Bulk Synchronous Message Passing: bspsort
(PSC2 §1.9)
Function `bspsort` is an implementation of Algorithm 1.4 for bulk synchronous parallel regular samplesort.
Bulk synchronous message passing (BSMP)

- A different way of communicating data.
- `bsp-send` primitive allows us to send data to a given processor without specifying the location where the data is to be stored.
- We view `bsp-send` as a `bsp-put` with a wildcard for the destination address.
- BSMP is one-sided communication, since it does not require any activity by the receiver in the same superstep.
- In the next superstep, the receiver must do something, at least if she wants to use the received data.
- BSPlib has 5 primitives for BSMP.
Motivation for BSMP

- Superstep (3) of samplesort uses `bsp_send` to send a local subblock $X_{st}$ of variable size from $P(s)$ to $P(t)$.
- The information on the size of the local subblock is only available at the sender.
- A sender does not know what others send to the same destination. Processors do not know what they will receive.
- If we were to use a `bsp_put`, we would have to specify a destination address.
- Reserving a maximum amount of space for each possible contributing subblock would require too much memory. First telling how much is going to be sent, reserving space, and asking the senders to put data there is clumsy, and requires 3 supersteps.
- `bsp_send` just sends the data to the right destination, without worrying about what happens afterwards.
Send operation from BSPlib

```c
bsp_send(pid, tag, source, nbytes);
```

- **bsp_send** copies `nbytes` of data from the local processor `bsp_pid` into a message, adds a tag, and sends the message to the destination processor `pid`.
- `source` points to the start of the data to be copied.
- In the next superstep, **bsp_move** writes at most `maxnbytes` into the memory pointed to by `dest`. 
Sending a subblock

```c
long i = 0;
for (long t=0; t<p ; t++){
    /* Send the values for P(t) */
    long i0= i;       // index of first value to be sent
    long count= 0;    // number of values to be sent
    while (i < nl &&
           (t==p-1 || (x[i] < Splitter[t+1].weight) ||
            (x[i] == Splitter[t+1].weight &&
             blocktotal_s+i < Splitter[t+1].index))){
        count++ ;  i++ ;
    }
    if (count > 0)
        bsp_send(t,&count,&x[i0],count*sizeof(double));
}
bsp_sync();
```

- The tag is the message size (not really needed).
Use it or lose it

bsp_move(dest, maxnbytes);

- The message sent using bsp_send is first stored by the system in a local send buffer, and then sent and stored in a buffer on the receiving processor.
- Some time after the message has been sent, it becomes available to the receiver. BSP philosophy: this happens at the end of the current superstep.
- In the next superstep, the messages can be read; reading messages means moving them from the receive buffer into the desired destination memory.
- At the end of the next superstep, all remaining unmoved messages will be lost, which saves buffer memory and forces the receiver into the right habit of cleaning his desk.
Concatenate the received parts

```c
bsp_qsize(&nparts_recvd,&nbytes_recvd);

start[0]= 0;
for (long j=0; j<nparts_recvd; j++){
    bsp_size_t payload_size; // payload size in bytes
    long count; // number of doubles in the message
    bsp_get_tag(&payload_size,&count);
    bsp_move(&x[start[j]],count*sizeof(double));
    start[j+1]= start[j] + count;
}
```

- `bsp_qsize` gives the number of messages received, i.e., the number of parts.
Set the tag size

```c
size_t tag_size = sizeof(long);
bsp_set_tagsize(&tag_size);
bsp_sync();
```

- When calling `bsp_set_tagsize`, `tag_size` represents the desired tag size.
- As a result, the system uses the desired tag size for all messages to be sent by `bsp_send`, starting from the next superstep.
- All processors must call the function with the same tag size.
- Side effect: `tag_size` is modified so that after the call it contains the previous tag size of the system. (This is a way of preserving the old system value.)
Summary

- We have encountered a new style of communication: **bulk synchronous message passing (BSMP)**, which uses the `bsp_send` primitive.
- In one superstep, an arbitrary number of communication operations can be performed, using either `bsp_put`, `bsp_get`, or `bsp_send`. These can be mixed freely.
- The BSP model and BSPlib do not favour any particular type of communication. It is up to the user to choose the most convenient primitive in a given situation. (But usually BSPlib is pretty paternalistic, forcing you to do the right thing.)
- **Irregular algorithms** benefit most from `bsp_send`.
- You now know the complete BSPlib, except for the dangerous high-performance primitives.