

Exam to Parallel and Distributed Programming

feb 2023, 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-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(&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 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); // statement 1
15        while(items.empty() && !isClosed) {
16            // place 1
17            cv.wait(lck);
18            // place 2
19        }
20        lck.unlock(); // statement 2
21        if(!items.empty()) {
22            optional<T> ret(items.front());
23            items.pop_front();
24            // place 3
25            return ret;
26        }
27        // place 4
28        return optional<T>();
29    }
30    void close() {
31        unique_lock<mutex> lck(mtx);
32        isClosed = true;
33        cv.notify_all();
34    }
35 };

```

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

- A: [issue] a call to dequeue() can deadlock if simultaneous with the call to enqueue();
 - B: [issue] two simultaneous calls to dequeue() may deadlock;
 - C: [issue] two simultaneous calls to enqueue() may deadlock;
 - D: [issue] two simultaneous calls to dequeue() may result in corrupted items list;
 - E: [issue] a call to dequeue() can result data corruption or undefined behavior if simultaneous with the call to enqueue();
 - F: [fix] move line marked statement 1 in the place marked place 1 and statement 2 in place 2
 - G: [fix] eliminate lines marked statement 1 and statement 2
 - H: [fix] remove line marked statement 2 and insert copies of it in placed marked place 3 and place 4
 - I: [fix] insert a statement unlocking the mutex in the place marked place 1 and lock it back in place 2
 - J: [fix] remove line marked statement 2 and insert copies of it in placed marked place 3 and place 4, and then move statement 1 in the place where statement 2 was
3. (3p) Write a parallel program that computes the sum of all elements in a matrix. It shall use a binary tree for computing the sum.

Exam to Parallel and Distributed Programming

feb 2023, subject no. 2

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.

```
void mergeSort(int* v, int dataSize, int myId, int nrProc) {
    if(nrProc == 1) {
        mergeSortLocal(v, dataSize);
    } else {
        int halfLen = dataSize / 2;
        int halfProc = (nrProc+1) / 2;
        int child = myId+halfProc;
        MPI_Ssend(&halfLen, 1, MPI_INT, child, 1, MPI_COMM_WORLD);
        MPI_Ssend(&halfProc, 1, MPI_INT, child, 2, MPI_COMM_WORLD);
        MPI_Ssend(v, halfSize, MPI_INT, child, 3, MPI_COMM_WORLD);
        mergeSort(v+halfSize, halfSize, myId, nrProc-halfProc);
        MPI_Recv(v, halfSize, MPI_INT, child, 4, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        mergeParts(v, dataSize, halfSize);
    }
}

void worker(int myId) {
    MPI_Status status;
    int dataSize, nrProc;
    MPI_Recv(&dataSize, 1, MPI_INT, MPI_ANY_SOURCE, 1, MPI_COMM_WORLD, &status);
    auto parent = status.MPI_SOURCE;
    MPI_Recv(&nrProc, 1, MPI_INT, parent, 2, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    std::vector v(dataSize);
    MPI_Recv(v.data(), dataSize, MPI_INT, parent, 3, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    mergeSort(v.data(), dataSize, myId, nrProc);
    MPI_Ssend(v.data(), dataSize, MPI_INT, parent, 3, MPI_COMM_WORLD);
}
```

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 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        // place 1
15        while(true) {
16            // place 2
17            if(!items.empty()) {
18                // place 3
19                optional<T> ret(items.front());
20                items.pop_front();
21                return ret;
22            }
23            if(isClosed) {
24                return optional<T>();
25            }
26            unique_lock<mutex> lck(mtx); // statement 1
27            cv.wait(lck);
28        }
29    }
30    void close() {
31        unique_lock<mutex> lck(mtx);
32        isClosed = true;
33        cv.notify_all();
34    }
35 };

```

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

- A: [issue] a call to `dequeue()` can deadlock if simultaneous with the call to `enqueue()`;
- B: [issue] two simultaneous calls to `dequeue()` may deadlock;
- C: [issue] two simultaneous calls to `enqueue()` may deadlock;
- D: [issue] two simultaneous calls to `dequeue()` may result in corrupted `items` list;
- E: [issue] a call to `dequeue()` can result data corruption or undefined behavior if simultaneous with the call to `enqueue()`;
- F: [fix] move line marked statement 1 in the place marked place 1
- G: [fix] move line marked statement 1 in the place marked place 2
- H: [fix] move line marked statement 1 in the place marked place 3
- I: [fix] eliminate line marked statement 1
- J: [fix] put a copy of the line marked statement 1 in the place marked place 3

3. (3p) Write a parallel program that computes the sum of all elements in a matrix. It shall use a binary tree for computing the sum.

Exam to Parallel and Distributed Programming

feb 2023, subject no. 3

1. (3p) Consider the following excerpt from a program that is supposed to compute the product of two non-zero polynomials. The function `worker()` is called in all processes except process 0, the function `product()` is called from process 0. The polynomials are represented with the coefficient for degree 0 at index 0 in the vector.

```
1 void product(int nrProc, std::vector<int> const& p, std::vector<int> const& q,
2   std::vector<int>& r) {
3     int sizes[2]; sizes[0] = p.size(); sizes[1] = q.size();
4     MPI_Bcast(sizes, 2, MPI_INT, 0, MPI_COMM_WORLD);
5     MPI_Bcast(const_cast<int*>(p.data()), p.size(), MPI_INT, 0, MPI_COMM_WORLD);
6     MPI_Bcast(const_cast<int*>(q.data()), q.size(), MPI_INT, 0, MPI_COMM_WORLD);
7     std::vector<int> partRes;
8     partProd(0, nrProc, p, q, partRes);
9     r.resize(p.size()+q.size()-1);
10    MPI_Gather(partRes.data(), partRes.size(), MPI_INT,
11      r.data(), partRes.size(), MPI_INT, 0, MPI_COMM_WORLD);
12  }
13 void worker(int myId, int nrProc) {
14   int sizes[2];
15   MPI_Bcast(sizes, 2, MPI_INT, 0, MPI_COMM_WORLD);
16   std::vector<int> p(sizes[0]);
17   std::vector<int> q(sizes[1]);
18   MPI_Bcast(p.data(), p.size(), MPI_INT, 0, MPI_COMM_WORLD);
19   MPI_Bcast(q.data(), q.size(), MPI_INT, 0, MPI_COMM_WORLD);
20   std::vector<int> r;
21   partProd(myId, nrProc, p, q, r);
22   MPI_Gather(r.data(), r.size(), MPI_INT, nullptr, 0, MPI_INT, 0, MPI_COMM_WORLD);
23 }
24 void partProd(int myId, int nrProc, std::vector<int> const& p, std::vector<int> const& q,
25   std::vector<int>& r) {
26   int chunkSize = (p.size()+q.size()-1) / nrProc;
27   r.resize(chunkSize, 0);
28   size_t baseIdx = chunkSize*myId;
29   for(int i=0; i<chunkSize; ++i) {
30     for(int j=0; j<=i+baseIdx; ++j) {
31       if(j<p.size() && i+baseIdx-j<q.size()) {
32         r[i] += p[j]*q[i+baseIdx-j];
33       }
34     }
35   }
36 }
```

Which of the following issues are present if the output degree plus one is not a multiple of the number of MPI processes? Describe the changes needed to solve them.

- A: the application can have memory corruption.
- B: the application can deadlock.
- C: some worker processes are not used.
- D: some coefficients are computed twice.
- E: some coefficients are not computed at all.
- F: some coefficients are computed incorrectly.

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 val;
21     }
22 };

```

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

- A: [issue] a call to `get()` can deadlock if simultaneous with the call to `set()`
- B: [issue] a call to `get()` can deadlock if called after `set()`
- C: [issue] a call to `get()` can return an uninitialized value if simultaneous with the call to `set()`
- 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

3. (3p) Write a parallel program that computes the scalar product of two vectors of the same length. (The scalar product is $a_0 \cdot b_0 + a_1 \cdot b_1 + \dots + a_{n-1} \cdot b_{n-1}$.) Use a binary tree for computing the sum.

Exam to Parallel and Distributed Programming
feb 2023, subject no. 4

1. (3p) Consider the following excerpt from a program that is supposed to compute the product of two non-zero polynomials. The function `worker()` is called in all processes except process 0, the function `product()` is called from process 0. The polynomials are represented with coefficient for degree 0 at index 0 in the vector.

```
1 void product(int nrProc, std::vector<int> const& p, std::vector<int> const& q,  
2   std::vector<int>& r) {  
3     int sizes[2]; sizes[0] = p.size(); sizes[1] = q.size();  
4     MPI_Bcast(sizes, 2, MPI_INT, 0, MPI_COMM_WORLD);  
5     MPI_Bcast(const_cast<int*>(p.data()), p.size(), MPI_INT, 0, MPI_COMM_WORLD);  
6     MPI_Bcast(const_cast<int*>(q.data()), q.size(), MPI_INT, 0, MPI_COMM_WORLD);  
7     std::vector<int> partRes;  
8     partProd(0, nrProc, p, q, partRes);  
9     r.resize(p.size()+q.size()-1);  
10    MPI_Gather(partRes.data(), partRes.size(), MPI_INT,  
11      r.data(), partRes.size(), MPI_INT, 0, MPI_COMM_WORLD);  
12  }  
13 void worker(int myId, int nrProc) {  
14   metadata[0] = nrProc;  
15   MPI_Bcast(sizes, 2, MPI_INT, 0, MPI_COMM_WORLD);  
16   std::vector<int> p(sizes[0]);  
17   std::vector<int> q(sizes[1]);  
18   MPI_Bcast(p.data(), p.size(), MPI_INT, 0, MPI_COMM_WORLD);  
19   MPI_Bcast(q.data(), q.size(), MPI_INT, 0, MPI_COMM_WORLD);  
20   std::vector<int> r;  
21   partProd(myId, nrProc, p, q, r);  
22   MPI_Gather(r.data(), r.size(), MPI_INT, nullptr, 0, MPI_INT, 0, MPI_COMM_WORLD);  
23 }  
24 void partProd(int myId, int nrProc, std::vector<int> const& p, std::vector<int> const& q,  
25   std::vector<int>& r) {  
26   int chunkSize = (p.size()+q.size()-1+nrProc-1) / nrProc;  
27   size_t baseIdx = chunkSize*myId;  
28   r.resize(chunkSize, 0);  
29   for(int i=0; i<chunkSize; ++i) {  
30     for(int j=0; j<=i+baseIdx; ++j) {  
31       if(j<p.size() && i+baseIdx-j<q.size()) {  
32         r[i] += p[j]*q[i+baseIdx-j];  
33       }  
34     }  
35   }  
36 }
```

Which of the following issues are present if the output degree plus one is not a multiple of the number of MPI processes? Describe the changes needed to solve them.

- A: the application can have memory corruption.
- B: the application can deadlock.
- C: some worker processes are not used.
- D: some coefficients are computed twice.
- E: some coefficients are not computed at all.
- F: some coefficients are computed incorrectly.

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         unique_lock<mutex> lck(mtx);
11         cv.notify_all();
12         hasValue = true;
13         val = v;
14     }
15     T get() {
16         while(!hasValue) {
17             unique_lock<mutex> lck(mtx);
18             cv.wait(lck);
19         }
20         return val;
21     }
22 };

```

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

- A: [issue] a call to `get()` can deadlock if simultaneous with the call to `set()`
 - B: [issue] a call to `get()` can deadlock if called after `set()`
 - C: [issue] a call to `get()` can return an uninitialized value if simultaneous with the call to `set()`
 - 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 move line 11 between line 13 and 14
 - G: [fix] a possible fix is to interchange lines 11 and 13
 - H: [fix] a possible fix is to remove line 17
 - I: [fix] a possible fix is to move line 17 between lines 15 and 16
 - J: [fix] a possible fix is to move line 17 between lines 15 and 16 and to unlock the mutex before line 18 (`wait()`) and lock it back afterwards
3. (3p) Write a parallel program that computes the scalar product of two vectors of the same length. (The scalar product is $a_0 \cdot b_0 + a_1 \cdot b_1 + \dots + a_{n-1} \cdot b_{n-1}$.) Use a binary tree for computing the sum.

Exam to Parallel and Distributed Programming
feb 2023, subject no. 5

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;
3     MPI_Bcast(&chunkSize, 1, MPI_INT, 0, MPI_COMM_WORLD);
4     std::vector<int> partResults(nrProc);
5     partProd(chunkSize, p.data(), q.data(), partResults.data());
6     int sum = 0;
7     for(int const& v : partResults) sum += v;
8     return sum;
9 }
10 void worker(int myId, int nrProc) {
11     int chunkSize;
12     MPI_Bcast(&chunkSize, 1, MPI_INT, 0, MPI_COMM_WORLD);
13     partProd(chunkSize, nullptr, nullptr, nullptr);
14 }
15 void partProd(int chunkSize, int const* p, int const* q, int* r) {
16     std::vector<int> pp(chunkSize);
17     std::vector<int> qq(chunkSize);
18     MPI_Scatter(p, chunkSize, MPI_INT, pp.data(), chunkSize, MPI_INT, 0, MPI_COMM_WORLD);
19     MPI_Scatter(q, chunkSize, MPI_INT, qq.data(), chunkSize, MPI_INT, 0, MPI_COMM_WORLD);
20     int sum = 0;
21     for(int i=0 ; i<chunkSize ; ++i) {
22         sum += pp[i]*qq[i];
23     }
24     MPI_Gather(r, 1, MPI_INT, &sum, 1, MPI_INT, 0, MPI_COMM_WORLD);
25 }
```

Which of the following issues are present if the output degree plus one is not a multiple of the number of MPI processes? Describe the changes needed to solve them.

- A: the application can have memory corruption.
- 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 explanation.

- 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;

3. (3p) Write a parallel algorithm that computes the product of 2 matrices. The program shall use a number of threads specified as an input.

Exam to Parallel and Distributed Programming

feb 2023, subject no. 6

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      MPI_Bcast(&chunkSize, 1, MPI_INT, 0, MPI_COMM_WORLD);
5      partProd(chunkSize, p.data(), q.data(), partResults.data());
6      int sum = 0;
7      for(int const& v : partResults) sum += v;
8      return sum;
9  }
10 void worker(int myId, int nrProc) {
11     int chunkSize;
12     MPI_Bcast(&chunkSize, 1, MPI_INT, 0, MPI_COMM_WORLD);
13     partProd(chunkSize, nullptr, nullptr, nullptr);
14 }
15 void partProd(int chunkSize, int const* p, int const* q, int* r) {
16     std::vector<int> pp(chunkSize);
17     std::vector<int> qq(chunkSize);
18     MPI_Scatter(p, chunkSize, MPI_INT, pp.data(), chunkSize, MPI_INT, 0, MPI_COMM_WORLD);
19     MPI_Scatter(q, chunkSize, MPI_INT, qq.data(), chunkSize, MPI_INT, 0, MPI_COMM_WORLD);
20     int sum = 0;
21     for(int i=0 ; i<chunkSize ; ++i) {
22         sum += pp[i]*qq[i];
23     }
24     MPI_Gather(r, 1, MPI_INT, &sum, 1, MPI_INT, 0, MPI_COMM_WORLD);
25 }
```

Which of the following issues are present if the output degree plus one is not a multiple of the number of MPI processes? Describe the changes needed to solve them.

- A: the application can have memory corruption.
- 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          hasValue = true;
11              val = v;
12          unique_lock<mutex> lck(mtx);
13          for(function<void(T)>& f : continuations) {
14              f(v);
15          }
16          continuations.clear();
17      }
18      void addContinuation(function<void(T)> f) {
19          unique_lock<mutex> lck(mtx);
20          if(hasValue) {
21              lck.unlock();
22              f(val);
23          } else {
24              continuations.push_back(f);
25          }
26      }
27 };

```

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

- ☒ 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 remove line 21;
- ☐ G: [fix] a possible fix is to move the content of line 12 to between lines 9 and 10;
- ☐ H: [fix] a possible fix is to move the content of line 12 to between lines 9 and 10 and add an `unlock()` in its place;
- ☐ I: [fix] a possible fix is to remove lines 19 and 21;
- ☒ J: [fix] a possible fix is to move the content of line 12 to between lines 9 and 10 and to remove line 21;

3. (3p) Write a parallel algorithm that computes the product of 2 matrices. The program shall use a number of threads specified as an input.

Exam to Parallel and Distributed Programming

feb 2023, subject no. 7

1. (3p) Consider the following excerpt from a program that is supposed to compute the sum of elements in a vector. 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& v) {
2     int chunkSize = v.size() / nrProc;
3     MPI_Bcast(&chunkSize, 1, MPI_INT, 0, MPI_COMM_WORLD);
4     return sumRec(chunkSize, v.data());
5 }
6 void worker(int myId, int nrProc) {
7     int chunkSize;
8     MPI_Bcast(&chunkSize, 1, MPI_INT, 0, MPI_COMM_WORLD);
9     sumRec(myId, nrProc, chunkSize, nullptr);
10 }
11 int sumRec(int myId, int nrProc, int chunkSize, int const* v) {
12     std::vector<int> vv(chunkSize);
13     MPI_Scatter(v, chunkSize, MPI_INT, vv.data(), chunkSize, MPI_INT, 0, MPI_COMM_WORLD);
14     int sum = 0;
15     for(int value : vv) {
16         sum += value;
17     }
18     int t = 0;
19     if(2*myId < nrProc) {
20         MPI_Recv(&t, 1, MPI_INT, 2*myId, 1, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
21         sum += t;
22     }
23     if(2*myId+1 < nrProc) {
24         MPI_Recv(&t, 1, MPI_INT, 2*myId+1, 1, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
25         sum += t;
26     }
27     if(myId != 0) {
28         MPI_Send(&sum, 1, MPI_INT, myId/2, 1, MPI_COMM_WORLD);
29     }
30     return sum;
31 }
```

Which of the following issues are present if the ~~output degree~~ plus one is not a multiple of the number of MPI processes? Describe the changes needed to solve them.

A: the application can have memory corruption.

B: the application can deadlock.

C: some worker processes are not used.

D: some terms may be added twice.

E: some terms may not be added at all.

F: the sum is incorrectly computed in some other way.

2. (3p) Consider the following code for a fixed size thread pool (the `close()` function is guaranteed to be called exactly once and no enqueues will happen then or afterwards):

```
1 class ThreadPool {
2     condition_variable cv;
3     mutex mtx;
4     list<function<void()>> work;
5     vector<thread> threads;
```

```

6     bool closed = false;
7     void run() {
8         unique_lock<mutex> lck(mtx);
9         while(true) {
10             if (!work.empty()) {
11                 function<void()> wi = work.front();
12                 work.pop_front();
13                 wi();
14             } else if (closed) {
15                 return;
16             } else {
17                 cv.wait(lck);
18             }
19         }
20     }
21 public:
22     explicit ThreadPool(int n) {
23         threads.reserve(n);
24         for(int i=0 ; i<n ; ++i) {
25             threads.emplace_back([this]() {run();});
26         }
27     }
28     void enqueue(function<void()> f) {
29         unique_lock<mutex> lck(mtx);
30         work.push_back(f);
31         cv.notify_one();
32     }
33     void close() {
34         unique_lock<mutex> lck(mtx);
35         closed = true;
36         cv.notify_all();
37         lck.unlock();
38         for(thread& th : threads) th.join();
39     }
40 };

```

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

- A: [issue] two simultaneous calls to enqueue() may deadlock;
- B: [issue] two simultaneous calls to enqueue() may lead to waking up a single thread even if multiple threads are sleeping;
- C: [issue] a work item may stay in the queue for a long time if enqueued simultaneously with the previous work item being finished;
- D: [issue] the work items are never executed in parallel;
- E: [issue] close() may deadlock;
- F: [fix] a fix is to add lck.unlock() just before line 13;
- G: [fix] a fix is to replace cv.notify_one() with cv.notify_all() in line 31;
- H: [fix] a fix is to add lck.unlock() just before line 13 and lck.lock() just after line 13;
- I: [fix] a fix is to add lck.unlock() between lines 30 and 31;
- J: [fix] a fix is to remove line 37;

3. (3p) Write a parallel algorithm that computes the scalar product of two vectors of the same length. The program shall use the number of threads given as an argument.

Exam to Parallel and Distributed Programming

jan-feb 2024, subject no. 1

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;
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 ^{input size} ~~output degree plus one~~ is not a multiple of the number of MPI processes? Describe the changes needed to solve them.

- 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         unique_lock
        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 explanation.

- 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;

3. (3p) Write a parallel algorithm that computes the product of 2 matrices. The program shall use a number of threads specified as an input.

Exam to Parallel and Distributed Programming

ian-feb 2024, 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.

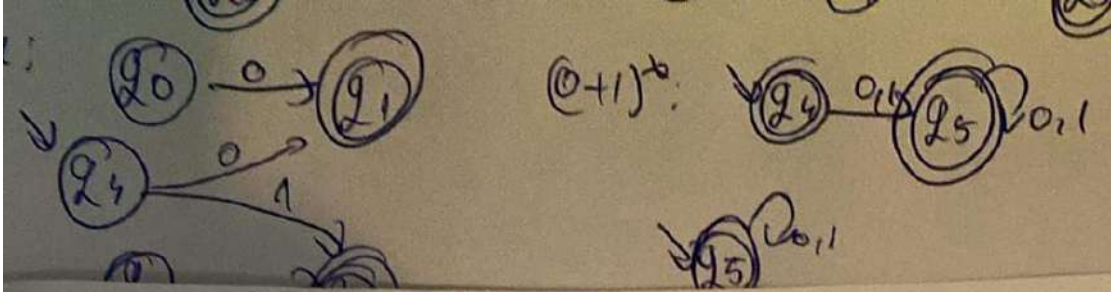
```
1 int product(int nrProc, std::vector<int> const& p, std::vector<int> const& q) {
2     int chunkSize = (p.size() + nrProc - 1) / nrProc;
3     std::vector<int> partResults(nrProc);
4     partProd(chunkSize, p.data(), q.data(), partResults.data());
5     int sum = 0;
6     for(int const& v : partResults) 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 ~~output degree plus one~~ ^{length} is not a multiple of the number of MPI processes? Describe the changes needed to solve them.

- 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         hasValue = true;
```

```

11     val = v;
12     unique_lock<mutex> lck(mtx);
13     for(function<void(T)>& f : continuations) {
14         f(v);
15     }
16     continuations.clear();
17 }
18 void addContinuation(function<void(T)> f) {
19     unique_lock<mutex> lck(mtx);
20     if(hasValue) {
21         lck.unlock();
22         f(val);
23     } else {
24         continuations.push_back(f);
25     }
26 }
27 };

```

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

- 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 remove line 21;
- G: [fix] a possible fix is to move the content of line 12 to between lines 9 and 10;
- H: [fix] a possible fix is to move the content of line 12 to between lines 9 and 10 and add an `unlock()` in its place;
- I: [fix] a possible fix is to remove lines 19 and 21;
- J: [fix] a possible fix is to move the content of line 12 to between lines 9 and 10 and to remove line 21;

3. (3p) Write a parallel algorithm that computes the product of 2 matrices. The program shall use a number of threads specified as an input.

Exam to Parallel and Distributed Programming

jan-feb 2024, subject no. 3

1. (3p) Consider the following excerpt from a program that is supposed to compute the product of two non-zero polynomials. The function `worker()` is called in all processes except process 0, the function `product()` is called from process 0. The polynomials are represented with the coefficient for degree 0 at index 0 in the vector.

```

1 void product(int nrProc, std::vector<int> const& p, std::vector<int> const& q,
2   std::vector<int>& r) {
3   int sizes[2]; sizes[0] = p.size(); sizes[1] = q.size();
4   MPI_Bcast(sizes, 2, MPI_INT, 0, MPI_COMM_WORLD);
5   MPI_Bcast(const_cast<int*>(p.data()), p.size(), MPI_INT, 0, MPI_COMM_WORLD);
6   MPI_Bcast(const_cast<int*>(q.data()), q.size(), MPI_INT, 0, MPI_COMM_WORLD);
7   std::vector<int> partRes;
8   partProd(0, nrProc, p, q, partRes);
9   r.resize(nrProc*partRes.size());
10  MPI_Gather(partRes.data(), partRes.size(), MPI_INT,
11    r.data(), partRes.size(), MPI_INT, 0, MPI_COMM_WORLD);
12  r.resize(p.size()+q.size()-1);
13 }
14 void worker(int myId, int nrProc) {
15   int sizes[2];
16   MPI_Bcast(sizes, 2, MPI_INT, 0, MPI_COMM_WORLD);
17   std::vector<int> p(sizes[0]);
18   std::vector<int> q(sizes[1]);
19   MPI_Bcast(p.data(), p.size(), MPI_INT, 0, MPI_COMM_WORLD);
20   MPI_Bcast(q.data(), q.size(), MPI_INT, 0, MPI_COMM_WORLD);
21   std::vector<int> r;
22   partProd(myId, nrProc, p, q, r);
23   MPI_Gather(r.data(), r.size(), MPI_INT, nullptr, 0, MPI_INT, 0, MPI_COMM_WORLD);
24 }
25 void partProd(int myId, int nrProc, std::vector<int> const& p, std::vector<int> const& q,
26   std::vector<int>& partRes) {
27   int chunkSize = (p.size()+q.size()+nrProc-2) / nrProc;
28   r.resize(chunkSize, 0);
29   for(int i=0; i<chunkSize; ++i) {
30     for(int j=0; j<=i; ++j) {
31       if(j<p.size() && i-j<q.size()) {
32         r[i] += p[j]*q[i-j];
33       }
34     }
35   }
36 }

```

CS = N+1

trapitty trap

} OK

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

- A: the application can make an illegal memory access.
- B: the application can deadlock.
- C: some worker processes are not used.
- ☒ D: some coefficients are not computed at all.
- ☒ E: some coefficients are computed incorrectly.

2. (3p) Consider the following code for implementing a countdown event mechanism, that is, a `wait()` call waits until `cnt` calls to `done()` are made from the creation of the `CountdownEvent` object.

s3

```

1  template<typename T>
2  class CountdownEvent {
3      std::atomic<unsigned> count;
4      mutex mtx;
5      condition_variable cv;
6  public:
7      explicit CountdownEvent(unsigned cnt)
8          :count(cnt) {}
9      void done() {
10         unsigned old = count.fetch_sub(1);
11         if(old == 1) {
12             cv.notify_one();
13         }
14     }
15     T wait() {
16         if(count == 0) return;
17         unique_lock<mutex> lck(mtx);
18         while(count > 0) {
19             cv.wait(lck);
20         }
21         return val;
22     }
23 };

```

lock (mutex) (handwritten note with arrow pointing to line 17)

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

- ☒ A: [issue] a single call to wait() can wait forever if simultaneous with the last call to done()
- ☒ B: [issue] a call to wait() can wait forever if called after the last done()
- ☒ C: [issue] a call to wait() can return before cnt calls to done()
- D: [issue] simultaneous multiple calls to wait() and done() cause some wait()s wait forever
- ☒ E: [issue] multiple calls to wait() can wait forever if called before done()
- ☒ F: [fix] the mutex must be locked just before line 12 and unlocked just after it
- ☒ G: [fix] the mutex must be locked just before line 10 and unlocked after line 13
- ☒ H: [fix] line 17 must be moved just before line 16
- ☒ I: [fix] the notify_one() in line 12 must be replaced with notify_all()
- J: [fix] both F and I are needed
- ☒ K: [fix] both G and I are needed
- ☒ L: [fix] modifications G, H and I are needed together

3. (3p) Write a parallel program that computes the scalar product of two vectors of the same length. (The scalar product is $a_0 \cdot b_0 + a_1 \cdot b_1 + \dots + a_{n-1} \cdot b_{n-1}$.) The program shall be local (not distributed) and use a number of threads given as an argument.