Heartbeat algorithms

This class of algorithms is used to solve problems that deal with iterative data parallelism. For example, they can be used when data needs to be split between processes/threads; each thread is responsible for updating a part of the problem data and, in addition, the updated values of the data depend on values owned by other threads (particularly direct neighbors).

Heartbeat paradigm expresses an interaction model between threads. The name comes from the threads working model, similar to an actual heart beat: each thread sends data, receives data, processes data and repeats. General algorithm (pseudocode):

```
process Worker[i = 1 to NW] {
    declare local variables;
    initialize local variables;
    while ( not done ) {
        send values to neighbors;
        receive values from neighbors;
        update local values;
    }
}
```

Practical example: Region labelling

Let's assume we have an image as a matrix mxn of pixels. All pixels in the image can only have two values: 1 if live pixel, or 0 if dead. The task is to find regions of pixels of the same type. Two pixels belong to the same region of they are neighbors in the matrix. Neighbor in this case strictly means directly above, below, on the left or on the right (not on the diagonal).

For solving this problem, we employ a second matrix in which we keep the current label of each pixel.

First way to solve this problem is in an iterative manner. Basically, each iteration we examine each pixel and its neighbors. If both a pixel and its neighbor are of the same type (same value), then we set the label for these pixels to the maximum between the values of their current labels.

We can parallelize this solution by having one thread take care of one pixel. The algorithm is finished when, after an entire iteration, no label is modified anymore. Pseudocode:

```
for (int i = 0; i < m; i++) {
    for (int j = 0; j < n; j++) {
        if (image[i,j] == 1)
        label[i,j] = i*n+j; // unique label for each pixel
    }
}
change = true;
while (change) {
change = false;
for (int i = 0; i < m; i++) {
    for (int j = 0; j < n; j++) {
          int oldlabel = label[i,j];
          if (image[i,j] == 1) {
               label[i,j] = max(label for neighbors);
          }
          if (label[i,j] != oldlabel)
               change = true;
     }
}
```

Another solution for this problem is to use the heartbeat paradigm. We have P+1 threads. We split the image in P strips and assign each worker thread to one strip. It's recommended to use strips of pixels rather than blocks of pixels for two reasons: it's easier to compute indexes on the initial image when splitting it; and there is less communication that needs to be handled between workers, since each worker thread will have to communicate with two neighbors instead of four. Pseudocode:

```
process Worker(w = 1 to P) {
     int stripSize = m / W; // local values plus edges
     int image[stripSize+2, n]; // from neighbors
     int label[stripSize+2, n];
     int change = true;
     initialize image[1:stripSize, *] and label[1:stripSize, *];

     //exchange edges and image with neighbors
     if (w != 1)
            send first[w-1](image[1,*]); // to worker above
     if (w != P)
            send second[w+1](image[stripSize,*]); // to worker below
     if (w != P)
            receive first[w](image[stripSize+1,*]); // from worker below
     if (w != 1)
            receive second[w](image[0,*]); // from worker above
     while (change) {
          exchange edges of label with neighbors, as above;
          update label[1:stripSize, *] and set change to true if the value of the
          label changes;
          send result(change); // tell coordinator
          receive answer[w](change); // and get back the answer
     }
}

process coordinator {
     bool chg, change = true;
     while (change) {
          change = false;
          //see if there has been a change in any strip
          for [i = 1 to P] {
               receive result(chg);
               change = change or chg;
            }
            // broadcast answer to every worker
            for (i = 1 to P) {
                  send answer[i](change);
            }
      }
}
```