1. (10) FBTC: The general recursive algorithmic approach which we noted several times lends itself to parallelization is called ________________.

2. (15) Consider the following pseudocode:

```plaintext
while true:
    request receive
do some computation
    check receive:
        if receive complete:
            process incoming data
            break
```

FBTC: This is an example of ________________ communication.

3. (15) Suppose we are doing a Fast Fourier Transform analysis of sound. We sample at a rate of 10,000 samples per second, for 10 seconds, with each sample recording one of 100 levels of sound loudness. Find the fundamental frequency.

4. (20) Here you will write MPI code to count the number of edges in a directed graph. (A link from i to j does not necessarily imply one from j to i.) In the context here, me is the node's rank; nv is the number of vertices; oh is the one-hop distance matrix; and mnodes is the number of MPI processes. At the beginning only the process of rank 0 has a copy of oh, but it sends that matrix out in chunks to the other nodes, each of which stores its chunk in an array ohchunk. Fill in the blanks:

```plaintext
MPI_Scatter(_______________________________________________
            ______________________________________________________);
mycount = 0;
for (i = 0; i < _____________________________________)
    if (____________________________) mycount++;
MPI_Reduce(_______________________________________________
            ______________________________________________________);
if (me == 0) printf("there are %d edges\n",numedge);
```

The call format of MPI_Scatter() is

```plaintext
MPI_Scatter(array at sender, number of items sent,
            MPI item type, array at receiver, number of
            items to receive, MPI item type, sender rank,
            communicator)
```

The reduction code for addition is MPI_SUM.

You are not allowed to add any code outside the blanks. Do not worry about declaring variables.

5. (20) Use Snow on a cluster of two machines to do a parallel sort using the following Quicksort-like scheme (fill in the blanks):

```plaintext
qs <- function(cls,x) {
    pivot <- x[1]
    chunks <- list()
    chunks[[1]] <- ________________
    chunks[[2]] <- ________________
    rcvd <- clusterApply(____________________)
    lx <- length(x)
    lc1 <- length(rcvd[[1]])
    lc2 <- length(rcvd[[2]])
    y <- vector(length=lx)
    if (lc1 > 0) ____________________ <- ________________
    if (lc2 > 0) ____________________ <- ________________
    return(y)
}
```

Note: Make use of R's built-in function sort(). You are not allowed to add any code outside the blanks.

6. (20) The following Snow code implements Shearsort on a cluster. Fill in the blanks.
is <- function(cls, dm) {
  n <- nrow(dm)
  numsteps <- ceiling(log2(n*n)) + 1
  for (step in 1:numsteps) {
    if (step %% 2 == 1) {
      augdm <- cbind(______________________, dm)
      dm <- parApply(______________________)  # parcel out to the cluster members for sorting
      dm <- t(dm)  # transpose the matrix
    } else dm <- parApply(______________________)
  }
  return(dm)
}

augsort <- function(augdmrow) {
  nelt <- length(augdmrow)
  if (______________________ % 2 == 0) {
    return(______________________)
  } else return(______________________)
}

Note that R's sort() function has a named argument decreasing, which is False for ascending sort and True for descending sort.

You are not allowed to add any material outside the blanks.

Solutions:
1. divide-and-conquer
2. nonblocking or asynchronous
3. There are 10000 × 10 = 100000 total sample points, i.e. the variable n in our PLN. So, f_0 = 10^{-5}.
4.
   MPI_Scatter(oh, nv*nv, MPI_INT, ohchunk, nv/nnodes, MPI_INT, 0, MPI_COMM_WORLD);
   mycount = 0;
   for (i = 0; i < nv*nv/nnodes)
       if (ohchunk[i] != 0) mycount++;
   MPI_Reduce(&mycount,&numedge,1,MPI_INT,MPI_SUM,0,MPI_COMM_WORLD);
   if (me == 0) printf("there are %d edges\n",numedge);

5.

   qs <- function(cls,x) {
     pvt <- x[1]
     chunks <- list()
     chunks[[1]] <- x[x <= pvt]
     chunks[[2]] <- x[x > pvt]
     rcvd <- clusterApply(cls,chunks,sort)
     lx <- length(x)
     lc1 <- length(rcvd[[1]])
     lc2 <- length(rcvd[[2]])
     y <- vector(length=lx)
     if (lc1 > 0) y[1:lc1] <- rcvd[[1]]
     if (lc2 > 0) y[(lc1+1):lx] <- rcvd[[2]]
     return(y)
   }

6.

is <- function(cls, dm) {
  n <- nrow(dm)
  numsteps <- ceiling(log2(n*n)) + 1
  for (step in 1:numsteps) {
    if (step %% 2 == 1) {
      # attach a row ID to each row
      augdm <- cbind(1:n, dm)
      # parcel out to the cluster members for sorting
      dm <- parApply(cls,augdm,1,augsort)
      dm <- t(dm)
    } else dm <- parApply(cls,dm,2,sort)
  }
  return(dm)
}

eaugsort <- function(augdmrow) {
  nelt <- length(augdmrow)
  if (augdmrow[1] % 2 == 0) {
    return(sort(augdmrow[2:nelt],decreasing=T))
  } else return(sort(augdmrow[2:nelt])))
}