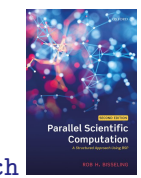
Program `bspmatch`

Section 5.9 of Parallel Scientific Computation, 2nd edition

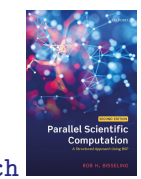
Rob H. Bisseling

Utrecht University



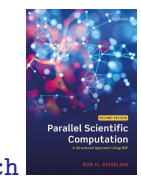
Parallel local domination algorithm

- The function `bspmatch` is an implementation of Algorithm 5.7 for parallel weighted graph matching.
- It can handle every possible vertex distribution ϕ, where the edges connected to a vertex are stored together with that vertex in an adjacency list.
- The distribution of the edges thus corresponds to a 1D row distribution of the adjacency matrix A of the graph.



Driver program `bspmatch_test`

- The driver program `bspmatch_test` from BSPedupack reads a sparse matrix in Matrix Market format, distributed in a 1D row distribution, without diagonal entries and with the symmetric partner a_{ji} included for every nonzero a_{ij}.
- The distribution is represented by a list of nonzeros assigned to $P(0)$, followed by those assigned to $P(1)$, and so on.
- The nonzeros are sent to the processors as prescribed by ϕ, where they become the edges in the adjacency lists of the vertices.
- The input procedure of `bspmatch` has much in common with that of the sparse matrix–vector multiplication function `bspmv`; the common functions are included in the file `bspsparse_input.c`.



Data structure: edge numbers and weights

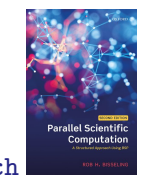
- `bspmatch_test` builds the data structures needed by `bspmatch`, which are a list of
  - internal edges, numbered $\mathtt{e} = 0, \ldots, \mathtt{nedges} - 1$;
  - external edges, numbered $\mathtt{e} = \mathtt{nedges}, \ldots, \mathtt{nedges} + \mathtt{nhalo} - 1$.
- For each edge $\mathtt{e} = (u, v)$, we can look up its weight

$$\mathtt{weight[e]} = \omega(e),$$

  which is a double, and its secondary weight

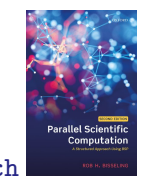
$$\mathtt{weight1[e]} = \omega_1(e) \cdot 2n + \omega_2(e).$$

- Here, $\omega_1(e) = 1$ if e is internal and $\omega_1(e) = 0$ if e is external, and $\omega_2(e) = u + v$.
- Since $\omega_2(e) < 2n$, this breaks ties by first giving preference to internal edges.



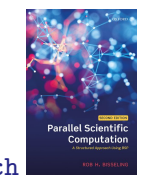
Comparing edges

```
bool heavier(long e0, long e1,
             double *weight, long *weight1){

    /* This function checks whether edge e0 is heavier
       than edge e1 */

    if (e0 == DUMMY)
        return false;

    if (e1 == DUMMY || weight[e0] > weight[e1] ||
       (weight[e0] == weight[e1] &&
        weight1[e0] > weight1[e1]))
        return true;

    return false;
}
```



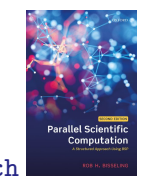
Data structure: vertex numbers

- We number the vertices locally, $\mathtt{v} = 0, \ldots, \mathtt{nvertices} - 1$, where we include only those with a nonempty adjacency list.
- If an edge e is internal, we store the local vertex numbers $\mathtt{v0}[\mathtt{e}]$ and $\mathtt{v1}[\mathtt{e}]$ of its two endpoints, where $\mathtt{v0}[\mathtt{e}] < \mathtt{v1}[\mathtt{e}]$.
- If e is external, we store its local vertex number $\mathtt{v0}[\mathtt{e}]$, but $\mathtt{v1}[\mathtt{e}]$ then stores the corresponding local edge number $\mathtt{e}'$ on the remote processor $P(t)$ that shares the edge with $P(s)$.
- The numbering with internal edges first makes it easy to see whether an edge is internal or external, which can be done simply by checking whether $\mathtt{e} < \mathtt{nedges}$.



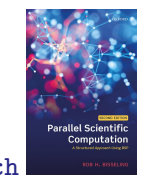
Communicating along an external edge

- If we send a message to a remote processor along an external edge e, we communicate in the language of the receiver, who can look up all information about the corresponding edge e', such as its weight and its local vertex.
- For each external edge e, we store the remote owner $P(t)$ as $\texttt{destproc}[\texttt{e} - \texttt{nedges}] = t$.



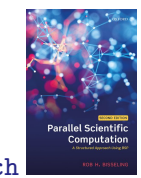
Rejecting a suitor

```
#define REJECT 2

void reject_suitor(long v, long e, long q_lo, long *nq,
                   long nvertices, long nedges,
                   long *v0, long *v1, long *destproc,
                   bool *Alive, long *Pref){

    /* This function rejects suitor e of vertex v */

    if (e < nedges){
        /* Determine other end point of edge e */
        long x = (v0[e]==v ? v1[e] : v0[e]);
        push(x, nvertices, q_lo, nq, Q, Pref);
    } else {
        long tag= REJECT;
        bsp_send(destproc[e-nedges], &tag,
                 &(v1[e]), sizeof(long));
    }
    Alive[e]= false;
}
```



Finding the remote owners

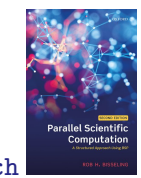
- In the program `bspmatch_test`, each processor $P(s)$ first writes its processor number s into a cyclically distributed temporary array, at all global indices corresponding to a local vertex.
- This announces that $P(s)$ is the owner of all its local vertices.
- Then, $P(s)$ reads the owners of its halo vertices from the temporary array.
- This procedure is similar to the notice board procedure used for parallel sparse matrix–vector multiplication in `bspmv_init`.



Establishing the correspondence between e and e′

- To talk in the language of the receiver, we need to establish the correspondence between e and e′.
- To achieve this, the local edge numbers e of the halo edges are first sent to the remote processors, together with their global indices (i, j), corresponding to a_{ij} in the adjacency matrix.
- The triples (e, i, j) are then sorted lexicographically by the receiver, with primary key j and secondary key i.
- The local triples (e', i, j) on the remote processor were already sorted lexicographically with primary key i and secondary key j when creating the CRS data structure.
- The coupled nonzeros a_{ji} and a_{ij} are now stored in the same order, so that their corresponding edge numbers e and e′ can be coupled on the remote processor by setting

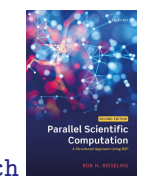
$$\mathtt{v1}[\mathrm{e}'] = \mathrm{e}.$$



Declaring edges dead

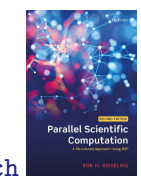
An edge (u, v) in the adjacency list of v can be declared dead:

- if u has a better suitor;
- if the degree $d_u = 0$, meaning that u has been matched or its adjacency list has been depleted; the exception is if u is the suitor of v, which may have caused d_u to become 0.



Finding the highest living edge

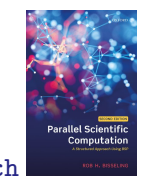
```
void find_alive (long v, long *Adj, long nedges,
     double *weight, long *weight1, long *v0, long *v1,
     bool *Alive, long *Suitor, long lo, long *d){

/* Finds highest living edge in Adj[lo,lo+d[v]-1] */

    for (long i= lo+d[v]-1; i>=lo; i--){
        long e = Adj[i];
        if (e < nedges){  // e=(u,v) is internal
            long u = (v0[e]==v ? v1[e] : v0[e]);
            if ((d[u]==0 && Suitor[v]!=e) ||
                heavier(Suitor[u], e, weight, weight1))
                Alive[e] = false;
        }
        if (Alive[e])
            return;
        else
            d[v]--;
    }
}
```



Pushing a vertex onto the queue

```
void push(long v, long nvertices, long q_lo,
          long *nq, long *Q, long *Pref){

    /* This function pushes vertex v onto the queue */

    long q_hi= q_lo + (*nq); // first free position
    if (q_hi >= nvertices)
        q_hi -= nvertices;
    Q[q_hi]= v;
    (*nq)++; // nq = number of vertices in the queue
    Pref[v]= DUMMY;
}
```

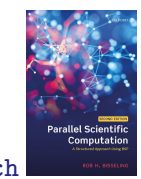
- The queue is stored as a circular list in positions `q_lo` to `q_lo`$+nq-1$ of array Q, wrapping around at `nvertices` (the number of local vertices).
- A new vertex v is pushed onto the queue at the tail.



Popping a vertex from the queue

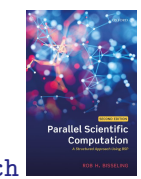
```
long pop(long nvertices, long *q_lo,
         long *nq, long *Q){

    /* This function pops a vertex v from the queue */

    long i= *q_lo;
    (*q_lo)++;
    if (*q_lo >= nvertices)
        *q_lo -= nvertices;
    (*nq)--;

    return Q[i];
}
```

- A vertex v is popped at the head of the queue, `q_lo` is incremented, again wrapping around at `nvertices`, and nq is decremented.



bspmatch: detecting termination

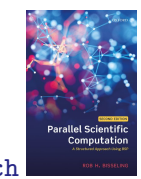
```
bool alldone= false;
while (!alldone){
    for (long t=0; t<p; t++)
        Done[t]= false;

    bsp_qsize(&nmessages,&nbytes);
    if (nmessages==0 && nq==0){
        long done= true; // P[s] is done
        for (long t=0; t<p; t++)
            bsp_put(t,&done,Done,s*sizeof(long),
                    sizeof(long));
    }...
    bsp_sync();
    alldone= true;
    for (long t=0; t<p; t++)
        if(Done[t] == false){
            alldone= false;
            break;
        }
}
```



bspmatch: counting operations

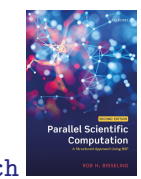
```
while (nq > 0 && nops_step < maxops){
    long v= pop(nvertices, &q_lo, &nq, Q);

    /* Find highest living edge */
    if (degree[v] > 0){
        long degree_old= degree[v];
        find_alive (v,Adj,nedges,weight,weight1,v0,v1,
                    Alive,Suitor,Start[v],degree);
        nops_step += degree_old-degree[v]+1;
    }
    ...
}
```

- `find_alive` contains a loop which runs downwards from the old degree to the new (possibly smaller) degree, taking `degree_old-degree[v]+1` operations.
- The operation count is added to the number of operations `nops_step` of the current superstep.



bspmatch: registering a match or proposing

```
#define ACCEPT 1
#define REJECT 2
Pref[v]=e;
if (e==Suitor[v]){
    match[*nmatch]= e;
    (*nmatch)++; // number of matches registered so far
    if (e < nedges){ // internal edge
        degree[v0[e]]= 0;
        degree[v1[e]]= 0;
    } else {
        degree[v]= 0;
        long tag= ACCEPT;
        bsp_send(destproc[e-nedges], &tag,
                 &(v1[e]), sizeof(long));
    }
} else if (e >= nedges){
    long tag= PROPOSE;
    bsp_send(destproc[e-nedges], &tag,
             &(v1[e]), sizeof(long));
}
```



Summary

- We have presented the function `bspmatch` from BSPedupack, which is an implementation of the parallel local domination algorithm for weighted graph matching.
- If we send a message to a remote processor along an external edge e of our graph, we prefer to communicate in the language of the receiver.
- For this purpose, we established the correspondence between a locally stored edge e and its remote partner e' by sending a triple (e, i, j) to the owner of e'.
- We use `bsp_send` to send proposals, accepts and rejects, because this BSP primitive is most convenient for an irregular algorithm such as graph matching.