# Implementation of Boruvka Algorithm for Minimum Spanning Tree Search on Charm++

Alexander Frolov
Science and Research
Centre of Computer Technology
NICEVT
Varshavskoe sh. 125, Moscow, Russia
Email: frolov@nicevt.ru

Alexander Semenov
Science and Research
Centre of Computer Technology
NICEVT
Varshavskoe sh. 125, Moscow, Russia
Email: semenov@nicevt.ru

***Abstract*—In this paper a parallel implementation of the Boruvka algorithm for minimum spanning tree search on the message-driven parallel programming language – Charm++ is presented.**

## I. Introduction

## II. Related work

## III. Charm++

## IV. Boruvka algorithm

### A. Canonical Boruvka's algorithm

### B. Boruvka's algorithm in Charm++ paradigm

We used straight forward vertex-centric approach for implementing Boruvka's algorithm on Charm++. Each vertex of the graph is represented by a separate chare, while a graph is a single dimension chare array. Each chare stores information about adjacent edges as well as supplementary data for the algorithm.

The main chare controls the overall execution and works as a global master, synchronizing execution of the vertex chares. The main loop is represented in the algorithm 1, at it can be seen the loop body consists of the tree stages with barrier synchronization after each of stage. As each stage is asynchronous and its execution depends fully on the graph structure and topology and its completion can not be predicted by any simple way it was decided to use quiescence detection mechanism built into Charm++ runtime system for detecting completion of the stages.

For making code simple we used in *threaded* entry method *Main::Boruvka* and obtained callback through *CkCallbackResumeThread* call and provided it to *CkStartQD*. That will suspend execution of the *Main::Boruvka* method until all other threads (instantiated by calling entry methods in each stage) will be completed.

The for loop in the algorithm 1 (lines 3–11) is being executed until *Stop* is true, that is no supervetices have external edge, that is all connected components of the graph are found. The value of the *Stop* can be changed by entry method *Main::SetStop* and is set to false each time external edge is found in any supervertex.

At the first stage, a search of the edge with minimum weight connecting one of the vertices of the supervertex with the vertex from another supervertex is performed for each supervertex. This is done by calling *SearchMinEdge* entry method of all vertices of the graph.

Each vertex has its adjacent edge list (*edges*). Its element of *edges* has the following properties: *dst* – neighbour vertex identifier, *weight* – weight of the edge, *local* – boolean flag showing if set to *true* then the neighbour vertex belongs to the same supervertex, and *mst* is boolean flag set to *true* if the edge belongs to minimum spanning tree. Also vertex has *SupervertexId* which specifies the supervertex identifier which the vertex belongs to. The *SupervertexId* variable initially set to *thisIndex*, that is chare identifier.

**Algorithm 1** Message-driven Boruvka's algorithm: main loop

```
1:  procedure THREADED MAIN::BORUVKA
2:      Stop ← false
3:      while Stop = false do
4:          Stop ← true
5:          Chares.SearchMinEdge()
6:          Main.WaitQD(CkCallbackResumeThread())
7:          Chares.MergeSubgraphs()
8:          Main.WaitQD(CkCallbackResumeThread())
9:          Chares.UpdateLocalState()
10:         Main.WaitQD(CkCallbackResumeThread())
11:     end while
12: end procedure
```

The *SearchMinEdge* method is shown in the algorithm 2. At first, a local search is performed to find an edge with minimum weight connecting this vertex with another one from another supervertex (lines 2 – 6). It is presumed that *edges* is sorted by *weight*. The *e* is an index of the external edge with minimum weight and it is set to zero at the initialization. After that, if the edge has been found, two variables *MinChareId* and *MinWeight* are set with the current vertex identifier (line 12) and the weight of the found edge (line 11). Otherwise, if there is no such edge (all adjacent edges are internal) then *MinChareId* and *MinWeight* are set to -1 and maximum possible value, correspondingly.

Then all vertices (chares) within a supervertex (subgraph) are asynchronously exchanging their values of *Min-*

**Algorithm 2** Message-driven Boruvka's algorithm: search of a minimal external edge in supervertices

```
 1: procedure CHARE::SEARCHMINEDGE
 2:     for e : e → (edges.size() − 1) do
 3:         if edges[e].local = false then
 4:             break
 5:         end if
 6:     end for
 7:     if e = (edges.size() − 1) then
 8:         MinWeight ← max(double)
 9:         MinChareId ← −1
10:     else
11:         MinWeight ← edges[e].weigth
12:         MinChareId ← thisIndex
13:         for i = 0 → (edges.size() − 1) do
14:             if edges[i].local = true then
15:                 Chares[i].PostMinEdge(MinChareId,
16:                     MinWeight)
17:             end if
18:         end for
19:     end if
20: end procedure
21: procedure CHARE::POSTMINEDGE(ID, W)
22:     if w < MinWeight then
23:         MinWeight ← w
24:         MinChareId ← id
25:         for i = 0 → (edges.size() − 1) do
26:             if edges[i].local = true then
27:                 Chares[i].PostMinEdge(MinChareId,
28:                     MinWeight)
29:             end if
30:         end for
31:     end if
32: end procedure
```

**Algorithm 3** Message-driven Boruvka's algorithm: merging supervertices

```
 1: procedure CHARE::MERGESUBGRAPHS
 2:     if MinChareId ≠ thisIndex then
 3:         return
 4:     end if
 5:     Main.setStop(false)
 6:     for i = 0 → (edges.size() − 1) do
 7:         if edges[i].dst = edges[e].dst then
 8:             edges[i].local ← true
 9:         end if
10:     end for
11:     edges[e].mst ← true
12:     Chares[edges[e].dst].PropagateSupervertexId(thisIndex,
13:                 svid,false)
14:     Chares[edges[e].dst].AddEdgeToMST(thisIndex,
15:                 edges[e].weight)
16: end procedure
17: procedure CHARE::PROPAGATESUPERVERTEXID(SRC, SVID,
    CONNECTED)
18:     if svid < SupervertexId then
19:         SupervertexId ← svid
20:         Updated ← true
21:         for i = 0 → (edges.size() − 1) do
22:             if edges[i].local = true then
23:                 Chares[i].PropagateSupervertexId(thisIndex,
24:                     SupervertexId,true)
25:             end if
26:         end for
27:     end if
28:     if connected = false then
29:         for i = 0 → (edges.size() − 1) do
30:             if edges[i].dst = src then
31:                 edges[i].local ← true
32:             end if
33:         end for
34:     end if
35: end procedure
36: procedure CHARE::ADDEDGETOMST(SRC, W)
37:     for i = 0 → (edges.size() − 1) do
38:         if (edges[i].dst = src) and
39:             (edges[i].weight = w)  then
40:             edges[i].mst ← true
41:         end if
42:     end for
43: end procedure
```

*Weight* and *MinChareId* (lines 13 – 17, and procedure *Chare::PostMinEdge*) until the all vertices of the subgraph have the same *MinWeight* and *MinChareId*. When all supervertices complete this process the CkStartQD returns and the next stage starts.

The next stage is initiated by calling the *MergeSubgraphs* method for all vertices (algorithm 1, line 7). The *MergeSubgraphs* method and supplementary methods *PropagateSupervertexId* and *AddEdgeToMST* are shown in algorithm 3.

In the *MergeSubgraphs* method a check is performed to find vertices which own minimum external edges found on the previous stage (lines 2 – 4). If such vertices exist then the Boruvka's algorithm stop condition is not met and each one calls the *setStop* method in the *main* chare to set *Stop* value to **false**. After that all the edges connecting the current vertex to the vertex adjacent via minimum external edge are marked as local (lines 6 – 9), and the external edge is marked as belonging to the minimum spanning tree edges (line 11). Then propagation of the supervertex identifier is started by calling the *PropagateSupervertexId* method, and the reflecting edge in the neighbore vertex is marked as MST edge as well by calling the *AddEdgeToMST* method.

The propagation of supervertex identifier is performed asynchronously. If propagated value is smaller than the current *svid* then it will be updated and broadcasted to all local neighbours. After the process is completed the minimal supervertex identifier of identifiers of merging supervertices will be set in *sid* variable of all chares in the resulting supervertex (see algorithm 3, lines 18 – 27).

For the vertices connected by the mst edge the *local* variable of the edge list is set to **true** (lines 29 – 33).

When all supervertices which are to be merged are merged the CkStartQD (algorithm 1, line 8) is returned and the final stage of the loop is started by calling *UpdateLocalState* in each vertex. The goal of this step is to update *local* values of edges in the edge lists of the vertices which have been updated, that is their *svid* changed.

The *UpdateState* method and supplementary *UpdateStateReq* and *UpdateStateResp* are shown in the algorithm 4. Each updated vertex scans its edge list and for edges

**Algorithm 4** Message-driven Boruvka's algorithm: updating edge lists

```
1: procedure CHARE::UPDATESTATE
2:     if Updated =true then
3:         for i = 0 → (edges.size() − 1) do
4:             if edges[i].local =false then
5:                 Chares[edges[i].dst].UpdateStateReq(
6:                     thisIndex, SupervertexId)
7:             end if
8:         end for
9:         Updated ← false
10:    end if
11: end procedure
12: procedure CHARE::UPDATESTATEREQ(SRC, SVID)
13:    if SupervertexId = svid then
14:        for i = 0 → (edges.size() − 1) do
15:            if (edges[i].dst = src) then
16:                edges[i].local ← true
17:            end if
18:        end for
19:        Chares[edges[i].dst].UpdateStateResp(thisIndex)
20:    end if
21: end procedure
22: procedure CHARE::UPDATESTATERESP(SRC)
23:    for i = 0 → (edges.size() − 1) do
24:        if (edges[i].dst = src) then
25:            edges[i].local ← true
26:        end if
27:    end for
28: end procedure
```

which connect it to vertices from other supervertices, i.e. its *local* is **false**, a request is send by calling *UpdateStateReq* at neighbours connected by such edges (lines 3 – 8). If such neighbours have the same *SupervertexId* then *local* is set to **true** for both vertices (lines 14 – 18 and lines 23 – 27).

When execution of the Boruvka's algorithm is completed all vertices will have *SupervertexId* variable set to vertex identifier representing a root of the minimum spanning tree. If there are more than one tree in the graph that is the graph has several disconnected components then each component will have its own root and *SupervertexId* of the vertices from different components will be different.

By marking edge belonging to MST on both ends it is possible to traverse trees in upward and downward directions.

## V. Evaluation results

### A. Platforms

### B. Results

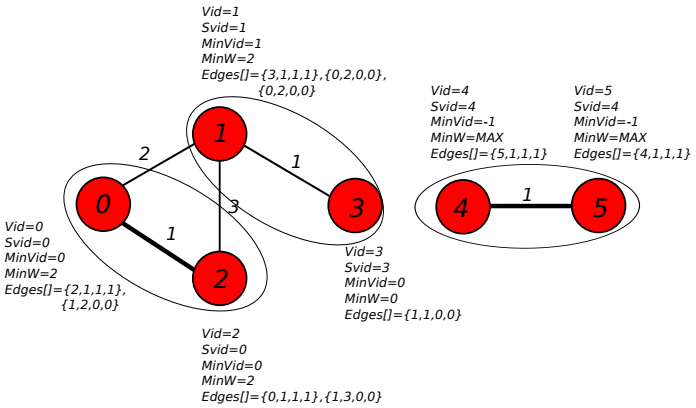


Fig. 1. Illustration of the Boruvka's algorithm: second iteration of the main loop, the minimal external edge search is completed.


Microseconds
280,000
300,000
320,000
340,000
360,000
380,000
400,000
420,000
440,000
460,000
480,000
500,000
520,000
540,000
PE 0 (85, 9)
PE 1 (72, 10)
PE 2 (79, 10)
PE 3 (78, 12)
PE 4 (83, 13)
PE 5 (65, 8)
PE 6 (70, 9)
PE 7 (69, 9)
PE 8 (71, 10)
PE 9 (70, 10)
PE 10 (73, 10)
PE 11 (73, 10)
PE 12 (69, 9)
PE 13 (65, 8)
PE 14 (68, 9)
PE 15 (68, 9)

Fig. 2.