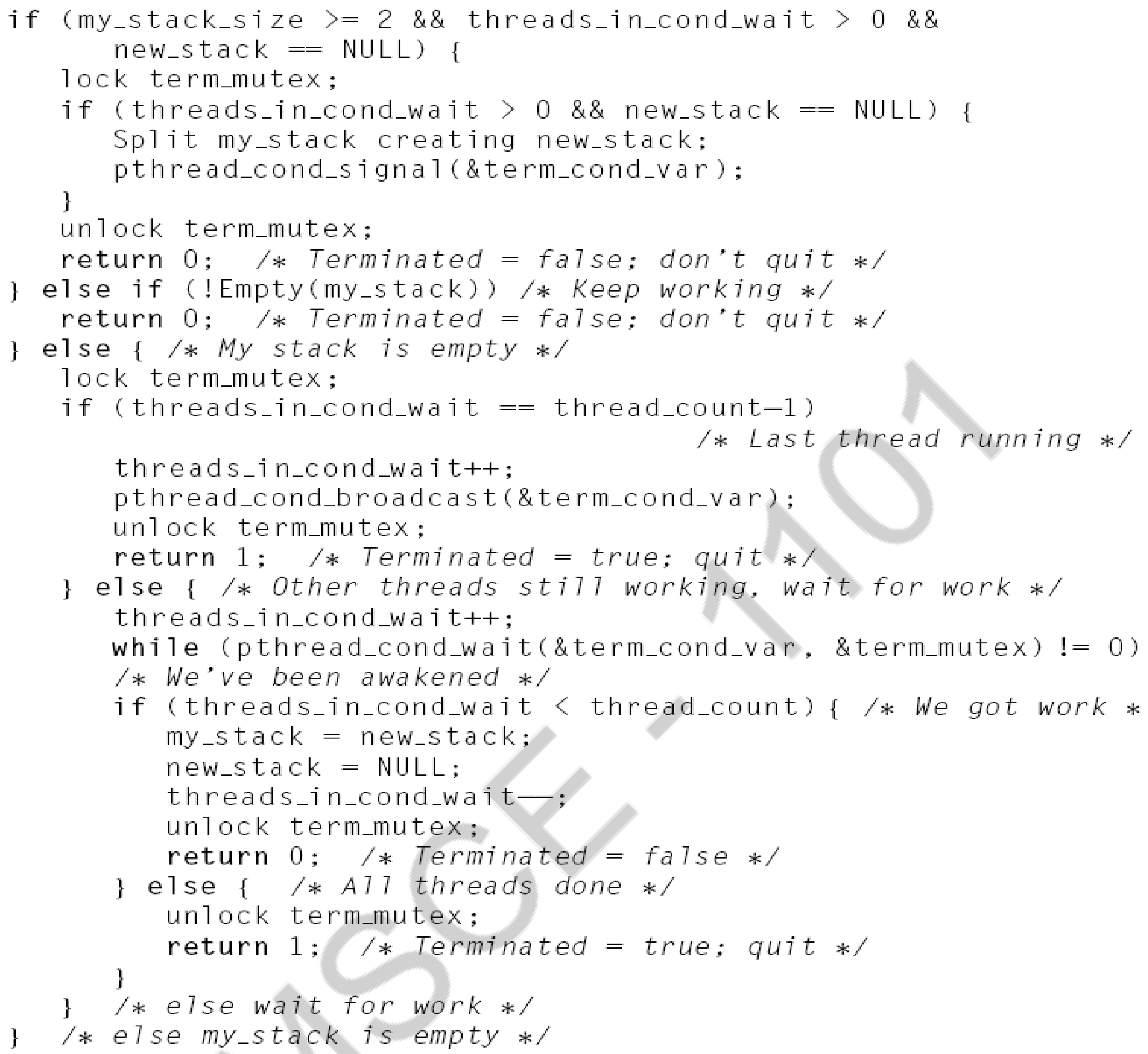
2. The Pthreads mutex that protects the best tour can be replaced by a single critical directive placed either inside the Update best tour function or immediately before the call to Update_ best _tour. This is the only potential source of a race condition after the distribution of the initial tours, so the simple critical directive won't cause a thread to block unnecessarily.

OpenMP provides a lock object omp _lock _t and the following functions for acquiring and relinquishing the lock, respectively:

void omp _set_ lock(omp _lock_ t* lock _p /* in/out */);
void omp _unset _lock(omp _lock _t* lock_ p /* in/out */);

It also provides the function
Int omp _test _lock(omp _lock _t* lock_ p /* in/out */);

which is analogous to pthread _mutex _trylock; it attempts to acquire the lock *lock _p, and if it succeeds it returns true (or nonzero). If the lock is being used by some other thread, it returns immediately with return value false (or zero).
If we examine the pseudocode for the Pthreads Terminated function in following Program. we see that in order to adapt the Pthreads version to OpenMP, we need to emulate the functionality of the Pthreads function calls in Lines 6, 17, and 22, respectively.
pthread _cond _signal(&term _cond _var);
pthread_ cond_ broadcast(&term _cond _var);
pthread _cond _wait(&term cond _var, &term _mutex);

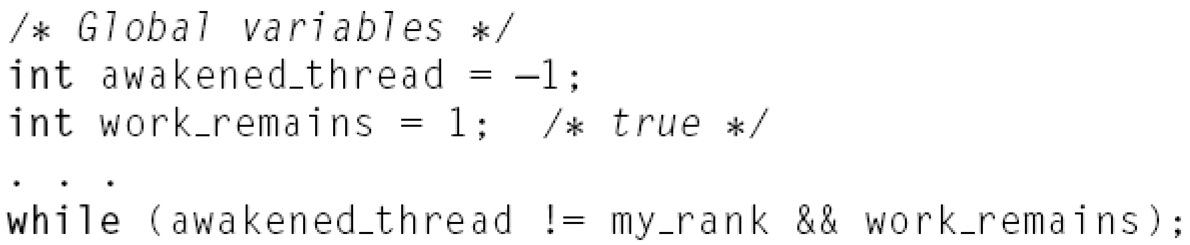


```
if (my_stack_size >= 2 && threads_in_cond_wait > 0 &&
      new_stack == NULL) {
   lock term_mutex;
   if (threads_in_cond_wait > 0 && new_stack == NULL) {
      Split my_stack creating new_stack;
      pthread_cond_signal(&term_cond_var);
   }
   unlock term_mutex;
   return 0;  /* Terminated = false; don't quit */
} else if (!Empty(my_stack)) /* Keep working */
   return 0;  /* Terminated = false; don't quit */
} else { /* My stack is empty */
   lock term_mutex;
   if (threads_in_cond_wait == thread_count-1)
                                        /* Last thread running */
      threads_in_cond_wait++;
      pthread_cond_broadcast(&term_cond_var);
      unlock term_mutex;
      return 1;  /* Terminated = true; quit */
   } else { /* Other threads still working, wait for work */
      threads_in_cond_wait++;
      while (pthread_cond_wait(&term_cond_var, &term_mutex) != 0)
      /* We've been awakened */
      if (threads_in_cond_wait < thread_count) { /* We got work *
         my_stack = new_stack;
         new_stack = NULL;
         threads_in_cond_wait--;
         unlock term_mutex;
         return 0;  /* Terminated = false */
      } else {  /* All threads done */
         unlock term_mutex;
         return 1;  /* Terminated = true; quit */
      }
   }  /* else wait for work */
}  /* else my_stack is empty */
```

Recall that a thread that has entered the condition wait by calling pthread _cond _wait(&term_ cond _var, &term _mutex); is waiting for either of two events:

- Another thread has split its stack and created work for the waiting thread.
- All of the threads have run out of work.

Perhaps the simplest solution to emulating a condition wait in OpenMP is to use busy-waiting. Since there are two conditions a waiting thread should test for, we can use two different variables in the busy-wait loop:



```
/* Global variables */
int awakened_thread = -1;
int work_remains = 1;  /* true */
. . .
while (awakened_thread != my_rank && work_remains);
```

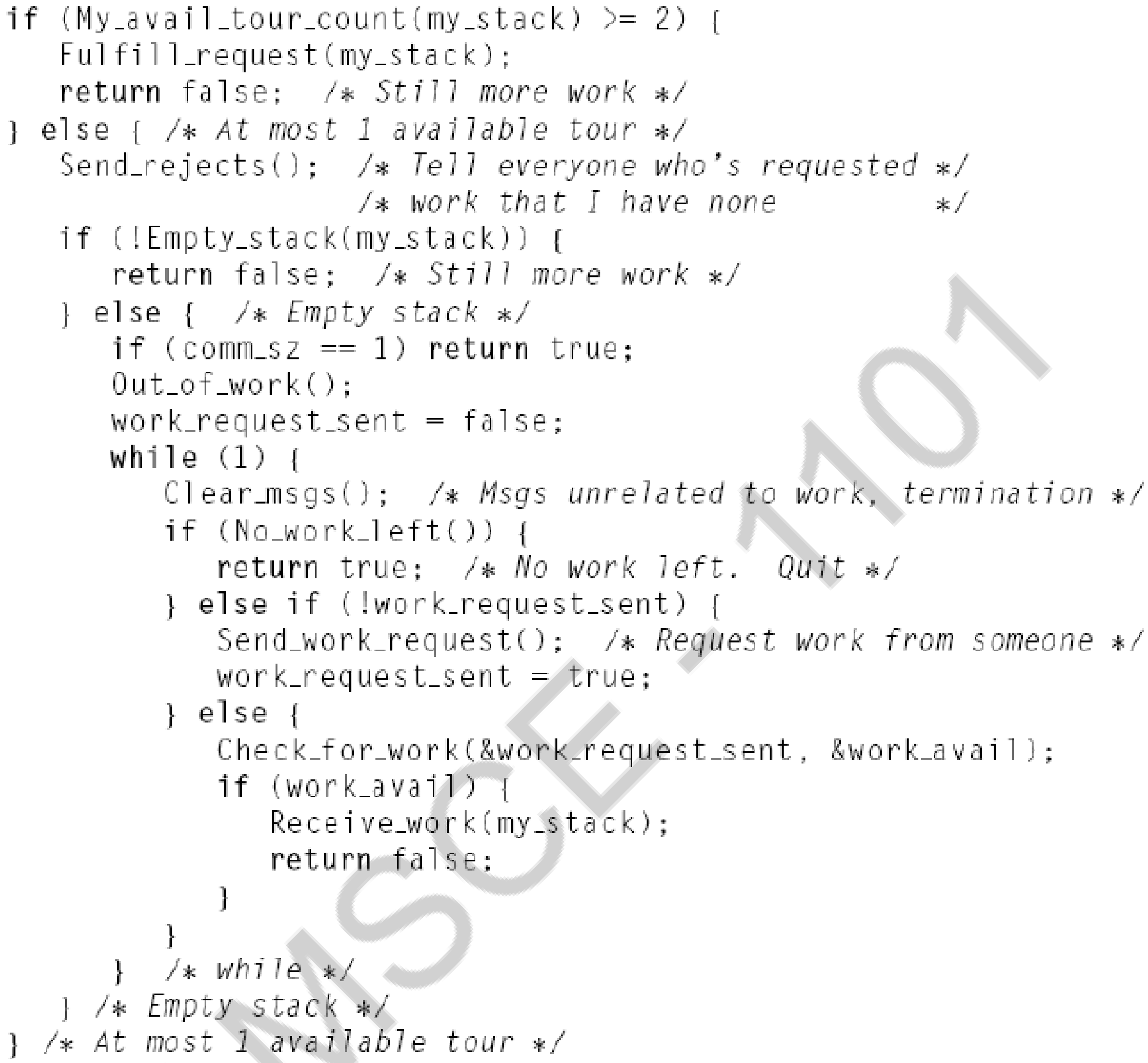
Initialization of the two variables is crucial: If awakened _thread has the value of some thread's rank, that thread will exit immediately from the **while**, but there may be no work available. Similarly, if work _remains is initialized to 0, all the threads will exit the **while** loop immediately and quit.

5. Explain about MPI based tree search dynamic portioning ?

Implementation of tree search using MPI and dynamic partitioning

In an MPI program that dynamically partitions the search tree, we can try to emulate the dynamic partitioning that we used in the Pthreads and OpenMP programs. Recall that in those programs, before each pass through the main **while** loop in the search function, a thread called a boolean-valued function called Terminated. When a thread ran out of work—that is, its stack was empty—it went into a condition wait (Pthreads) or a busy-wait (OpenMP) until it either received additional work or it was notified that there was no more work. In the first case, it returned to searching for a best tour. In the second case, it quit. A thread that had at least two records on its stack would give half of its stack to one of the waiting threads.

Much of this can be emulated in a distributed-memory setting. When a process runs out of work, there's no condition wait, but it can enter a busy-wait, in which it waits to either receive more work or notification that the program is terminating. Similarly, a process with work can split its stack and send work to an idle process. The key difference is that there is no central repository of information on which processes are waiting for work, so a process that splits its stack can't just dequeue a queue of waiting processes or call a function such as pthread cond signal. It needs to "know" a process that's waiting for work so it can send the waiting process more work. Thus, rather than simply going into a busy-wait for additional work or termination, a process that has run out of work should send a request for work to another process. If it does this, then, when a process enters the Terminated function, it can check to see if there's a request for work from some other process. If there is, and the process that has just entered Terminated has work, it can send part of its stack to the requesting process. If there is a request, and the process has no work available, it can send a rejection. Thus, when we have distributed-memory, pseudocode for our Terminated function can look something like the pseudocode shown in Program



```
if (My_avail_tour_count(my_stack) >= 2) {
   Fulfill_request(my_stack);
   return false;  /* Still more work */
} else { /* At most 1 available tour */
   Send_rejects();  /* Tell everyone who's requested */
                    /* work that I have none         */
   if (!Empty_stack(my_stack)) {
      return false;  /* Still more work */
   } else {  /* Empty stack */
      if (comm_sz == 1) return true;
      Out_of_work();
      work_request_sent = false;
      while (1) {
         Clear_msgs();  /* Msgs unrelated to work, termination */
         if (No_work_left()) {
            return true;  /* No work left.  Quit */
         } else if (!work_request_sent) {
            Send_work_request();  /* Request work from someone */
            work_request_sent = true;
         } else {
            Check_for_work(&work_request_sent, &work_avail);
            if (work_avail) {
               Receive_work(my_stack);
               return false;
            }
         }
      }  /* while */
   } /* Empty stack */
} /* At most 1 available tour */
```

My _avail_ tour_ count 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. Since sending a partial tour is likely to be a relatively expensive operation, it may make sense to only send partial tours with fewer than some cutoff number of edges.

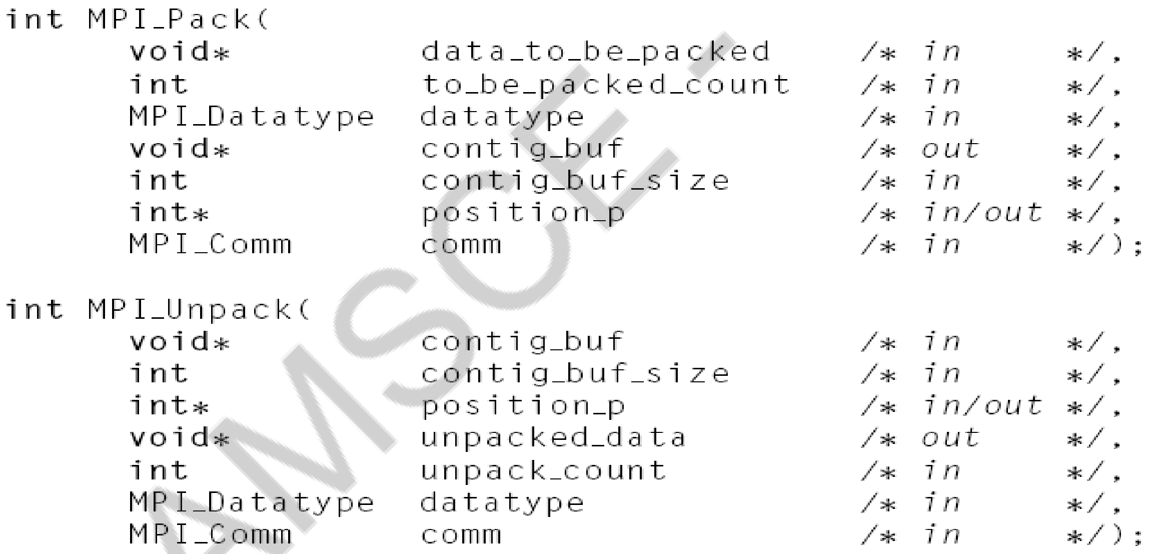
Fulfill _request

If a process has enough work so that it can usefully split its stack, it calls Fulfill _request (Line 2). 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. If there isn’t a request for work, the process just returns.

Splitting the stack

A Split _stack function is called by Fulfill _request. It uses the same basic algorithm as the Pthreads and OpenMP functions, that is, alternate partial tours with fewer than split _cutoff cities are collected for sending to the process that has requested work. However, in the shared-memory programs, we simply copy the tours (which are pointers) from the original stack to a new stack. Unfortunately, because of the pointers involved in the new stack, such a data structure cannot be simply sent to another process . Thus, the MPI version of Split _stack *packs* the contents of the new stack into contiguous memory and sends the block of contiguous memory, which is *unpacked* by the receiver into a new stack. MPI provides a function, MPI _Pack, for packing data into a buffer of contiguous memory. It also provides a function, MPI_ Unpack, for unpacking data from a buffer



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

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

MPI _Pack takes the data in data to be packed and packs it into contig _buf. The *position _p argument keeps track of where we are in contig _buf. When the function is called, it should refer to the first available location in contig _buf before data _to_ be_ packed is added. When the function returns, it should refer to the first available location in contig _buf after data _to_ be_ packed has been added.