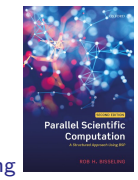
Bulk Synchronous Message Passing: `bspsort`

Section 1.9 of Parallel Scientific Computation, 2nd edition

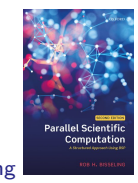
Rob H. Bisseling

Utrecht University



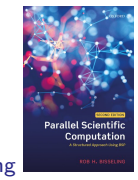
Parallel regular samplesort

- Function `bspsort` is an implementation of Algorithm 1.4 for bulk synchronous parallel regular samplesort.
- The communication pattern of its main communication superstep (3) is irregular, because varying amounts of data are being sent between the processors.
- For irregular communications, a different way of communicating data is convenient, called bulk synchronous message passing.



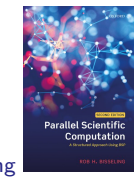
Bulk synchronous message passing (BSMP)

- The `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, which we will discuss, and 2 high-performance versions, which we will ignore.



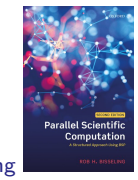
Motivation for BSMP

- Superstep (3) of samplesort uses `bsp_send` to send a local data set X_{st} of variable size from $P(s)$ to $P(t)$. The set size is only known to 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 sufficient space for each possible set size would require too much memory.
- First telling how much is going to be sent, then reserving the right amount of space, and finally asking the senders to put data there is clumsy, and requires 3 supersteps.

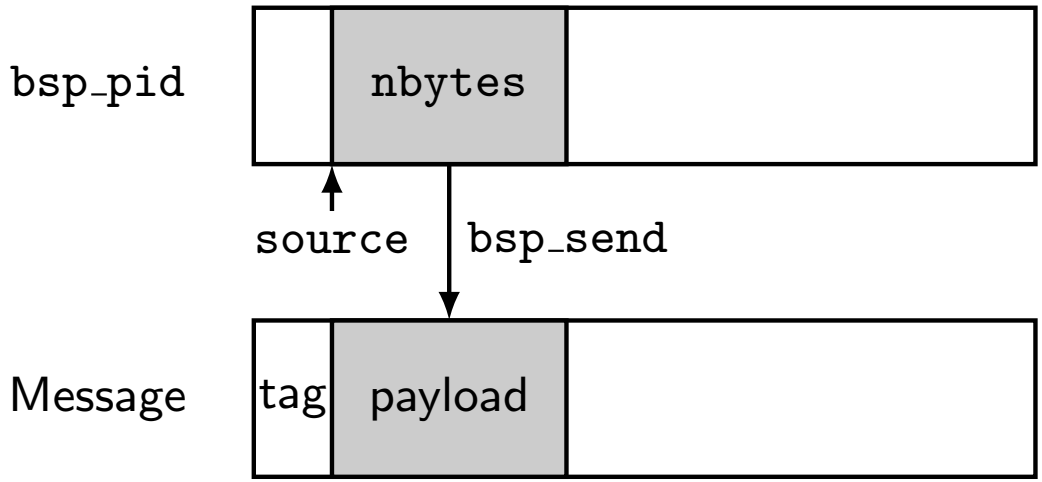


Send operation from BSPlib

- `bsp_send` just sends the data to the right destination processor, without worrying about what happens afterwards.
- In the next superstep, `bsp_move` moves all or part of the received data into the desired location.

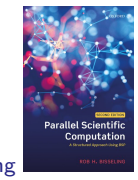


Primitive bsp_send




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

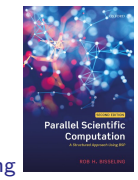
- bsp_send copies nbytes of data from the local processor bsp_pid into a message, adds a tag, and sends the message to the destination processor pid.
- source points to the start of the data to be copied.



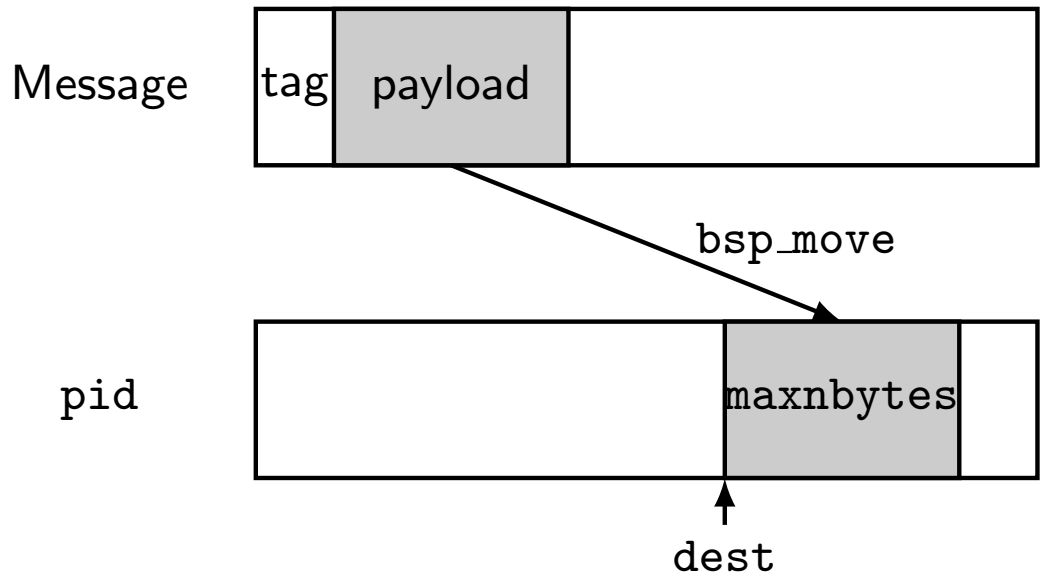
Sending a data set

```
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 < nloc(n,s,p) &&
           (t==p-1 ||
            (x[i] < Splitter[t+1].weight) ||
             x[i] ==Splitter[t+1].weight && ...)
           ){
        count++;
        i++;
    }
    if (count > 0)
        bsp_send(t,&count,&x[i0],count*sizeof(double));
}
```

- Here, the tag is the data count (for demonstration purposes).

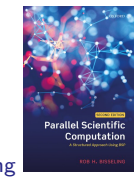


Primitive bsp_move



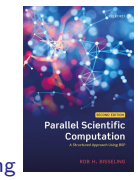

```
bsp_move(dest, maxnbytes);
```

- bsp_move writes at most maxnbytes into the memory pointed to by dest.



Use it or lose it

- 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 her desk.

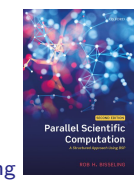


Getting information on the received data

```
bsp_nprocs_t nparts_recvd;
bsp_size_t nbytes_recvd;

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

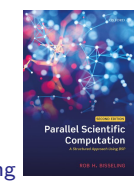
- `bsp_qsize` gives the number of messages (parts) in the receive buffer, which is a queue, and the total number of bytes.
- `bsp_nprocs_t` and `bsp_size_t` are integer types that can be used as wrappers to ensure compatibility with both the modernized version of BSPlib and the previous one.



Getting the tag of the first message in the queue

```
start[0]= 0;
for (long j=0; j<nparts_recvd; j++){
    bsp_size_t payload_size;
    long count;
    bsp_get_tag(&payload_size,&count);
    ...
}
```

- `bsp_get_tag` obtains the size in bytes and the tag of the first message in the queue.
- Here, we used the tag to give the data count of the message (which we also could derive from `payload_size`).

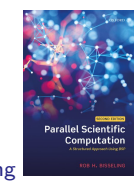


Concatenating the received parts

```
start[0]= 0;
for (long j=0; j<nparts_recvd; j++){
    bsp_size_t payload_size;
    long count;
    bsp_get_tag(&payload_size,&count);

    bsp_move(&x[start[j]],count*sizeof(double));
    start[j+1]= start[j] + count;
}
```

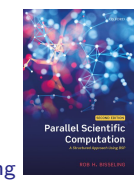
- The data words of the message are moved into locations `start[j]` to `start[j+1]-1` of the array `x`.



Setting the tag size

```
bsp_size_t tag_size= sizeof(long);
bsp_set_tagsize(&tag_size);
bsp_sync();
```

- When calling `bsp_set_tagsize`, the variable `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, thus 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. Apart from this, 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 advanced (dangerous!) high-performance primitives.