2202-COL380 Major

Chinmay Mittal

TOTAL POINTS

40.5 / 77

QUESTION 1

1 Q1 **1 / 2**

+ 2 pts Correct

✓ **+ 1 pts** *pragma omp critical is correctly explained*

+ 1 pts Significance of Anonymous is explained correctly.

+ 0 pts Incorrect / Unattempted

QUESTION 2

2 **4 / 4**

✓ **+ 1 pts** *Correct Explanation for Processor consistent*

✓ **+ 1 pts** *Correct Explanation for Sequentially consistent*

✓ **+ 2 pts** *Correct example*

+ 0 pts wrong

QUESTION 3

3 **1 / 4**

+ 1 pts Correct definition of Atomic

+ 1 pts Correct definition of Instantaneous

+ 1 pts explaination for why the instruction is not atomic

✓ **+ 1 pts** *explaination for why the instruction is not instantaneous*

+ 0 pts wrong

QUESTION 4

4 **0 / 3**

+ 3 pts Correct

+ 2 pts memory flush with barrier

+ 1 pts partially correct

✓ **+ 0 pts** *Incorrect / Unattempted*

QUESTION 5

5 **2 / 2**

✓ **+ 1 pts** *Correct Definition of Lock-free*

Lock free protocols ensure that there is system wide progress even if individual threads might be suspended

✓ **+ 1 pts** *Correct Explanation*

Protocols with a critical section may not always be lock free because the entity which has acquired the lock and is in the critical section might be suspended or stuck. In that case no other thread would be able to make progress.

+ 0 pts Incorrect

QUESTION 6

6 **2 / 3**

+ 3 pts Fully correct

+ 0 pts Not Attempted/Incorrect

✓ **+ 1 pts** ***Not** lock free since it is a mutual exclusion algorithm. Lock-free by definition must ensure system-wide progress — at least one ready thread, no matter the state of the other threads.*

✓ **+ 1 pts** ***Not** wait free, since by definition wait-freeness ensures that every ready thread progresses, no matter the state of other threads.*

+ 1 pts Starvation free, since algorithm ensures every thread to get an **opportunity** to proceed as long as all threads are scheduled.

QUESTION 7

7 **1.5 / 2**

+ 2 pts Parallelism at **every level** i.e network, disk, and server level. There shouldn't be a single point of throughput.

✓ **+ 1 pts** *Partial correct*

+ 0 pts Not attempted/incorrect

+ 0.5 *Point adjustment*

1 It should be parallel at every level, not just for read and writes (see rubrics)

QUESTION 8

8 **3 / 3**

✓ **+ 3 pts** *The map function generates (key, value) pairs. There can be multiple pairs for a single key. The map-reduce framework must sort these pairs into bins by key.*

+ 1.5 pts The student mentions the separation of keys, however, the explanation is unclear.

+ 0 pts Incorrect or not attempted.

QUESTION 9

9 **3.5 / 4**

+ 4 pts Fully correct

✓ **+ 2 pts** *Knows linearizability and sequential consistency, and the difference in terms of globally visible time.*

+ 1 pts General idea of ordering is there, but the distinction on global time is not clear

✓ **+ 1 pts** *Usefulness of linearizability*

+ 1 pts Usefulness of sequential consistency

+ 0 pts Incorrect or Didn't attempted

+ 0.5 *Point adjustment*

2 use case of sequential consistency is missing

QUESTION 10

10 **4 / 4**

✓ **+ 4 pts** *Fully Correct answer*

+ 1 pts Definition of common CRCW

+ 1 pts Definition of Priority CRCW

+ 1 pts Why common CRCW is also a priority CRCW algorithm

+ 1 pts Why priority CRCW may not be common CRCW algorithm

+ 0 pts No/Incorrect answer

QUESTION 11

11 **5 / 5**

+ 0 pts Model Solution:

```
void MPI_Barrier(MPI_Comm comm) {
int barrier_value = 1;
int result;
MPI_Allreduce(&barrier_value, &result, 1,
MPI_INT, MPI_SUM, comm);
}
```

Collectives like MPI_AllReduce, MPI_AllGather and MPI_Alltoall can be used as barriers, as these are blocking calls involving all entities and none may return until the receive buffer has their data —

and data needs to be received from all, which can only happen after all have called that collective.

The Communicator and send and receive buffers are the important parameters. (Root is also important for non-barrier based collectives)

Correct Implementation

✓ **+ 2 pts** *Implemented the barrier using the above method*

+ 2 pts Used other mechanisms which also work (for eg each thread can call broadcast and wait for the root, the root can call gather and call broadcast again when it has received from everyone, to signal they can proceed)

✓ **+ 2 pts** *Argued for correctness*

✓ **+ 1 pts** *Listed important parameters*

Partially correct with collectives

+ 2 pts Used collectives but minor mistakes result in an almost correct barrier

+ 1 pts Tried to justify

Without collectives

+ 1.5 pts Used send/receive operations to construct a barrier

+ 0.5 pts Tried to justify

+ 0 pts Incorrect/Unattempted

QUESTION 12

12 **0 / 5**

+ 5 pts Correct

- 1 pts Early scheduling not ensured (task 4/5/7 go early. 3/6/9 go last)

- 2 pts incorrect use of depend clauses

✓ **+ 0 pts** *Incorrect / Unattempted*

QUESTION 13

13 **5 / 5**

✓ **+ 1 pts** *Identifying the critical paths: `(4, 5, 8, 9), (4, 7, 8, 9)` and hence should be scheduled accordingly.*

✓ **+ 4 pts** ***Parallel streams***

Synchronization across streams is guaranteed by busy-waiting on a flag or synchronizing individual streams.

+ 4 pts **Two streams: Serial**

- `(1, 2, 3)` in stream 1. Rest in stream 2.
- Serialize stream 2 as follows:

4 before 5 & 7.

5 & 7 before 8.

8 before 6 & 9.

+ 2 pts **Without streams**

Order guaranteed using global synchronization.

Without using Streams, it becomes all sequential. The sync calls are not useful. While maintaining correct dependency order is okay, it becomes a trivial problem. Scheduling earlier and allowing concurrency is important.

+ 0 pts Incorrect or not attempted.

+ 5 pts Cuda graph

QUESTION 14

14 **2 / 6**

+ 0 pts Non-answer

✓ **+ 0.5 pts** *Identified any race*

+ 2 pts Eliminate Race on testing of parent and update.

- 1 pts Elimination of race on parent is not lock-free (Remember critical section/barriers do not

make it lock-free.)

+ 1 pts Shows lock-freeness on parent test correctly

✓ **+ 2 pts** *Eliminate race on front*

✓ **- 1 pts** *Elimination of Race on front is incorrect/not lock-free*

+ 0.5 pts Show lock-free elimination of race on front

+ 0.5 *Point adjustment*

3 atomic?

QUESTION 15

15 **2.5 / 6**

+ 0 pts Non-answer

✓ **+ 2 pts** *Problem with Kernel: separate flags per block*

+ 1 pts Problem in Kernel: initialized flag in each thread: could be after another thread has entered critical section.

+ 1 pts Problem with Blocks: Initialized flag per thread

+ 2 pts Problem with Warp: Usually no divergence in flag=0, but lock loop can have divergence and could deadlock on CAS if the failed threads continue to schedule. unlock should be updating flag globally -- fence required

- 1 pts Inaccurate/Incomplete Warp explanation

+ 0.5 *Point adjustment*

QUESTION 16

16 **4 / 6**

+ 0 pts Non-answer

✓ **+ 2 pts** *t(n,p) = ?*

✓ **+ 1 pts** *log n version and its time on P Iso-efficiency*

✓ **- 0.5 pts** *Demonstrates log n knowledge, but incorrect computation*

✓ **+ 1 pts** *loglog n version and its Iso-efficiency*

✓ **- 0.5 pts** *Demonstrates loglog n knowledge, but incorrect computation*

✓ **+ 1 pts** *Accelerated cascading version and Iso-efficiency*

✓ **- 0.5 pts** *Demonstrates accelerated cascading knowledge, but incorrect computation*

✓ **+ 1 pts** *comparisons*

✓ **- 0.5 pts** *Incomplete comparisons*

4 Is-eff cannot be a fn of n.

QUESTION 17

17 **0 / 6**

+ 6 pts Completely Correct answer

+ 2 pts O(1) time Correct answer but not work-optimal.

+ 2 pts Correct work optimal but not O(1) time

+ 3 pts Correct answer with partitioning approach but not work optimal.

✓ **+ 0 pts** *No/Incorrect answer*

QUESTION 18

18 **0 / 7**

✓ **+ 0 pts** *Non-answer*

+ 4 pts Data collection and multiplication

- 2 pts Missing collectives

- 2 pts Partially correct data collection with collectives

- 3 pts Partially correct collection without proper collectives.

+ 2 pts Full Cost analysis

- 1 pts Erroneous analysis

+ 1 pts Result in the right place

- 0.5 pts Redistributing result

Entry Number: 2020CS10336 Name: CHINNAY MITTAL.

-2 marks for missing or illegible entry no./name on any sheet.

COL 380 MAJOR EXAM

SEMESTER II 2022-2023

2 hours, 18 questions, 75 marks (+2 extra credits)

Maximum marks are listed in [] for each question. Justify your answers: most marks are designated for correct justification. Write legibly to receive credit. Follow IITD/COL380 Academic Integrity Code.

1. [2] What does the following stand for in OpenMP? What is the significance of 'Anonymous?'

```
#pragma omp critical(Anonymous)
```

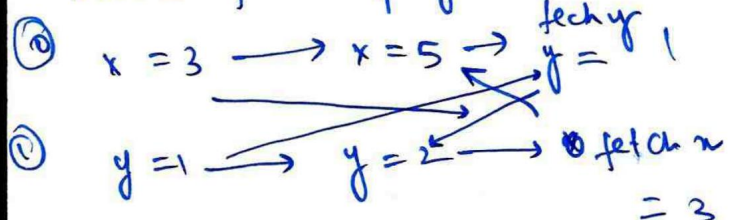
These are named critical sections, in openMP. if we use names for critical sections then at one time only one thread can be executing a critical section with a particular name. two threads can execute critical sections with different names simultaneously.

Anonymous means no names. Out of all sections with no names, only one of them can be active at a given time. (at most)

2. [4] Explain Processor Consistent and Sequentially consistent execution. Show an example that is Processor consistent but sequentially inconsistent.

Processor consistent ⟹ All writes made by a thread should be seen in that order by all other threads. Also there should be a globally consistent sequential view of writes made to a particular variable by all threads.

Sequentially consistent ⟹ There should exist a consistent sequential global ordering of instructions of all threads such that the instructions of every thread follow program order in this global sequence.



sequentially inconsistent because of the cycle.

processor consistent

↳ Thread 0's view

x=3 → x = 5 → y=1 → fetch y → y=2

Thread 1's view

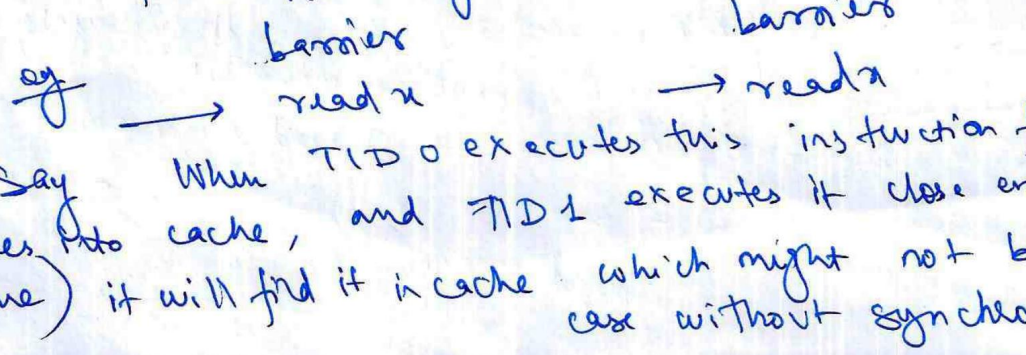
y=1 → y=2 → x=3 → fetch x → x=5

3. [4] Is an architecture (ISA) instruction like `mov (%rbp), %eax` instantaneous? Atomic? Explain.

These instructions are ~~not~~ not. instantaneous. They take a finite time to complete. Which involves the time the hardware takes to transfer data from one register to the other using the ~~wire~~ wire interconnect between these registers. But the hardware ensures that these instructions are atomic wrt the registers involved. No other thread can access these registers when this instruction is being executed, which makes it atomic wrt these registers and this atomicity is ensured by the hardware.

4. [3] I add a missing `#pragma omp barrier` in a parallel region in my code, but notice a significant increase in the the number of cache transaction shown by my profiler. What could the reason(s) be?

Threads executing on different cores can share the same cache and can also have shared memory. if there is no barrier then there is no synchronization and all threads can be executing at their own different pace, and accesses to shared variables by different threads can occur at very different global times not allowing them to benefit from caching. On the other hand barrier imposed synchronization which can potentially lead to shared variables being accessed at similar times (global) and thus can benefit from caching.



Say When TID 0 executes this instruction first x comes into cache, and TID 1 executes it close enough (in global time) it will find it in cache which might not be the case without synchronization.

Entry Number: 2020CS10336 Name: CHINMAY MITTAL

5. [2] Explain why protocols with critical section are NOT lock-free. (Define lock-free and explain.)

Lock free algorithms ⟹ some synchronizer must progress independent of the scheduler.

Consider a program with one thread in the critical section and others waiting to execute the critical section.

if the scheduler chooses to repeatedly schedule one of the waiting threads, then no body makes progress.

Since the waiting threads keep waiting for the thread in the critical section to exit and this thread is never scheduled.

So depending on who the scheduler ~~tries to~~ schedules nobody might make progress.

6. [3] Which of the following describe the Bakery algorithm: wait-free/lock-free/starvation-free. Explain

Bakery algorithm relies on the scheduler and is not wait free or lock free ⟶ it is possible that nobody can progress depending on the scheduler.

if the thread executing the critical section is not scheduled repeatedly, then nobody makes progress. All the waiting threads keep waiting for the thread in the critical section which never exits because it is never scheduled by the scheduler

But it is starvation free, with the proper scheduling algorithm, the thread in the critical section will exit. and all threads will make progress one by one.

⟹ everyone progresses with the help of the scheduler ⟶ starvation free.

7. [2] What makes a parallel file system parallel?

The ability to read multiple files concurrently, at different locations by different processors.

1

8. [3] In the Map-Reduce programming model, what must the underlying framework do after the end user's Map() calls and before the Reduce() calls (in a message-passing environment).

In a message passing environment each processor will be responsible for reducing a particular key, which processor processes which key can be figured out by the id of that processor. It is possible that the <key, value> pairs of same key are spread across the processors. Each processes needs to transmit a message to the correct processor responsible for that key, (containing <key, value>) or transmit that it has no such pair,

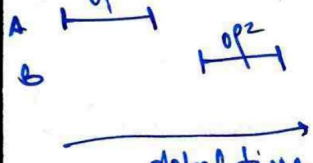
Once each processor gets all pairs of the key for which it is responsible it can perform the reduce function.

Each processor needs to ensure that it gets all pairs corresponding to a key from all other processors.

Entry Number: 2020CS10336 Name: CHINMAY MITTAL

9. [4] What is the difference between Linearizability and Sequential consistency? Explain how they are useful.

Sequential consistency ⇒ there exists a consistent global ordering of all instructions executed by all threads, such that the instructions of every thread appear in program order in this global execution. The write to a variable must appear before the read of that variable of which it is a read effect. Without any writes to the same variable in between. Sequential consistency is about relative ordering without any notion of global time. Linearizability is a stronger condition where for every instruction. There is a start and end time (global clock) and the instruction takes effect at some time b/w them.



op2 op1 is sequentially consistent, but is not linearizable since op1 must end before op2 starts. ⇒ thus linearizability leads to additional constraints based on this time. And it is harder to achieve in hardware with real time guarantees are important. eg A is check amount balance for a global clock. But for certain applications these type of B is credit card payment

10. [4] Explain how a common-CRCW PRAM algorithm is also a Priority-CRCW algorithm, but not *vice-versa*.

Consider a common-CRCW PRAM algorithm it is also a valid priority CRCW algorithm, the read steps and computation steps do not pose any problem if they are valid in common CRCW they are valid in priority CRCW. Writes to different locations also do not pose a problem. Consider x threads writing to the same location. All threads try to write the same value of to the memory location. After common CRCW write the memory location will be updated to of. These writes remain valid in priority CRCW, the top priority processor will write of to the memory location and hence all memory updates are correctly preserved.

A priority CRCW algorithm is not be a valid common-CRCW algorithm. At some concurrent write step many processor might be trying to write different values to the same location and the correctness of the algorithm depends on the fact that the highest priority processor writes the correct value. Since they might be writing different values, this is not a valid common CRCW algorithm.

11. [5] Implement *MPI_barrier* using MPI collective message passing calls (list important parameters.) Explain why it is correct.

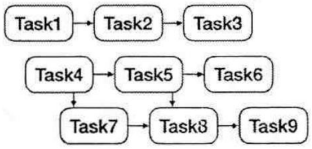
```
func Barrier ( MPI_Comm ) .
{
    int size ; int rank ; MPI_Comm_Rank (MPI_Comm, &rank);
    MPI_Comm_size (MPI_Comm, &size);
    MPI_Request arr[] = new MPI_Request [size];
    int recv_arr[] = new recv[size];
    for (int i = 0 ; i < size ; i++)
    {
        if (i == rank) {
            int send = 1;
            MPI_Ibcast (&send, 1, MPI_INT, i, MPI_Comm, &request_arr[i])
        } else {
            MPI_Ibcast (&recv[i], 1, MPI_INT, i, MPI_Comm, &request_arr[i]);
        }
    }
    MPI_Waitall ( request_arr, size );

    return ;
}
```

Each process will make n non blocking Broadcast calls each with root 1 . . . . . n . Hence n groupwise matching occurs. process will return only when it has received confirmation from all n processes. Using Ibcast & Waitall ensures no implicit synchronization used by these calls . Otherwise we would use synchronization to implement synchronization.

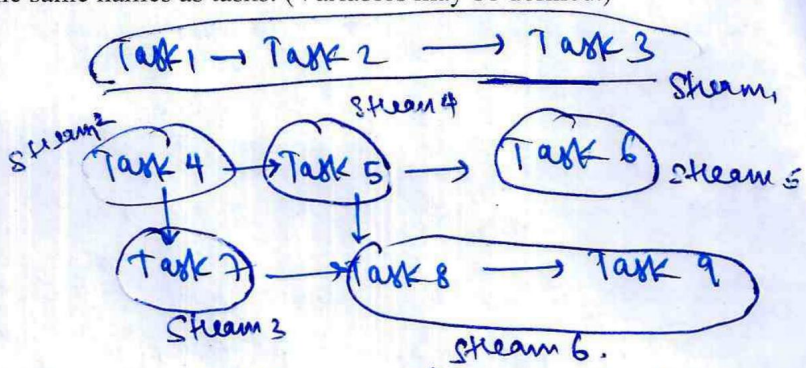
Entry Number: 2020CS10336 Name: CHINMAY MITTAL

12. [5] Implement the given task graph in OpenMP (approximate code). Assume tasks are functions (with the same name). Do not define variables. Identify and ensure early scheduling of the critical path.



13. [5] Repeat ~~Q15~~ Q12 for Cuda. Assume kernels with the same names as tasks. (Variables may be defined.)

We can use Cuda Streams to implement this task graph




```
cudaStream_t stream1, stream2, stream3, stream4, stream5, stream6;

cudaStreamCreate(&stream1); cudaStreamCreate(&stream2); cudaStreamCreate(&stream3);
cudaStreamCreate(&stream4); cudaStreamCreate(&stream5); cudaStreamCreate(&stream6);

Task4 <<< params, 0, stream2 >>> ( );
task1 <<< params, 0, stream1 >>> ( );
task2 <<< params, 0, stream1 >>> ( );
task3 <<< params, 0, stream1 >>> ( );
cudaStreamSynchronize ( stream2 );
task7 <<< params, 0, stream3 >>> ( );
task8 <<< params, 0, stream4 >>> ( );
task5 <<< params, 0, stream4 >>> ( );
cudaStreamSynchronize ( stream3 ); cudaStreamSynchronize (stream4);
task8 <<< params, 0, stream6 >>> ( );
task9 <<< params, 0, stream6 >>> ( );
task6 <<< params, 0, stream5 >>> ( );
cudaStreamSynchronize (stream5);
cudaStreamSynchronize (stream6);
cudaStreamSynchronize (stream1);
```

params are the appropriate block & grid dims can be set to 1, 1 if only one thread is required;

Entry Number: 2020CS10336 Name: CHINMAY MITTAL

14. [6] Consider pseudo-code shown below for breadth-first traversal of a directed graph G = (V, E), where $e \in E = (v1, v2)$, where $v1, v2 \in V$. Assume *forall* continues to schedule threads in parallel until members remain in the set. Identify and correct the race conditions in the pseudo-code below in a lock-free manner. (Prove lock-freeness.)

```
BFS(G):
   forall v ∈ G.V:
      parent[v] = -1
   Front = {random vertex r ∈ G.V}
   parent[r] = r
   Update(Front)
```

```
Update(front):
   forall v1 ~ front:
      forall v2 ~ G.out_neighbor(v1):
         if (parent(v2) == -1:
            parent[v2] = v1; visit(v2)
            front += v2 // Add to front
```

Since front is a shared variable it can be potentially updated by many threads in parallel and lead to incorrect updates.

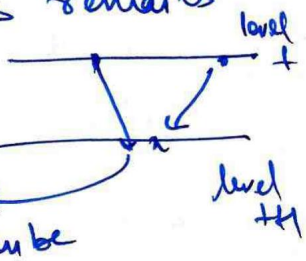
We can use CAS to update front in an atomic manner.

do { local ← front
new ← front + v2;
} while (CAS(front, local, new));

CAS ensures that the algorithm remains lock free

CAS depends on hardware and is updated independent of the scheduler. Hence remains lock free.

Updating parent or visit is not a race condition. The BFS remains correct irrespective of the order of execution of instructions



The last operation to succeed is correct both can be valid answers, hence we don't need mutual exclusion.

We must ensure that depth t+1 starts only when depth t is completely computed. Hence there is a need for synchronization after the inner for loop

15. [6] I wrote the following for mutual exclusion using CAS on Cuda for `akernel`, but failed. I tried limiting it to threads within a block and still failed. I finally tried it only within threads of a warp, and failed again. Explain.

```
__device__
void unlock(int &flag){
    flag = 0;
}
```

```
__device__
void lock(int &flag){
    int old;
    do old = atomicCAS(&flag, 0, 1);
    while (old != 0);
}
```

```
__global__
void akernel(void) {
    __shared__ int flag;
    flag = 0;
    lock(flag);
    critalsection();
    unlock(flag);
}
```

The flag variable is a shared variable and is shared b/w threads of a block. It is possible that threads of different blocks both acquire the lock since the flag variable copies they maintain are different variables and both can be updated to one.

There is ~~no memory synchronization performed~~ by CAS. it is possible that each thread maintains the shared flag variable in its register and updates it their without other threads in the block ignoring that out. We need to use volatile to ensure flag is not kept in registers.

Entry Number: 2020CSB10336 Name: CHINMAY

16. [6] Compare the scaling behavior of O(log n)-time binary-tree based minima-finding, O(loglog n) non-optimal minima-finding, and the accelerated cascading based minima-finding in terms of their iso-efficiency functions.

Consider the $O(\log(n))$ time algorithm

$t(n) = O(\log(n))$

$W(n) = O(n) \Rightarrow t(n,p) = \left\lfloor \frac{W(n)}{p} \right\rfloor + t(n)$

$= O\left(\frac{n}{p} + \log n\right)$

$t_1(n,1) = O(n)$

for iso efficiency we have $\frac{t_1(n,1)}{p\, t(n,p)} \approx$ ~~$\leq$~~ $\leq$

$\Rightarrow \frac{n}{p * \left(\frac{n}{p} + \log n\right)} \approx K$

$\Rightarrow \frac{n}{n + p\log n} \approx K$

$\Rightarrow p\log n$ is $\Omega(n)$

$\Rightarrow p$ is $\Omega\left(\frac{n}{\log n}\right)$

since $t_1(n,1)$ is $O(n)$

isoefficiency $\approx \Omega\left(\frac{n}{\log n}\right)$ → Weakly scalable

~~State~~

For $O(\log\log n)$ non optimal solution

$t(n) = O(\log\log n)$

$W(n) = O(n \log\log n)$

$t(n,p) = \frac{W(n)}{p} + t(n) \approx O\left(\log\log n + \frac{n}{p}\log\log n\right)$

$\frac{O(n)}{O\left(\frac{n}{p}\log\log n + \log\log n\right)p} \approx \frac{O(n)}{n\log\log n + p\log\log n}$ → not scalable irrespective of p will keep θ decreasing.

Accelerated Cascading

$t(n) \approx O(\log\log n)$

$W(n) \approx O(n)$

$f(n,p) = \frac{W(n)}{p} + f(n)$

$\approx O\left(\frac{n}{p} + \log\log n\right)$

$\frac{t(n,1)}{p(t(n,p)} \approx t \approx \frac{n}{n + p\log\log n} \approx t$

$\rightarrow p\log\log(n)$ is $\Omega(n)$

p is $\Omega\left(\frac{n}{\log\log n}\right)$ → weakly scalable

less scalable than $O(\log n)$ algorithm

17. [6] Provide a work-optimal O(1)-time common-CRCW PRAM algorithm to find the first occurrence (i.e., the smallest index) of value v in array A. (v may appear 0 or more times in A.)

A sequential algorithm can simply iterate through the array and keep track of the first occurence of V ⇒
best sequential time $O(n)$ ⇒ optimal work $O(n)$.

Assuming priority CRCW per index
each processor tries to write its index (if it has the element ↓ read A[i]) on the answer location

Entry Number: 2020CS10336 Name:

18. [7] Consider two nxn matrices A and B, initially distributed *block-wise* among pxp ranks of a communicator. Each block is bxb (and $n = bp$). Provide MPI-like code to implement C = AxB. C also must be distributed block-wise like A and B. Use collectives where possible (list important parameters). Measure message passing cost.

ROUGH PAGE. WILL NOT BE GRADED