Q1

Give a definition of *Atomic Operation*. **0 of 2 points**

Solution Path

A set of operations that appears to the rest of the system to occur at once without being interrupted (it is uninterruptable and indivisible).

Give a description of the *lock-free synchronization pattern* and its main concept. Explain what the system has to provide to enable an efficient implementation of this pattern. **0 of 4 points**

Solution Path

Description: Thread-safe (in terms of correctness) access to shared data without the use of synchronization primitives such as mutexes (which are based on a lock).

System requirements: atomic operations on (some) plain datatypes, examples from the lecture were CAS / TAS.

Q2

Consider the following C++ code:

```
1  /* headers left out for brevity */
2
3  // file-scope (= global) variables
4  std::atomic<int> d;
5  std::atomic<int> n;
6  int x;
7  int y;
8
9  void thread1()
10 {
11    d = 108;
12    x = n;
13 }
14
15 void thread2()
16 {
17    n = 108;
18    y = d;
19 }
20
21 int main()
22 {
23    // initialization
24    d = 0; n = 0;
25
26    // code left out for brevity
27    // function thread1() will be executed by thread with id 1
28    // function thread2() will be executed by thread with id 2
29    // both threads may potentially execute simulatenously
30
31
32    // code left out for brevity
33    // threads with id 1 and 2 are completed at this point
34    std::cout << "x: " << x << " and y: " << y << std::endl;
35
36    // ...
37 }
```

List all possible outcome tuples `(x, y)` in line 34. Then, state for each tuple if it is a valid outcome in sequential consistency. **0 of 4 points**

Solution Path

Possible results for `(x,y)` in line 32 in sequential consistency (SC) are:

- (0, 108)
- (108, 0)
- (108, 108)

The result (0, 0) is not possible because it violates the program order requirement of SC.

Q3

Shared Memory: Asynchronicity

Consider the following C++ code:

```
1  /* headers left out for brevity */
2
3
4  void function_A(MyDataStructure & ds)
5  {
6     // ...
7     // reads and modifies parts of ds (passed as reference)
8  }
9
10 void function_B(MyDataStructure & ds)
11 {
12    // ...
13    // reads and modifies parts of ds (passed as reference)
14 }
15
16 void function_C(MyDataStructure & ds)
17 {
18    // ...
19    // reads and modifies parts of ds (passed as reference)
20 }
21
22 void function_D(MyDataStructure & ds)
23 {
24    // ...
25    // reads and modifies parts of ds (passed as reference)
26 }
27
28 void function_E(MyDataStructure & ds)
29 {
30    // ...
31    // reads and modifies parts of ds (passed as reference)
32 }
33
34 int main()
35 {
36    // declaration and initialization left out for brevity
37    MyDataStructure ds;
38
39    // *** begin your code ***
40
41    // *** end your code ***
42
43    // ...
44 }
```

Requirements

- Function A has to be finished before functions B - E may be executed.
- Function E may only be executed after functions A - D are finished.
- Extract as much concurrency / parallelism as possible.

Insert code between lines 39 and 41 that implements the given requirements. **0 of 8 points**

Solution Path

```
auto a = std::async(function_A, std::ref(ds));
a.wait();

auto b = std::async(function_B, std::ref(ds));
auto c = std::async(function_C, std::ref(ds));
auto d = std::async(function_D, std::ref(ds));

b.wait(); c.wait(); d.wait();
auto e = std::async(function_E, std::ref(ds));

e.wait();
```

Q4

GPU

Briefly explain the term "warp" on the GPU as introduced in the lecture. **0 of 2 points**

Solution Path

A warp is a scheduling unit consisting of 32 threads. Threads in a warp share the same program counter and execute a single instruction simultaneously (in lockstep). The warp scheduler schedules units of warps and not threads.

Enter the four steps of the design cycle for porting an application to the GPU as introduced in the lecture. **0 of 1 point**

Step 1 — 0.25 points — --- ✗ Assess

Step 2 — 0.25 points — --- ✗ Parallelize

Step 3 — 0.25 points — --- ✗ Optimize

Step 4 — 0.25 points — --- ✗ Deploy

Name two typical compute kernels often found in Deep Learning (DL) applications as introduced in the lecture. **0 of 1 point**

Solution Path

convolutions, rectifiers, pooling, fully connected layers

Consider the following CUDA source code for the next question:

```
__global__ void kernel(image *a)
{
    a[threadIdx.x].r = 2.0 * a[threadIdx.x].r;
    a[threadIdx.x].g = 2.0 * a[threadIdx.x].g;
    a[threadIdx.x].b = 2.0 * a[threadIdx.x].b;
    if (a[threadIdx.x].r > 1.0)
        a[threadIdx.x].r = 1.0;
    if (a[threadIdx.x].g > 1.0)
        a[threadIdx.x].g = 1.0;
    if (a[threadIdx.x].b > 1.0)
        a[threadIdx.x].b = 1.0;
}

int main(int argc, char** argv) {
    int N = 1000000;
    struct pixel {
        float r;
        float g;
        float b;
    }
    struct pixel image[N];

    // More code left out for brevity

    // Call kernel on GPU

    // save image to file
}

```

Identify a performance issue when executing the code above on a GPU as introduced in the lecture. Explain briefly the performance issue in general on GPUs. **0 of 3 points**

Solution Path

Branch divergence due to the if conditions. There is no issue with coalescing since all data elements of the image are accessed.

Consider the following CUDA source code for the next question:

```
#include <stdio.h>
#include <stdlib.h>
// TODO: Kernel definition




int main(int argc, char** argv) {
  char *arg = argv[1]; int len = atoi(arg);
  float* a = (float *) malloc(len * sizeof(float));
  float* b = (float *) malloc(len * sizeof(float));
  float* c = (float *) malloc(len * sizeof(float));
  // Initialize array with random numbers on CPU
  srand(42);
  for (int i = 0; i < len; i++) {
    a[i] = rand();
    b[i] = rand();
    c[i] = rand();
  }
  // TODO: Memory management and kernel invocation




  for (int i = 0; i < len; i++)
    printf("%f ", c[i]);
}
```

Implement the calculation $c_i = \sqrt{(a_i - b_i)^2}$ with vectors a, b, c of len elements with CUDA-C such that it is fully executed on a GPU. Moreover, implement the required data transfers and the call of the kernel in the main function. Copy and complete the code stub above for your solution. You may use the default mathematical function sqrt(). You may assume that len is large and evenly divisible by the number of threads per block. Please use the typical block sizes as introduced in the lecture.

0 of 8 points

Solution Path

```
#include <stdio.h>
#include <stdlib.h>
__global__
void calc(float *a, float *b, float *c, int len) {
  int tid = blockDim.x * blockIdx.x + threadIdx.x;
  c[tid] = sqrt((a[tid] - b[tid]) * (a[tid] - b[tid]));
}

int main(int argc, char** argv) {
  char *arg = argv[1]; int len = atoi(arg);
  float* a = (float *) malloc(len * sizeof(float));
  float* b = (float *) malloc(len * sizeof(float));
  float* c = (float *) malloc(len * sizeof(float));

  // Initialize array with random numbers on CPU
  srand(42);
  int i;
  for (i = 0; i < len; i++) {
    a[i] = rand();
    b[i] = rand();
    c[i] = rand();
  }

  // Memory management and kernel invocation
  float* a_g; float* b_g; float* c_g;
  cudaMalloc(&a_g, sizeof(float) * len);
  cudaMalloc(&b_g, sizeof(float) * len);
  cudaMalloc(&c_g, sizeof(float) * len);
  cudaMemcpy(a_g, a, sizeof(float) * len, cudaMemcpyHostToDevice);
  cudaMemcpy(b_g, b, sizeof(float) * len, cudaMemcpyHostToDevice);
  calc<<<(len + 128 - 1)/128, 128>>>(a_g, b_g, c_g, len);
  cudaMemcpy(c, c_g, sizeof(float) * len, cudaMemcpyDeviceToHost);
  cudaFree(a_g);
  cudaFree(b_g);
  cudaFree(c_g);

  for (int i = 0; i < len; i++)
    printf("%f ", c[i]);
}
```

Q5

Distributed Memory: Questions

A BSP computer is characterized as a 4-tuple (p, g, l, r). Write down the meaning of each value. **0 of 2 points**

Solution Path

- p: number of processors
- g: communication costs per data word [s/word]
- l: latency [s]
- r: computation rate per processor [flop/s]

Define a cyclic distribution function $distr$ for a vector $x = (x_0, \ldots, x_{n-1})^T \in \mathbb{R}^n$ among $p > 0$ processors. The function should distribute the vector elements uniformly across the processors. The signature of $distr$ is $distr : \{0, \ldots, n-1\} \to \{0, \ldots, p-1\}$ where $distr(i) = k$ means that element x_i will be located at processor k. **0 of 2 points**

Solution Path

$$distr : \{0, \ldots, n-1\} \to \{0, \ldots, p-1\}$$
$$distr(i) = i \mod p \quad \forall\, i \in \{0, \ldots, n-1\}$$

Describe a main difference between the BSP model (with the Bulk implementation) and the PGAS model (with the DASH implementation) *in terms of communication*. **0 of 2 points**

Solution Path

In BSP, communication can only be performed in communication phases of a superstep. In PGAS, communication can be done everywhere in the program.

Other possible answer: In BSP, communication has to explicitly define the target rank, e.g. access element i of array a at rank t: $a(t)[i]$. In PGAS, accesses to an arbitrary array element without specifying the target rank is possible, the access is managed and translated to the correct target rank by the runtime (e.g., DASH)

Name and describe (in one sentence) one of the MPI collective operations. **0 of 2 points**

Solution Path

Different possibilites, e.g.

- MPI_Bcast(data, root), broadcast "data" from "root" to all other processes
- MPI_Scatter(data, subdata, root), distribute "data" from "root" into "subdata" in all processes

Q6

Distributed Memory: Inner Product in a Ring

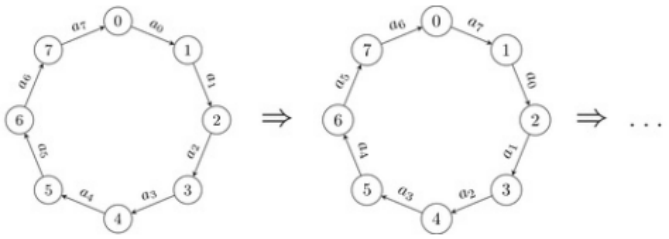
A given number of $p > 1$ processors from 0 to $p - 1$ is arranged in a ring network topology (see figure below).
In the following, the inner product calculation algorithm of the lecture should be adapted to utilize this ring structure:

- The input is given as two vectors $x = (x_0, \ldots, x_{n-1})^T, y = (y_0, \ldots, y_{n-1})^T \in \mathbb{R}^n$.
- The inner product can be calculated as $k = x^T y = \sum_{i=0}^{n-1} x_i y_i$.
- The elements of the vectors x and y are distributed with a *cyclic* distribution function to the n processes.

At the end of the algorithm, the inner product k should be available at **all** processors. Consider the following communication constraints:

- Each processor n should put its data to its right (clockwise) neighbor (e.g., 0 should put to 1, 1 should put to 2, ...).
- No processor is allowed to put data to any other processor than the right neighbor.

The figure below shows how the computation should work: In each step, all processors put their local inner product value to the right neighbor. After the values from the neighbors have been received, each processor merges the received value with its local product value **and** puts the received value to its right neighbor and so on.



Sketch of the first two steps of the ring exchange for the inner product calculation for $p = 8$, a_i stands for the local inner product value of processor i. In the first step, all processors i send their inner local product a_i to the right neighbor. In the next step, all processors put the received inner product of their left neighbor to the right neighbor and so on.

In the following subtasks, a BSP algorithm is developed step by step using the skeleton below:

```
// Algorithm for processor s in {0,...,p-1}
// TODO: compute_local_product

exchange_product := loc_product; // initialize exchange_product with computed local product
for (i := 0; i < num_steps; i++) {
    // TODO: ring_exchange_step
}

// final inner product available at all processors in variable "loc_product"
```

Assuming $p > 1$ processors, give the number of required ring exchange steps (`num_steps`) that will be needed to compute the inner product. **0 of 2 points**

2 points

The number of required exchange steps is...

--- ✗ p-1

The *local inner product* has to be calculated on every processor s first. Write down the BSP pseudocode for the placeholder `compute_local_product`. The result should be stored in the local variable `loc_product`. Use a `for` loop to iterate over the assigned elements. Reconsider that the vectors x and y are distributed in a cyclic way and use *global indexing* for the accesses to x and y (as done in the lecture). **0 of 3 points**

Solution Path

```
loc_product := 0;
for (i := s; i < n; i += p) do
    loc_product := loc_product + x_i * y_i;
```

Now assume that the local product at processor `s` has been computed and stored in `loc_product`.

You can use the BSP communication primitive `put 'my_var' in P(t)` to put the value of `my_var` at processor s to the same variable `my_var` at processor t. The variable `exchange_product` (see algorithm skeleton) might be helpful for that.

Note: Assume that after the `put` call the new value for `my_var` will be buffered at target process t first (and not directly updated). Only after the next `barrier()` is called and finished, the new value for `my_var` is written and visible at t.

Define the BSP pseudocode of the for-loop and implement the placeholder `ring_exchange_step`. For that, your code should put the values to the correct target process and compute the global inner product in `num_steps` iterations. **Hint:** Use the process identifier `s` in your code, think about how to calculate a clockwise neighbor. Do not forget to place a `barrier` at the right position.

0 of 6 points

Solution Path

Variant 1 (just forward received value to the right neighbor and accumulate in `loc_product`, then the final value is in `loc_product`):

```
for (i := 0; i < num_steps; i++) {
    // send current exchange product at process s to right neighbor
    put exchange_product to P((s+1) mod p);
    barrier(); // communication done, new exchange_product received
    // add exchange_product to local_product
    loc_product := loc_product + exchange_product;
    // next step...
}
```

Variant 2 (accumulate values in `exchange_product` and forward it to right neighbor, the final value is in `exchange_product`)

```
for (i := 0; i < num_steps; i++) {
    // send current exchange product at process s to right neighbor
    put exchange_product to P((s+1) mod p);
    barrier(); // communication done, new exchange_product received
    // add local_product to exchange_product
    exchange_product := exchange_product + loc_product;
    // next step...
}
```

Q7

Distributed Memory: BSP Cost Calculation

Assume $p > 1$ processors and a vector $m = (m_0, \ldots, m_{p-1}) \in \mathbb{R}^p$ that will be used to communicate data. Each processor $s \in \{0, \ldots, p-1\}$ has exactly one local value m_s of the corresponding distributed array m.
In pseudocode, a value m_s is referenced with the term `m(s)`.

Consider the following BSP algorithm for $s \in \{0, \ldots, p-1\}$:

```
if s == 0 then
    barrier();
    result = 0;
    for (i := 1; i < p; i++) do
        result += m(i);
else
    m(s) = s * s;
    put m(s) in P(0);
    barrier();
fi
barrier();
```

Provide a formula for the calculation result that is finally stored in the variable `result`. The formula should only depend on p, not on m. **0 of 2 points**

Solution Path

`result` $= \sum_{i=1}^{p-1} i^2$

Provide the number of supersteps in the algorithm. **0 of 1 point**

1 point

The number of supersteps is...

--- ✗ 2

Use the BSP cost model to calculate the costs T_{alg} of this algorithm.

Remember the definition of T_i for the costs of superstep i and the definition of the h-relation:

$$T_i := \frac{1}{r} \cdot \max_{0 \le s < p} w_i^{(s)} + g h_i + l$$

$$h_i := \max_{0 \le s < p} \max\{r_i^{(s)}, t_i^{(s)}\}.$$

Provide h_i for all supersteps i. Note: It is enough to provide the final numerical values only. **0 of 2 points**

Solution Path

$h_0 = p - 1$
$h_1 = 0$

Provide $f_i := \max_{0 \le s < p} w_i^{(s)}$ for all supersteps i. Note: It is enough to provide the final numerical values only. **0 of 2 points**

Solution Path

$f_0 := \max_{0 \le s < p} w_0^{(s)} = 1$
$f_1 := \max_{0 \le s < p} w_1^{(s)} = p - 1$

Finally provide the resulting T_{alg} assuming $p > 1$ processors. **0 of 2 points**

Solution Path

$T_0 = \frac{1}{r} + g(p-1) + l$
$T_1 = \frac{p-1}{r} + l$
$T_{alg} = T_0 + T_1 = \frac{p}{r} + g(p-1) + 2l$

Q8

Parallel I/O: Questions

Name two parallel I/O schemes. Select one property and compare your given schemes (advantage / disadvantage). **0 of 3 points**

Solution Path

I/O schemes: Centralized I/O, Task-Local I/O, Shared File I/O, Multi-File Shared I/O

For the comparison, different answers are possible, details are in the lecture slides.

Explain (in two sentences) the main architectural difference in the parallel file system of a traditional supercomputer using Lustre and a big data cluster using HDFS. **0 of 2 points**

Solution Path

HPC-System: Separate storage and compute nodes.
HDFS big data cluster: No separation, each node has locally attached storage (data locality).

Q9

MapReduce Questions

The Hadoop Filesystem uses HDFS blocks and other mechanisms to achieve fault tolerance. Explain how HDFS detects a failed DataNode and what happens in that case (in up to three sentences). **2 of 3 points**

Solution Path

The DataNode sends periodically heartbeats to the NameNode. If there has no hearbeat been sent to the NameNode before a certain timeout is reached, the DataNode is marked as dead, so no further requests are sent to the node. If the replication factor for some HDFS blocks is below the specified value, then those blocks are re-replicated on other DataNodes.

A *straggler* is critical for the total execution time of a MapReduce job. Explain (in one sentence) what a straggler is. Then, name and explain (in one or two additional sentences) the mechanism of Hadoop MapReduce which reduces the impact of stragglers on the total execution time. **2 of 3 points**

Solution Path

A straggler is a machine taking unusually long time to complete (one of the last) map or reduce tasks.

Hadoop implements *speculative execution* to reduce the impact of stragglers: When a job is close to completion, detect tasks with much less progress than others and launch duplicate task for those on another machine to reduce total execution time.

Apache Hadoop extends the MapReduce model by a `Combiner` function. Write down its *general* type signature. Then, explain how the `Combiner` function can speed up job executions in general (in one sentence). **0 of 3 points**

Solution Path

General type signature: $Combine(k, list(v)) \rightarrow list(k, v)$

Aggregate (if possible) map output before shuffling and reduction to reduce amount of data transferred between map and reduce tasks (less network traffic).

Out of a large set of temperature values, you want to determine the 20 largest temperature values. Name the MapReduce design pattern that fits best to that computation. **0 of 1 point**

Solution Path

Top Ten Pattern / Top K Pattern

Q10

MapReduce Application: Average Total Order Weights

You have a large set of packaging data from a logistics center as key-value pairs. Each key-value pair *(customer, item weights)* represents an order where

- *customer* is the key of type `String` and represents a *unique* customer name and
- *item weights* is the value as list of integers (`List<Integer>`) containing the invidual weights of the ordered items (in grams).

Each customer can have made multiple orders. Thus, there can be multiple key-value pairs with the same customer as key. Consider the following key-value pairs:

- (Bob, [200, 100])
- (Alice, [100, 100, 50])
- (Bob, [100, 200, 600])
- (Bob, [1000, 200, 300])
- (Alice, [200, 150, 150])
- (Carol, [400, 350, 50])

You should determine for each customer the **average total order weight** in grams. The *total order weight* is the sum of the item weights of a *single* order. For example, the total order weight of the first key-value pair *(Bob, [200, 100])* is

200 + 100 = 300.

In order to compute the average total order weight of a customer, you have to compute the arithmetic mean of the total order weights of all his/her orders. Finally, for each customer *c*, the MapReduce computation should output a key-value pair *(c, average total order weight of c)*.

To solve this problem, you have to define and perform a MapReduce computation in the following subtasks. As described above, the inputs of the `Map()` function are key-value pairs of the form *(customer, item weights)*.
Note: A Combiner is **not** used for the computation.

Give the *specific* type signatures of the required `Map()` and `Reduce()` function to compute the average total order weight in grams for each customer. **0 of 3 points**

Solution Path

```
Map(String, List<Integer>) -> List<String, Integer>
Reduce(String, List<Integer>) -> List<String, Float>
```

Give the pseudocode of the `Map()` and `Reduce()` function to compute the average total order weight in grams for each customer. **3 of 8 points**

Solution Path

```
Map(String customer, List<Integer> item_weights):
    total_weight = 0
    for weight in item_weights:
        total_weight += weight
    Emit(customer, total_weight)
```

```
Reduce(String customer, List<Integer> total_weights):
    int count = 0
    int total_weight_sum = 0
    for total_weight in total_weights:
        count++
        total_weight_sum += total_weight
    Emit(customer, ((float) total_weight_sum) / count)
```

Reconsider the six input key-value pairs:

- (Bob, [200, 100])
- (Alice, [100, 100, 50])
- (Bob, [100, 200, 600])
- (Bob, [1000, 200, 300])
- (Alice, [200, 150, 150])
- (Carol, [400, 350, 50])

Perform the MapReduce computation on the six key-value pairs in the following steps:

1. Apply the defined `Map()` function on each input key-value pair.
2. Group the outputs by key.
3. Apply the defined `Reduce()` function on the groups.

Hint: It is sufficient to write down the results of each step.

Perform the MapReduce computation on the six key-values pairs according to your defined pseudocode. **0 of 6 points**

Solution Path

Map output:

```
Map(Bob,   [200, 100]) = [(Bob, 300)]
Map(Alice, [100, 100, 50])  = [(Alice,  250)]
Map(Bob,   [100, 200, 600]) = [(Bob, 900)]
Map(Bob,   [1000, 200, 300]) = [(Bob,  1500)]
Map(Alice, [200, 150, 150]) = [(Alice,  400)]
Map(Carol, [400, 350, 50]) = [(Carol,  800)]
```

Grouped output:

```
(Alice, [250, 400])
(Bob,   [300, 900, 1500])
(Carol, [800])
```

Reduce output:

```
Reduce(Alice,  [250, 400]) = [(Alice, 325.0)]
Reduce(Bob,   [300, 900, 1500]) = [(Bob, 900.0)]
Reduce(Carol, [800]) = [(Carol, 800.0)]
```

Name the MapReduce design pattern that fits best to the average total order weights computation. **0 of 1 point**

Solution Path

Numerical Summarization

Q11

Apache Spark

Apache Spark introduces a data structure called *RDD*. Explain (in one sentence) the main advantage of using RDDs for *iterative* algorithms compared to Hadoop MapReduce. **0 of 2 points**

Solution Path

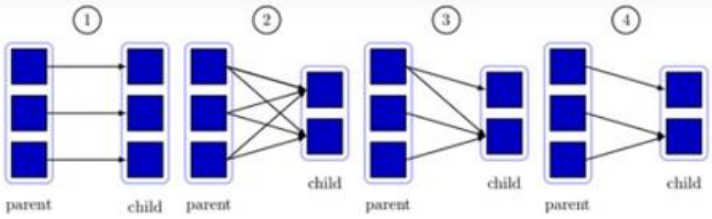
Caching of RDD partitions in memory to reuse data. (This is not possible in Hadoop MapReduce, where each job reads and writes input and output data to and from disk.)

Apache Spark introduces *lineage graphs* for RDDs. Explain what the vertices and edges represent. Then, explain what the lineage graph is used for. **0 of 2 points**

Solution Path

Vertices: RDDs / RDD objects
Edges: RDD transformations / Dependencies of the RDDs

A lineage graph stores for a given RDD how it can be derived out of other RDDs (stores the "history" of an RDD). It allows Apache Spark to postpone the actual computation of an RDD to a later point in time.



RDD Dependencies

State for the depicted RDD dependencies (1 - 4) whether they are wide or narrow. The boxes filled in blue represent the partitions of the RDDs. An arrow from the partition of a parent RDD to a partition of a child RDD indicates a dependency between those partitions, i.e., the corresponding parent partition is used to compute the child partition. *Fill out the blanks below with "narrow" or "wide".*

0.5 of 2 points

RDD dependency 1 is...

narrow ✔

0.5 points

RDD dependency 2 is...

--- ✖ wide

0.5 points

RDD dependency 3 is...

--- ✖ wide

0.5 points

RDD dependency 4 is...

--- ✖ narrow

Assume you have two co-partitioned RDDs A and B which are joined through a *join* transformation: `AB = A.join(B)` Explain whether the new RDD AB has a narrow dependency or a wide dependency on each of its parent RDDs A and B. Justify your answer.

0 of 2 points

Solution Path

AB has a narrow dependency on both parent RDDs A and B. Since both RDDs are co-partitioned, same keys of the key-value pairs are in the same partitions in A and B, so the join can take place without shuffling data between all partitions.