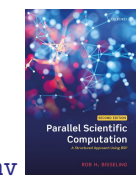
Program `bspmv`

Section 4.11 of Parallel Scientific Computation, 2nd edition

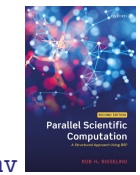
Rob H. Bisseling

Utrecht University



Parallel sparse matrix–vector multiplication

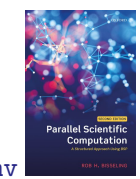
- The function `bspmv` is an implementation of Algorithm 4.5 for parallel sparse matrix–vector multiplication.
- It can handle every possible distribution of the matrix and the vectors.



Data structure: indexing

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

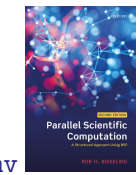
- Each processor first builds its own local data structure for representing the local part of the sparse matrix.
- Local nonempty rows are numbered $\mathtt{i} = 0, \ldots, \mathtt{nrows} - 1$, where $\mathtt{nrows} = |I_s|$.
- The global index of the row with local index `i` is $i = \mathtt{rowindex[i]}$.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



Data structure: nonzeros

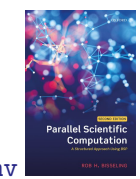
```
void bspmv(long n, long nz, long nrows, long ncols,
           double *a, long *inc, ...){
```

- Nonzeros are stored in order of increasing local row index `i`.
- The nonzeros of each local row are 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 (`a[k]`, `inc[k]`), where `a[k]` is the numerical value of the nonzero and `inc[k]` the increment in the local column index.



Creating the matrix data structure

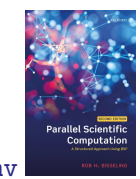
- Each triple (i, j, a_{ij}) is read from an input file and sent to the responsible processor, as determined by the matrix distribution.
- This is done in batches of size at most `MAXSEND` to save buffer space, at the expense of an increase in the number of supersteps.
- The local triples are then sorted 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(long n, long nrows, long ncols,
                long nv, long nu,
                long *rowindex, long *colindex,
                long *vindex, long *uindex, ...){
```

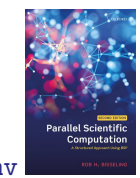
- Vector component v_j corresponds to a local component `v[k]` in $P(\phi_{\mathbf{v}}(j))$, where $j =$ `vindex[k]`. Here, $0 \leq$ `k` $<$ `nv`.
- All the needed vector components v_j, whether obtained from other processors or already present locally, are written into a local array `vloc`, which has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



Where to get the input vector components

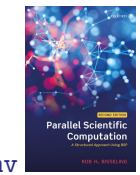
```
void bspmv(long n, long nz, long nrows, long ncols,
           double *a, long *inc,
           long *srcprocv, long *srcindv, ...){
  ...
  bsp_get(srcprocv[j],v,srcindv[j]*sizeof(double),
          &vloc[j],sizeof(double));
  ...
```

- `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 $\phi_{\mathbf{v}}(j) =$ `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]`.



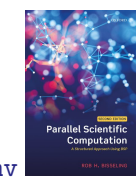
Possible optimizations

- We use `bsp_get` to obtain v_j, but we still need preprocessing to determine `srcprocv[j]` and `srcindv[j]`.
- With some extra preprocessing we could have used `bsp_put` instead.
- With even more preprocessing we could have put all the data for the same destination together, as one packet. This would attain optimistic g-values.
- Principle: more preprocessing gives less work in repeated multiplications.
- Optimization makes a program faster but sometimes it also creates a mess. Therefore, we did not implement the above optimizations in BSPedupack.
- A straightforward optimization is the direct assignment in the fanout of the value v_j to `vloc[j]` in case v_j is local.
- We implemented this to avoid the substantial overhead of a call to `bsp_get` in the local case.



Motivation for using Bulk Synchronous Message Passing

- The 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.
- `bsp_send` is convenient here: it just sends the data to the right destination, without worrying about what happens afterwards.

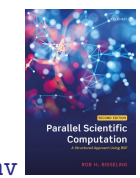


Sending a partial sum

```
for (long i=0; i<nrows; i++){
    double sum= 0.0;
    ... /* compute sum */

    if (destprocu[i] == s)
        u[destindu[i]]= sum;
    else
        bsp_send(destprocu[i],&destindu[i],
                 &sum, sizeof(double));
```

- The tag is an index `destindu[i]` corresponding to i and the payload is `sum` $= u_{it}$ consisting of 1 double.
- The tag should be chosen such that it enables the receiver to handle the payload easily.
- The destination processor, given by $\phi_{\mathbf{u}}(i) =$ `destprocu[i]`, has been initialized beforehand by `bspmv_init`.
- The identity of the source processor is irrelevant and is not sent along with the data.

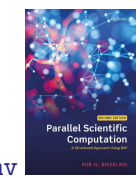


Summation of received partial sums

```
bsp_qsize(&nsums,&nbytes);
bsp_get_tag(&status,&i);

for (long k=0; k<nsums; k++){
    /* status != -1, but its value is not used */
    double sum;
    bsp_move(&sum, sizeof(double));
    u[i] += sum;
    bsp_get_tag(&status,&i);
}
```

- `bsp_qsize` gives the number of messages received, i.e., the number `nsums` of partial sums.
- The index `i` of a message is obtained from its tag and the sum from its payload. The index `i` is the local index at the receiver.

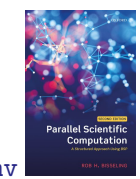


Pointer magic for ICRS in local SpMV

```
double *pa= a; // pointer to a
long *pinc= inc;
double *pvloc= vloc;
double *pvloc_end= pvloc + ncols;

for (long i=0; i<nu; i++)
    u[i]= 0.0;

pvloc += *pinc;
for (long i=0; i<nrows; i++){
    double sum= 0.0;
    while (pvloc<pvloc_end){
        sum += (*pa) * (*pvloc); // = a[k]*vloc[j]
        pa++;
        pinc++;
        pvloc += *pinc;
    }
    ... // send sum
    pvloc -= ncols;
}
```



Initialization function bspmv_init

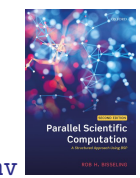
- This is what I have. Write the owner of every local component v_j cyclically into a temporary array.

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

- Where can I find what I need? In processor $P(j \bmod p)$ at location $P(j \operatorname{div} p)$.

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

- I can get v_j from P(srcprocv[j]). Similar for its location srcindv[j].



Summary

- The input of a sparse matrix requires a lot of preprocessing:
  - we send the matrix nonzeros as triples (i, j, a_{ij}) to the processors that own them according to the matrix distribution;
  - we sort the local nonzeros twice, in a stable way, first by global column index and then by global row index;
  - we create the Incremental Compressed Row Storage (ICRS) data structure.
- Furthermore, we announce the owner and location of vector components to the processors that need this information.
- It is often worthwhile to remove the overhead of a function call to `bsp_put`, `bsp_get`, or `bsp_send` in case communication stays within a processor.
- Bulk Synchronous Message Passing (BSMP) is convenient for irregular computations such as sparse matrix–vector multiplication.