Concurrent Programming ID1217

Course Summary

Module I: Parallel Programming Concepts, Models, and Paradigms

Q. Propose software implementation of shared memory abstraction.

A. One possible solution is to have one centralized server which provides the illusion of a shared memory for the nodes. The server is accessed thru message passing or RPC similar to a remote file server.

Each call to the server specifies a shared location, operation and the data in case of a store op. All accesses to the server are serialized.

The downsides of this solution is that the server is a single point of failure and cannot easily be scaled.

Another possible solution is a P2P DHT on a set of nodes with a key value store that is more robust and scalable.

Amdahl's law:

The speedup that can be achieved by executing a program in parallel is estimated by utilizing Amdahl's law.

Given a fixed problem size: Speedup (n processors) = $\frac{execution\ time\ (1\ processor)}{execution\ time\ (n\ processors)}$

The limiting factor of the speedup is the sequential parts of the program that cannot be parallelized due to data dependencies.

Amdahl's law states that the speedup that can be achieved by executing a parallel program on n processors is equal to:

$$\frac{1}{s + \frac{1-s}{n}}$$

Where s is the serial fraction of the serialized execution. The speedup is limited by $\frac{1}{s}$ when n approches infinity.

Example: if 10% of a program is serial the achievable speedup on 10 processors are: $\frac{1}{0,1+\frac{0,9}{10}} = 5,26$. While the maximum theoretical speedup on ∞ processors are $\frac{1}{0,1} = 10$

Assume a program, Let a be the percentage of its sequential execution time that can be executed simultaneously by a cores in a homogeneous multicore processor. Assume that the

remaining fraction of its sequential execution (1 - a) is inherently serial and must be executed sequentially by a single core. Each core has an execution rate of x MIPS (Million Instructions Per Second). 1. Derive an expression for the effective system MIPS rate when using the multicore system for exclusive execution of this program, in terms of n, a, and x.

Q.

Denote by T the sequential execution time and by y the effective system MIPS rate on an n-core system. The effective n-core system MIPS rate is the total number of instructions executed in the program T*x divided by the time it takes to execute this amount of instructions in parallel, i.e.

A.

$$y = \frac{T * x}{(1-a)T + \frac{aT}{n}} = \frac{x}{(1-a) + \frac{a}{n}} = \frac{1}{(1-a) + \frac{a}{n}} * x$$

Q. If n=16 and x=6 MIPS, determine the value of a that will yield a system performance of 51 MIPS.

A.

$$\frac{6}{(1-a) + \frac{a}{16}} = 51 \rightarrow a = \frac{16}{17} \approx 0{,}94\ i.e.\ 94\%$$

Q. Describe three distributed programming paradigms:

Iterative parallelism, recursive parallelism and pipeline parallelism.

Iterative parallelism:

Iterative parallelism is parallelism of independent iterations executed concurrently in separated processes or threads. Iterative parallelism results from parallelizing loops in sequential programs so that independent loop iterations are executed in parallel threads. Typically data parallel algorithms based on the idea of domain-composition use iterative parallelism. Example is matrix multiplication.

Recursive parallelism:

Parallelization of independent function calls i.e. making function calls in parallel.
Often based on sequential programs that utilizes divide and conquer algorithms.
Example is parallel sort.

Pipeline parallelism:

Parallelization of producers and consumers i.e. parallelism of different pipeline stages which involves producing (next data) and consuming (previous) data. There might be several

functional filters between producer and consumer. Example: Video decoder or streaming application.

Client-servers

Q. Propose and describe a software implementation of the synchronous and asynchronous message passing programming model in a shared memory multiprocessor.

A. The idea is to use a set of shared buffers to send and receive messages by storing/fetching to/from the buffers. The buffers can be provided in the system address space i.e. in the kernel. Buffers can be implemented using a data structure monitor that operates as a queue to store/load messages where a synchronous message passing can be used with a synchronous queue that is a blocking queue in which each put must wait for take and vice versa.

Implementation of message passing using shared memory is a consumer-producer problem and any solution to this problem can also be used to implement message passing.

Q. Name and define two basic parallel programming models. For each of the models, — describe basic operations used for process interactions; — name at least one programming environment (API) based on the model; — give at least one reason that can motivate a programmer to use the model instead of another one for development of a parallel application; give an example.

1. Shared memory programming model a.k.a. shared address space (SAS) in which multiple processes or threads communicate thru shared variables by read and write operations that requires some type of synchronization such as mutual exclusion locks and condition variables. Examples of this model is the Pthread and openMP APIs. The SAS model is convenient with parallelism such as iterative, recursive and pipeline executed on a multiprocessor system.

2. Message passing model a.k.a. distributed memory model in which processes communicate thru message passing using (a)synchronous send and receive operations. MPI, sockets, RPC, Java RMI APIs are all examples of environments used to develop distributed applications that provide access to remote resources such as video-on-demand or media live streaming.

Module II: Processes and Synchronization

Q. Explain interleaving semantics and non-determinism of concurrent execution. Indicate major reason for non-determinism in concurrent execution. Give examples of non-determinism in concurrent execution.

A.

Interleaving semantics: The concurrent execution of a parallel program by several processes can be viewed as the interleaving of execution histories of the processes.

Non-determism: The behavior of a concurrent program is not reproducible as different histories can be observed on different executions due to different execution rates of processes and dynamism of the execution environment.

Example: Concurrent increments of an unprotected shared counter by concurrent threads.

Q. Explain why and how synchronization together with a global invariant defined for a parallel application helps to develop a correct concurrent program.

A.

Global invariant (GI) is a predicate that is true in every visible program state – namely before and after every atomic action. It captures the relations between values of global and local variables as well as constraints on possible values of the variables.

Synchronization is an interaction between processes that controls the order in which the processes execute. There are two types of synchronization, mutual exclusion and condition synchronization.

In general, synchronization delays a process until it can proceed safely so that the program does not enter a bad state. By synchronizing processes the set of possible execution histories can be reduced to those in which the GI holds true.

Q. Name and define two major types of synchronization.

A. Mutual exclusion is used to guarantee that a shared resource / critical section is only accessed by one thread / process at a time. This is done by protecting the resource with a lock or semaphore that needs to be locked before accessing the resource. After the critical section has been accessed the lock is then unlocked allowing another thread to take the lock.

An example of this is if n >= 2 processes needs to update a shared counter then the incrementing operation becomes a critical section that must be executed by one process at a time with mutual exclusion.

Conditional synchronization on the other hand allows delaying a process until a certain condition becomes true. This is done by signal() and wait(). Example is the producer / scheme were the two waits for the other before proceeding.

Q. Explain differences between heavy-wait process and thread.

A.

A heavy-weight process executes in its own (virtual) address space that is completely isolated from address spaces of other HWPs. A HWP can create another HWP that is an exact copy of the parent but executes in its own address space. Normally, processes are scheduled and controlled by the OS kernel. A thread is essentially a program counter, a set of registers. and a stack used to store its private local variables, pass parameters to functions and return values. Threads are created in a HWP and share the heap. local variables, and the text segment (code), which belongs to the process. Threads ("green threads") can be scheduled and controlled by a run-time system in the user mode.

Q. Explain pros and cons of Shared single ready list vs one list per processor.

Shared single RL:	+ automatic load balancing - contention for shared lock - processes executes on different cores -> cold cache
RL per processor	-poor load balancing + less lock contention + processes executes on same processor -> warm cache

A hybrid strategy would be to in case of empty RL on one processor “steal” a process from the tail of another cores RL. In this way we combine load balancing with warm caches.

Disadvantages is higher contention for ready list but can be solved by stealing and dealing processes from tail and head.

Module III: Critical Sections. Locks and Condition Variables

Q. Describe properties and possible implementations of two different locking algorithms.

A. Test / Set lock and Ticket lock:

Test and set lock is an algorithm that returns the old value of a memory location and set the memory location to 1 as a single atomic operation. Lock value of 0 means the critical section is vacant and no process in it. Value of 1 means the critical section is being occupied.

Ticket lock is a synchronization lock that uses tickets to control which threads that is allowed to enter a critical section. The basic concept is similar to many ticket queue systems were customers / threads are served in the order they arrive. Like this, ticket lock is a FIFO queue based mechanism. It adds the benefit of fairness and works as follows.

There are two integer values, queue ticket and dequeue ticket. The first represents the threads position in the queue and the latter is the ticket. When a thread arrives it atomically obtains and increments the queue ticket. It then compares its ticket value with the dequeue ticket and if equal the thread can enter the CS. When a thread leaves the CS it must increment the dequeue ticket.

Q. Describe the covering condition technique used in condition synchronization.

A. Covering condition technique is typically used with monitors. A process signals a condition variable letting waiting threads know they can proceed. The covering condition might be weaker than the actual condition the threads waiting for.

The covering condition technique is worth using when processes synchronize on many different conditions. Instead of signaling a specific condition one “covering” condition that is a disjunction of the actual conditions is signaled to the waiting processes. This allows for reducing the number of condition variables used on the cost of false alarms.

The conditions are instead re-evaluated, this can be done in parallel by the signaled processes rather than sequentially by the signaling process. However, it results in some overhead of false alarms.

Q. For each of the following two options, show implementation of the await statement (i) using only locks; (ii) using locks and condition variables. For each of the options, explain when, do you think, it is better to use the option.

(i) Wait using only locks:

Mutex l;
lock(&l); while(!condition); {unlock(&l); lock(&l)} statements; unlock(&l);
Signaling code; lock(&l); condition = true; unlock(&l);

Drawback: busy-waiting for condition (waste of cpu time).
Use lock-only implementation when the waiting time is expected to be relative short compared to time it takes to block a thread as this is an expensive op performed by the kernel that requires context switching. Use lock-only when low contention, for example each process executes on separate processors. In case of limited resources this requires time-slicing that anyway requires context switches.

(ii) Wait using lock and condition variable.

Mutex l; cond cv;

Waiting process; lock(&l); while(!condition); wait(cv, &lock); statements; unlock(&l);
Signaling process; lock(&l); make condition true; signal(cv); unlock(&l);

For multiple waiting processes use signalAll().

This option shall be used in the case of high contention for limited resources (e.g. single CPU) so that waiting processes can be blocked / yielded and give up CPU. Use this option when the waiting time is expected to be rather long compared to a context switch.

Using only semaphores:

Sem m = 1, s = 0; Boolean B = false;

Waiting process; P(m); while(!B) {V(m); P(s); P(m);} statements; V(m);
Signaling process; P(m) B = true; V(s); V(m);

Q. Briefly explain what a critical section is.

A.

Critical section is a piece of code that must be executed with mutual exclusion (only one thread at a time) with respect to critical sections in other processes that reference the same shared variables. Critical sections becomes a problem when multiple threads/processes simultaneously execute their critical sections. This results in race conditions which alters the results of the program.

For example, if two processes increments a shared variable with a non-atomic instruction this may result in that the variable can have a multitude of values after completion.

The solution to this a different types of locks, test-set, ticket, bakery.

Q. What is a condition variable and what is it used for?

A. Condition variable is an opaque object that represents a queue of suspended processes waiting to be resumed. CVs provide a mechanism to wait and signal conditions in condition synchronization. There are three operations used:

1. wait(cv, lock) – Release the lock and wait in the cv queue until signaled.
2. signal(cv) – signal that the condition has been met and awaken a suspended process at the head of the waiting queue if any.
3. signalAll(cv) – signal that the condition has been met and awaken all suspended process if any.

There are three main signaling disciplines:

1. Signal and continue (SC) – the signaling process continues and the signaled process reacquires the lock.
2. Signal and wait (SW) – the signaling process passes the lock to the resumed process and reacquires the lock.
3. Signal and urgent wait (SUW) – as SW but the signaling process is placed at the head of the lock queue.

Q. Why is atomic instruction more effective than mutex?

A. Mutexes eventually end up being implemented with atomics. Since you need at least one atomic operation to lock a mutex, and one atomic operation to unlock a mutex, it takes at least twice as long to do a mutex lock, even in the best of cases compared to just update a counter with an atomic instruction.

Suppose your machine has the following atomic instruction:

int flip(int* lock): < *lock = (*lock + 1) % 2; return *lock; >

Consider the following suggested solution to the critical section problem for 2 processes:

```
int lock = 0;

process P[1 = 1 to 2] {

while (true) {

                while (flip(&lock) !- 1)
                                while (lock != 0) skip;
                critical section;
                lock = 0;
                noncritical section;
   }
}
```

Q. Does this solution work correctly? If not, give an execution order that results in both processes being in their critical sections at the same time.

A. No, the solution does not work correctly. One process flips lock to 1 and enters the critical section. Another process flips lock to 0 and enters the inner while loop – as lock is 0, so the

process goes back and flips lock again to 1 and enters the critical section. Thus, both processes execute the critical section at the same time.

Q. Suppose that the flip instruction is changed to do addition modulo 3 rather than modulo 2. Will the solution work for two processes? Shortly explain.

A. Yes it will, One process "flips" lock to 1 and enters the cs. Another process flips lock to 2 and spins in the inner loop until lock == 1, which happens when the first process exits the cs. Solution is not fair but safe.

Q. What is contention and how and how can it be solved?

A. Contention is when several processes competes for the same lock executing test and set, but the lock is already locked. Each TS is a write operation which causes a lot of memory accesses i.e. high memory contention and a lot of bus traffic in a bus-based shared memory multiprocessor (SMP). Or intensive network communication in distributed SMP. This makes the protocol slow and unfair. It could be improved with test, test and set or TS with exponential back off.

Module IV: Semaphores

Q. Explain "Passing the baton" technique used with semaphores.

A. Passing the baton is a synchronization technique used with semaphores. The baton is a mutex binary semaphore which controls access to a shared resource.

When one process decides to signal another process and give the mutex, it signals a (signaling) semaphore on which that process is waiting, the signaling process does not release the mutex. This signal has the effect of passing the baton (ownership of mutex) to the signaled process.

This is used in condition synchronization used with mutual exclusion. A signaling process holding the lock signals a signaling semaphore and passes the lock ownership and right to execute with mutual exclusion to the second process. The signaled process becomes the owner of the lock and is responsible for releasing the lock or pass it to another process.

Q. Write pseudo-code illustrating usage of semaphores for (i) mutual exclusion for critical sections (ii) condition synchronization (iii) 2-process barrier synchronization.

(i) Sem mutex = 1; P(mutex); critical section; V(mutex);

(ii) <await(B)> using passing the baton
Sem m = 1, s = 0; boolean B = false;
Waiting process; P(m); while(!B) { V(m); P(s); } V(m)
Signaling process; P(m); B = true; P(s);

(iii) Sem ar1 = 0, ar2 = 0;

Process1 { …; V(ar1); P(ar2); … }
Process2 { …; V(ar2); P(ar1); … }

Q. What is a semaphore?

A. Semaphore is a special kind of shared integer variable which can be only accessed using only two operations: P: <await (s > 0) s = s – 1> and V: < s = s + 1 >

The P operation waits until the semaphore is > 0 and then automatically decrements it.
The V operation increments the semaphore and resumes a process (if any) waiting on the semaphore.

Q. The await statement abstraction dictates that an executing thread has to wait for condition to become true, and then execute statements atomically. For each of the following options, give implementation of the await statement that blocks a waiting process until the condition becomes true: (i) using only semaphores; (ii) using only locks and condition variables. Do not use busy waiting. For each of the above options show also a code executed in another thread that sets condition to true and signals to all waiters if any.

A.

(i) Using semaphores

Sem m = 1; s = 0; Boolean B = false;
Waiting process: P(m); while(!B) {V(m); P(s); P(m); } statements; V(m);
Signaling process; P(m); B = true; V(s); V(m);

(ii) Using lock and cv

Mutex l; cond cv; boolean B = false;

Waiting process; lock(&l); while(!B) wait(cv, &l); statements; unlock(&l);
Signaling process; lock(&l); B = true; notify(&cv); unlock(&l);

Consider the following parallel program that creates two processes and uses semaphores:

```
Int x = 0; y = 0; z = 0;
Sem lock1 = 1; lock2 = 1;

Process foo {                   process bar {

z = z + 2;                          P(lock2);
P(lock1);                           y = y + 1;
x = x + 2;                          P(lock1);
P(lock2);                           x = x + 1;
V(lock1);                           V(lock1);
y = y + 2;                          V(lock2);
V(lock2);                           z = z + 1;
}                               }
```

Q. Will the program terminate?

A. Sometimes, Deadlock if both processes hold one lock and need the other lock to continue. Each process executes its first P operation and gets to the second P operation. Now both will wait forever in deadlock.

Q. What are all possible values for x, y and z?

A.

Deadlock state: x == 2, y == 1, z == 2:
Terminates: x == 3, y == 3, z == 1, 2 or 3: (not protected)

Module V: Monitors

Q. What is a monitor?

A monitor is a synchronization construct that allows threads to have both mutual exclusion and the ability to wait (block) for a certain condition to become false. Monitor is a thread-safe class, object, or module that wraps around a mutex in order to safely allow access to a method or variable by more than one thread. A monitor is in Java implemented with the *synchronized* keyword along with the constructs wait(), signal() and signalAll().

A monitor is a programming language construct (a synchronized object) which encapsulates variables, access procedures and initialization code within an abstract data type (a class).

Q. Consider the following monitor, which is proposed as a solution to the (SJN) allocation problem. Client processes call request and then release. The resource can be used by at most one client at a time. When there are (two or more competing requests, the one with the minimum value for argument time is to be serviced next. Consider two different signaling disciplines for the condition variable turn. For each of the disciplines indicate whether the monitor works correctly, and explain why or why not.

```
monitor SJN {

    bool free = true;
    cond turn;
    procedure request(int time) {

        if(not true) wait(turn, time);
        free = false;
    }
    procedure release( ) {
       free = true;
       Signal(turn);
    }
}
```

A.

Signal-and-continue

The above monitor does not work correctly for the Signal-and-Continue discipline, because of race conditions: After signal, if there is a new call of request, free is true so request does not block and the calling process can take the resource. Now the signaled process can also take the resource. This leads to the situation when two processes use the resource at the same time.

Signal-and-wait

The above monitor does work correctly for the Signal-and-Wait discipline. The signaled process executes next and sets free to false, so any subsequent request will wait.

Q. Regarding code of honeyBees and bear;

A. Consider the following case. Bear has not called eat() yet, Bees are filling the pot some bee increments the pot to h and signals full. The signal is lost because bear has not called eat yet nor waiting on full. So when bear finally calls eat it will wait on full forever and bees will wait on empty forever in deadlock. To avoid deadlock, bear should check in eat whether pot is full

Need to check for if(portions < h); wait(full);

Module VI: Message Passing; RPC and Rendezvous

Q. Shortly describe the relative advantages of synchronous and asynchronous message passing, RPC and rendezvous with respect to each other.

A.

Synchronous:

- Ensures that the message has been arrived
- Channels can be implemented without extra buffering
- Deterministic programming model – could be advantage

Asynchronous:

- More convenient for programming, more powerful programming model:
- Wildcard receive -> non-deterministic – could be advantage

RPC and rendezvous:

- High-level API; more convenient communication mechanisms for client/server applications.

Rendezvous vs RPC;

- Rendezvous calls are accepted, selected and served by an existing process that allows avoiding interference in processing of different calls that might occur in

RPC where calls are processed by separate processes, However, RPC might have more concurrency.

(i) Request/response – use RPC or rendezvous as it's convenient to program.
(ii) Notification – use asynchronous send more performance but synchronous is more reliable.
(iii) Exchange of values – use asynchronous send as it is more efficient and deadlock-free.
(iv) Producer-consumer – use asynchronous message passing

Q. Interaction between processes in distributed systems can be implemented using synchronous message passing, asynchronous message passing, or RPC (RMI). For each of the above communication mechanisms, assume that you are using only one mechanism to program interactions between processes. Can deadlock occur due to communication? If yes, how it can occur, and how do you avoid it?

A.

Synchronous message passing

Deadlock situations	To avoid deadlock
P1: synch_send(P2, &msg) receive(P2, &msg) P2: synch_send(P1, &msg) receive(P1, &msg)	P1: synch_send(P2, &msg) receive(P2, &msg) P2: receive(P1, &msg) synch_send(P1, &msg)
P1: receive(P2, &msg) synch_send(P2, &msg) P2: receive(P1, &msg) synch_send(P1, &msg)	

Assynchronous message passing

Deadlock situations	To avoid deadlock
P1: receive(P2, &msg) send(P2, &msg) P2: receive(P1, &msg) send(P1, &msg)	P1: send(P2, &msg) receive(P2, &msg) P2: send(P1, &msg) receive(P1, &msg)

Using only RPC (RMI): Deadlock can be caused by circular, nested calls from synchronized methods (procedures holding local locks). To avoid deadlock, release lock before calling, use unsynchronized methods (if possible), or avoid circular calls (if possible).

Q. In a multicore processor, each core has one or two levels of cache which means that there may be several cache copies of a data item in multiple cores at any given time. Answer the following questions briefly but accurately:

a) Assume a processor with four cores each running a thread sharing memory space with the other thread. Three of the cores have accessed a variable A. One of these threads then stores a

new value to A. What happens next and how does the hardware guarantee that the other cores see the new value, when needed?

A. Assume a bus-like interconnect (with broadcast of global memory operations). When the writing core executes the store-instruction, the cache controller will issue an invalidation message on the bus. The other cores will snoop on this transaction and the two that have the variable in their caches will take action and invalidate their copies. If they access the data again, they will experience a cache miss and the data will be swapped in from the core that did the update.

---

b) What is false sharing and how can you prevent it?

A. False sharing refers to the situation when two or more cores update separate variables that happen to reside in the same cache line. The cache coherence mechanism will then acts as if the same variable was accessed and invalidate the other cores copy. If this pattern happens repeatedly, performance will suffer greatly. The solution is to make sure that distinct data structures accessed by different cores are not allocated in close proximity with other data structures. Sometimes artificial padding is needed.

Q. Parallelize the following loop expressed in C using OpenMP, rewrite it as necessary.

```
for(i = 1; i < N; i++) {
   if(b[i] > 0.0)
     a[i] = 2.0 * b[i];
  else
     a[i] = 2.0 * operation_x(b[i]);
b[i] = a[i-1];
}
```

A.

```
#pragma omp parallel {

#pragma omp for schedule(guided, 10)

for(i = 1; i < N; i++) {
   if(b[i] > 0.0)
     a[i] = 2.0 * b[i];
  else
     a[i] = 2.0 * operation_x(b[i]);
}

#pragma omp for schedule(static)

for(i = 1; i < N; i++)

b[i] = a[i-1];
}
```

---

b) Consider the following code proposed as an implementation of the `sleep()` and `wakeup()` atomic operations using semaphores: `sleep` always blocks a calling thread until some other thread wakes it up; `wakeup` awakens *all* processes blocked in `sleep` since the last `wakeup`.

```
int dp = 0;
sem delay = 0;
sem e = 1;
sleep () {
        P(e); dp++; V(e); P(delay);
    }
 wakeup(){
        P(e);
        while (dp > 0)  {
            dp--; V(delay);
        }
        V(e);
    }
```

What is wrong with the above code? Explain. Fix the code so that it is correct. (6 p)

The above code causes race conditions on `P(delay)` in the `sleep` method. The fixed code below is using the passing-the-baton technique to wakeup the sleeping threads when they are signaled.

```
int dp = 0;
sem delay = 0;
sem e = 1;
sleep () {
    P(e); dp++; V(e); P(delay); dp--;
    if (dp > 0) V(delay);
    else V(e);
}
wakeup(){
    P(e);
    if (dp > 0) V(delay);
    else V(e);
}
```

Problem V. Concurrent Programming with OpenMP

Consider the following simple (incomplete) block of code:

```
p=head;
while (p) {
   processwork(p);
   p = p->next;
}
```

The function `processwork(void *p)` does not have any side-effects.

a) Parallelize this code using the OpenMP task concept.

b) Parallelize this code in OpenMP without using the task concept.

For both sub-problems, rewrite as necessary.

Solution outline:

a)

```
#pragma omp parallel     // Create parallel threads
#pragma omp single       // Let one thread continue, the
                         // other are waiting for work
{                        // Begin single region
    p=head;
    while (p) {
#pragma omp task firstprivate(p)    // create a task of processwork
       processwork(p);               // p must be private and not shared
       p = p->next;
    }









}                  // end single region
```

b)

```
while (p != NULL) {      // find out number of elements
       p = p->next;
       count++;
 }
 p = head;
 for(i=0; i<count; i++) {      // put them in an array
       parr[i] = p;
       p = p->next;
    }

// Do the processing in parallel
 #pragma omp parallel for schedule(static,1)
      for(i=0; i<count; i++)
         processwork(parr[i]);
```

Assume a program, which solves a problem of a varying size. Let s be a *serial* fraction of its sequential execution that is inherently serial and must be executed sequentially on a single core. The remaining fraction $(1-s)$ can be executed in parallel on n cores. Assume that increasing the number of cores n allows solving the problem of a larger size, i.e., the problem size grows with n, and the serial fraction s varies with n. Derive an expression for s as a function of n that shows how s would have to vary with n in order to achieve linear speedup as the problem size grows with n (i.e., speedup proportional to n).

According to the Amdahl's law (see Lec. 1), the speedup due to parallel execution on n cores is

$$S_n = \frac{n}{n \cdot s + (1-s)}$$

Assuming linear speedup $S_n = C \cdot n$ for a constant $0 < C \le 1$, we get the following equation

$$C \cdot n = \frac{n}{n \cdot s + (1-s)}$$

From the above equation, we obtain the expression for s, as a function of n, to achieve linear speedup with the constant C

$$s = \frac{1-C}{C \cdot (n-1)} = \left(\frac{1-C}{C}\right) \cdot \frac{1}{n-1}$$