Exam to Parallel and Distributed Programming
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.

```
void mergeSort(int* v, int dataSize, int myId, int nrProc) {
   if(nrProc == 1 || dataSize <= 1) {
      mergeSortLocal(v, dataSize);
   } else {
      int halfLen = dataSize / 2;
      int halfProc = nrProc / 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, dataSize-halfSize, myId, nrProc);
      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 implementing a future mechanism (the `set()` function is guaranteed to be called exactly once by the user code)

```
template<typename T>
class Future {
    T val;
    bool hasValue;
    mutex mtx;
    condition_variable cv;
public:
    Future() :hasValue(false) {}
    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()

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 prime numbers up to N. It is assumed to have the list of primes up to $\sqrt{N}$, and will check each of the other numbers if it is divisible with a number from the initial list.