jan–feb 2026, subject no. 3

1. (3p) Consider the following excerpt from a program that is supposed to merge-sort a vector. The function `worker()` is called in all processes except process 0, the function `mergeSort()` is called from process 0 (and from the places described in this excerpt), the function `mergeSortLocal()` sorts the specified vector and the function `mergeParts()` merges two sorted adjacent vectors, given the pointer to the first element, the total length and the length of the first vector.

```
1  void mergeSort(int* v, int dataSize, int myId, int nrProc) {
2     if(dataSize <= 1) {
3        mergeSortLocal(v, dataSize);
4     } else {
5        int halfProc = (nrProc+1) / 2;
6        int child = myId+halfProc;
7        int halfSize = (dataSize+1) / 2;
8        MPI_Ssend(&halfProc, 1, MPI_INT, child, 1, MPI_COMM_WORLD);
9        MPI_Ssend(&halfSize, 1, MPI_INT, child, 2, MPI_COMM_WORLD);
10       MPI_Ssend(v, halfSize, MPI_INT, child, 3, MPI_COMM_WORLD);
11       mergeSort(v+halfSize, dataSize/2, myId, halfProc);
12       MPI_Recv(v, halfSize, MPI_INT, child, 4, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
13       mergeParts(v, dataSize, halfSize);
14    }
15 }
16 void worker(int myId) {
17    MPI_Status status;
18    int dataSize, nrProc;
19    MPI_Recv(&nrProc, 1, MPI_INT, MPI_ANY_SOURCE, 1, MPI_COMM_WORLD, &status);
20    auto parent = status.MPI_SOURCE;
21    MPI_Recv(&dataSize, 1, MPI_INT, parent, 2, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
22    std::vector v(dataSize);
23    MPI_Recv(v.data(), dataSize, MPI_INT, parent, 3, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
24    mergeSort(v.data(), dataSize, myId, nrProc);
25    MPI_Ssend(v.data(), dataSize, MPI_INT, parent, 4, MPI_COMM_WORLD);
26 }
```

Which of the following issues are present? Describe the changes needed to solve them.

A: some worker processes are not used if the number of processes is not a power of 2.

B: the application can deadlock if the length of the vector is smaller than the number of MPI processes.

C: the application can deadlock if the number of processes is not a power of 2.

D: the application can produce a wrong result if the input vector size is not a power of 2.

feb 2022, subject no. 1

1. (3p) Consider the following excerpt from a program that is supposed to merge-sort a vector. The function **`worker()`** is called in all processes except process 0, the function **`mergeSort()`** is called from process 0 (and from the places described in this excerpt), the function **`mergeSortLocal()`** sorts the specified vector and the function **`mergeParts()`** merges two sorted adjacent vectors, given the pointer to the first element, the total length and the length of the first vector.

```
1  void mergeSort(int* v, int dataSize, int myId, int nrProc) {
2     if(nrProc == 1 || dataSize <= 1) {
3        mergeSortLocal(v, dataSize);
4     } else {
5        int halfLen = dataSize / 2;
6        int halfProc = nrProc / 2;
7        int child = myId+halfProc;
8        MPI_Ssend(&halfLen, 1, MPI_INT, child, 1, MPI_COMM_WORLD);
9        MPI_Ssend(&halfProc, 1, MPI_INT, child, 2, MPI_COMM_WORLD);
10       MPI_Ssend(v, halfSize, MPI_INT, child, 3, MPI_COMM_WORLD);
11       mergeSort(v+halfSize, dataSize-halfSize, myId, nrProc);
12       MPI_Recv(v, halfSize, MPI_INT, child, 4, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
13       mergeParts(v, dataSize, halfSize);
14    }
15 }
16 void worker(int myId) {
17    MPI_Status status;
18    int dataSize, nrProc;
19    MPI_Recv(&dataSize, 1, MPI_INT, MPI_ANY_SOURCE, 1, MPI_COMM_WORLD, &status);
20    auto parent = status.MPI_SOURCE;
21    MPI_Recv(&nrProc, 1, MPI_INT, parent, 2, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
22    std::vector v(dataSize);
23    MPI_Recv(v.data(), dataSize, MPI_INT, parent, 3, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
24    mergeSort(v.data(), dataSize, myId, nrProc);
25    MPI_Ssend(v.data(), dataSize, MPI_INT, parent, 3, MPI_COMM_WORLD);
26 }
```

Which of the following issues are present? Describe the changes needed to solve them.

A: the application can deadlock if the length of the vector is smaller than the number of MPI processes.

B: the application can produce a wrong result if the input vector size is not a power of 2.

C: some worker processes are not used if the number of processes is not a power of 2.

D: the application can deadlock if the number of processes is not a power of 2.

2. (3p) Consider the following code for implementing a future mechanism (the `set()` function is guaranteed to be called exactly once by the user code)

```
1  template<typename T>
2  class Future {
3      T val;
4      bool hasValue;
5      mutex mtx;
6      condition_variable cv;
7  public:
8      Future() :hasValue(false) {}
9      void set(T v) {
10         cv.notify_all();
11         unique_lock<mutex> lck(mtx);
12         hasValue = true;
13         val = v;
14     }
15     T get() {
16         unique_lock<mutex> lck(mtx);
17         while(!hasValue) {
18             cv.wait(lck);
19         }
20         return value;
21     }
22 };
```

Which of the following are true? Give a short explaination.

A: `[issue]` a call to get() can deadlock if simultaneous with the call to set() T

B: `[issue]` a call to get() can deadlock if called after set() F

C: `[issue]` a call to get() can return an uninitialized value if simultaneous with the call to set() F

D: `[issue]` simultaneous calls to get() and set() can make future calls to get() deadlock

E: `[issue]` a call to get() can deadlock if called before set()

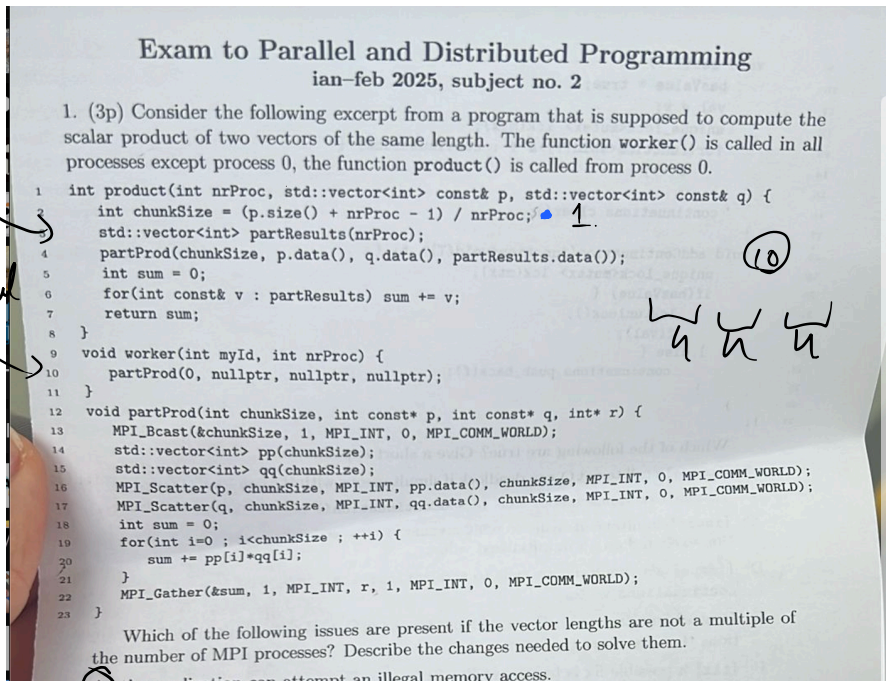
F: `[fix]` a possible fix is to remove the line 11

G: `[fix]` a possible fix is to interchange lines 12 and 13

H: `[fix]` a possible fix is to reorder lines 10–13 in the order 11, 13, 12, 10

I: `[fix]` a possible fix is to interchange lines 10 and 11

J: `[fix]` a possible fix is to unlock the mutex just before line 18 and to lock it back just afterwards



Exam to Parallel and Distributed Programming

ian–feb 2025, subject no. 2

1. (3p) Consider the following excerpt from a program that is supposed to compute the scalar product of two vectors of the same length. The function `worker()` is called in all processes except process 0, the function `product()` is called from process 0.

```
1  int product(int nrProc, std::vector<int> const& p, std::vector<int> const& q) {
2      int chunkSize = (p.size() + nrProc - 1) / nrProc;
3      std::vector<int> partResults(nrProc);
4      partProd(chunkSize, p.data(), q.data(), partResults.data());
5      int sum = 0;
6      for(int const& v : partResults) sum += v;
7      return sum;
8  }
9  void worker(int myId, int nrProc) {
10     partProd(0, nullptr, nullptr, nullptr);
11 }
12 void partProd(int chunkSize, int const* p, int const* q, int* r) {
13     MPI_Bcast(&chunkSize, 1, MPI_INT, 0, MPI_COMM_WORLD);
14     std::vector<int> pp(chunkSize);
15     std::vector<int> qq(chunkSize);
16     MPI_Scatter(p, chunkSize, MPI_INT, pp.data(), chunkSize, MPI_INT, 0, MPI_COMM_WORLD);
17     MPI_Scatter(q, chunkSize, MPI_INT, qq.data(), chunkSize, MPI_INT, 0, MPI_COMM_WORLD);
18     int sum = 0;
19     for(int i=0 ; i<chunkSize ; ++i) {
20         sum += pp[i]*qq[i];
21     }
22     MPI_Gather(&sum, 1, MPI_INT, r, 1, MPI_INT, 0, MPI_COMM_WORLD);
23 }
```

Which of the following issues are present if the vector lengths are not a multiple of the number of MPI processes? Describe the changes needed to solve them.

p = [10, 20, 30, 40]
proc = 3
(4+3-1) / 3
= 2

2 2 2

p.size 1
nrProc 10

A: the application can attempt an illegal memory access;
B: the application can deadlock.
C: some worker processes are not used.
D: some terms are added twice.
E: some terms are not added at all.
F: the scalar product is incorrectly computed in some other way.

2. (3p) Consider the following code for enqueueing a continuation on a future (the `set()` function is guaranteed to be called exactly once by the user code):

```
1  template<typename T>
2  class Future {
3      list<function<void(T)> > continuations;
4      T val;
5      bool hasValue;
6      mutex mtx;
7  public:
8      Future() :hasValue(false) {}
9      void set(T v) {
10         unique_lock<mutex> lck(mtx);
11         hasValue = true;
12         val = v;
13         lck.unlock();
14         for(function<void(T)>& f : continuations) {
15             f(v);
16         }
17         continuations.clear();
18     }
19     void addContinuation(function<void(T)> f) {
20         if(hasValue) {
21             f(val);
22         } else {
23             unique_lock<mutex> lck(mtx);
24             continuations.push_back(f);
25         }
26     }
27 };
```

Which of the following are true? Give a short explaination.

A: [issue] a call to `set()` can deadlock if simultaneous with the call to `addContinuation()`;

B: [issue] two simultaneous calls to `addContinuation()` may deadlock;

C: [issue] simultaneous calls to `addContinuation()` and `set()` may lead to continuations that are executed twice;

D: [issue] two simultaneous calls to `addContinuation()` may lead to a corrupted `continuations` vector;

E: [issue] simultaneous calls to `addContinuation()` and `set()` may lead to continuations that are never executed;

F: [fix] a possible fix is to move the content of line 13 to between lines 17 and 18;

G: [fix] a possible fix is to move the content of line 23 to between lines 19 and 20;

H: [fix] a possible fix is to move the content of line 23 to between lines 19 and 20 and add an `unlock()` call between lines 20 and 21;

I: [fix] a possible fix is to move the content of line 13 to between lines 17 and 18 and to move line 23 between lines 19 and 20;

J: [fix] a possible fix is to move the content of lines 11 and 13 (in this order) to between lines 17 and 18;

2. (3p) Consider the following code for a queue with multiple producers and consumers.

The close() function is guaranteed to be called exactly once by the user code, and enqueue() will not be called after that. dequeue() is supposed to block if the queue is empty and to return an empty optional if the queue is closed and all the elements have been dequeued.

```
1  template<typename T>
2  class ProducerConsumerQueue {
3      list<T> items;
4      bool isClosed = false;
5      condition_variable cv;
6      mutex mtx;
7  public:
8      void enqueue(T v) {
9          unique_lock<mutex> lck(mtx);
10         items.push_back(v);
11         cv.notify_one();
12     }
13     optional<T> dequeue() {
14         unique_lock<mutex> lck(mtx);
15         while(items.empty() && !isClosed) {
16             cv.wait(lck);
17         }
18         lck.unlock();
19         if(!items.empty()) {
20             optional<T> ret(items.front());
21             items.pop_front();
22             return ret;
23         }
24         return optional<T>();
25     }
26     void close() {
27         unique_lock<mutex> lck(mtx);
28         isClosed = true;
29         cv.notify_all();
30     }
31 };
```

Which of the following are true? Give a short explaination.

A: [issue] a call to dequeue() can result data corruption or undefined behavior if simultaneous with the call to enqueue();

B: [issue] a call to dequeue() can deadlock if simultaneous with the call to enqueue();

C: [issue] two simultaneous calls to enqueue() may deadlock;

D: [issue] two simultaneous calls to dequeue() may result in corrupted items list;

E: [issue] two simultaneous calls to dequeue() may deadlock;

F: [fix] move the content of line 14 to between lines 15 and 16 and line 18 between 16 and 17

G: [fix] eliminate lines 14 and 18

H: [fix] remove line 18 and insert copies of it between lines 21 and 22 and between 23 and 24

I: [fix] unlock the mutex before line 16 and lock it back after line 16

J: [fix] apply fix H above and, additionally, move line 14 in place of line 18