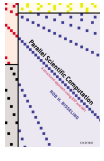
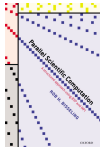
Bulk Synchronous Message Passing: `bspmv`
(PSC §4.9)



Parallel sparse matrix–vector multiplication

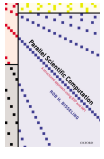
- Function `bspmv` is an implementation of Algorithm 4.5 for parallel sparse matrix–vector multiplication.
- It can handle every possible distribution of the matrix and vectors.



Data structure: indexing

```
void bspmv_init(int p, int s, int n,
     int nrows, int ncols, int nv, int nu,
     int *rowindex, int *colindex, ... ){
```

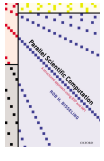
- Each processor first builds its own local data structure for representing the local part of the sparse matrix.
- Local nonempty rows are numbered $\texttt{i} = 0, \ldots, \texttt{nrows} - 1$, where $\texttt{nrows} = |I_s|$.
- Global index of the row with local index `i` is $i = \texttt{rowindex}[\texttt{i}]$.
- Global index of the column with local index `j` is $j = \texttt{colindex}[\texttt{j}]$.



Data structure: nonzeros

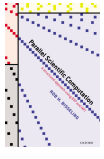
```
void bspmv(int p, int s, int n, int nz,
     int nrows, int ncols,
     double *a, int *inc, ... ){
```

- Nonzeros stored in order of increasing local row index `i`.
- Nonzeros of each local row stored consecutively in order of increasing local column index `j`, using the incremental compressed row storage (ICRS) data structure.
- The `k`th nonzero is stored as a pair $(\texttt{a}[\texttt{k}], \texttt{inc}[\texttt{k}])$, where $\texttt{a}[\texttt{k}]$ is the numerical value of the nonzero and $\texttt{inc}[\texttt{k}]$ the increment in the local column index.



Creating the matrix data structure

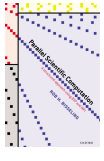
- Each triple (i, j, a_{ij}) is read from an input file and sent to the responsible processor, as determined by the matrix distribution.
- The local triples are then sorted by increasing global column index.
- This enables conversion to local column indices. During the conversion, the global indices are registered in `colindex`.
- The triples are sorted again, now by global row index. The original mutual precedences between triples from the same matrix row are maintained (i.e., the sort is stable).



Data structure: vector components

```
void bspmv_init(int p, int s, int n,
     int nrows, int ncols, int nv, int nu,
     int *rowindex, int *colindex,
     int *vindex, int *uindex, ... ){
```

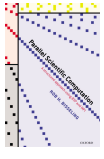
- Vector component v_j corresponds to a local component $\mathtt{v}[\mathtt{k}]$ in $P(\phi_{\mathbf{v}}(j))$, where $j = \mathtt{vindex}[\mathtt{k}]$. Here, $0 \leq \mathtt{k} < \mathtt{nv}$.
- All the needed vector components v_j, whether obtained from other processors or already present locally, are written into a local array `vloc`, which has the same local indices as the matrix columns.
- $\mathtt{vloc}[\mathtt{j}]$ stores a copy of v_j, where $j = \mathtt{colindex}[\mathtt{j}]$. Here, $0 \leq \mathtt{j} < \mathtt{ncols}$.



Where to get the input vector components

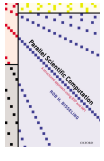
```
void bspmv(int p, int s, int n, int nz,
     int nrows, int ncols,
     double *a, int *inc,
     int *srcprocv, int *srcindv,  ... ){
```

- `bsp_get` is used to obtain v_j, because the receiver knows it needs v_j.
- The processor from which to get the value has processor number $\phi_{\mathbf{v}}(j) = \texttt{srcprocv[j]}$.
- The source processor needs to be determined only once. Its processor number can be used without additional cost in repeated application of the matrix–vector multiplication.
- We also store the location of v_j in the source processor as the local index `srcindv[j]`.



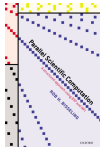
Bulk synchronous message passing (BSMP)

- 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 1-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 it wants to use the received data.
- BSPlib has 5 primitives for BSMP.

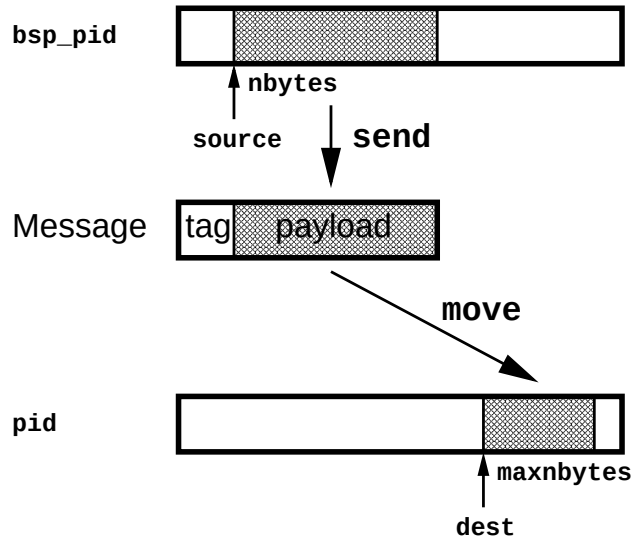


Motivation for BSMP

- Fanin uses `bsp_send` to send nonzero partial sum u_{it} to $P(\phi_{\mathbf{u}}(i))$.
- The information whether a nonzero partial sum for a certain row exists 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 space for each possible partial sum 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.

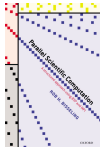


Send operation from BSPlib




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

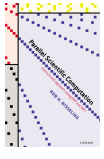
- `bsp_send` copies `nbytes` of data from the local processor `bsp_pid` into a message, adds a tag, and sends the message to the destination processor `pid`.
- `source` points to the start of the data to be copied.
- In the next superstep, `bsp_move` writes at most `maxnbytes` into the memory pointed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



Sending a partial sum

```
for(i=0; i<nrows; i++){
    *psum= 0.0;
    ... /* compute psum */
    bsp_send(destprocu[i],&destindu[i],
             psum,SZDBL);
}
```

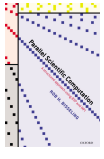
- The tag is an index $\texttt{destindu}[\texttt{i}]$ corresponding to i and the payload is the partial sum $\texttt{psum} = u_{it}$ consisting of 1 double.
- The destination processor, given by $\phi_{\mathbf{u}}(i) = \texttt{destprocu}[\texttt{i}]$, has been initialised beforehand by `bspmv_init`.
- The identity of the source processor is irrelevant and is not sent along with the data.



Use it or lose it

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

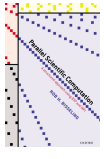
- The message sent using `bsp_send` is first stored by the system in a local send buffer.
- The message is then sent and stored in a buffer on the receiving processor.
- Some time after the message has been sent, it becomes available to the user. BSP philosophy: this happens at the end of the current 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 user into the right habit of cleaning his desk.



Summation of received partial sums

```
bsp_qsize(&nsums,&nbytes);
bsp_get_tag(&status,&i);
for(k=0; k<nsums; k++){
    bsp_move(&sum,SZDBL);
    u[i] += sum;
    bsp_get_tag(&status,&i);
}
```

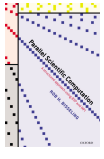
- `bsp_qsize` gives the number of messages received, i.e., the number `nsums` of partial sums.
- Each message is processed immediately. Instead, we could also have allocated `nbytes` storage space and process all the messages together.
- The index `i` is obtained from the tag; the sum from the payload.



Set the tag size

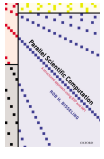
```
int tagsz= SZINT;
bsp_set_tagsize(&tagsz);
bsp_sync();
```

- When calling `bsp_set_tagsize`, `tagsz` 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: `tagsz` 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.)



Pointer magic for ICRS

```
psum= &sum;
pa= a; pinc= inc;
pvloc= vloc; pvloc_end= pvloc + ncols;
pvloc += *pinc;
for(i=0; i<nrows; i++){
    *psum= 0.0;
    while (pvloc<pvloc_end){
        /* sum += a[k] * vloc[j] */
        *psum += (*pa) * (*pvloc);
        pa++;
        pinc++;
        pvloc += *pinc;
    }
    bsp_send(destprocu[i],&destindu[i],
             psum,SZDBL);
    pvloc -= ncols;
}
```



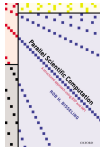
Initialisation function `bspmv_init`

This is what I have. Write owner of every local component v_j cyclically into a temporary array.

```
for(j=0; j<nv; j++){
    jglob= vindex[j];
    bsp_put(jglob%p,&s,tmpprocv,
            (jglob/p)*SZINT,SZINT);
}
bsp_sync();
```

Where can I find what I need?

```
for(j=0; j<ncols; j++){
    jglob= colindex[j];
    bsp_get(jglob%p,tmpprocv,
            (jglob/p)*SZINT,&srcprocv[j],SZINT);
}
```



I can get v_j from $P($`srcprocv[j]`$)$ at location `srcindv[j]`

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`.