**Datenbank Spektrum manuscript No.**
(will be inserted by the editor)

# Iterative Computation of Connected Graph Components with MapReduce

***Preprint, accepted for publication in "Datenbank-Spektrum"***

**Lars Kolb · Ziad Sehili · Erhard Rahm**

Received: date / Accepted: date

**Abstract** The use of the MapReduce framework for iterative graph algorithms is challenging. To achieve high performance it is critical to limit the amount of intermediate results as well as the number of necessary iterations. We address these issues for the important problem of finding connected components in large graphs. We analyze an existing MapReduce algorithm, CC-MR, and present techniques to improve its performance including a memory-based connection of subgraphs in the map phase. Our evaluation with several large graph datasets shows that the improvements can substantially reduce the amount of generated data by up to a factor of 8.8 and runtime by up to factor of 3.5.

**Keywords** MapReduce · Hadoop · Connected Graph Components · Transitive Closure

## 1 Introduction

Many Big Data applications require the efficient processing of very large graphs, e.g., for social networks or bibliographic datasets. In enterprise applications, there are also numerous interconnected entities such as customers, products, employees and associated business activities like quotations and invoices that can be represented in large graphs for improved analysis [17]. Finding connected graph components within such graphs is a fundamental step to cluster related entities or to find new patterns among entities. A connected component (CC) of an undirected graph is a maximal subgraph in which any two vertices are interconnected by a path. Figure 1 shows a graph consisting of two CCs.

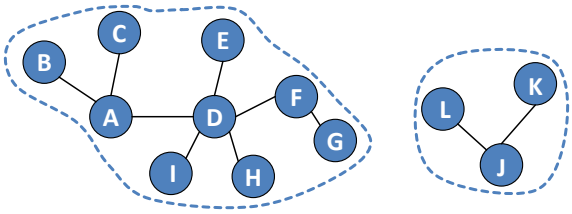



**Fig. 1** Example graph with two connected components.

Efficiently analyzing large graphs with millions of vertices and edges requires a parallel processing. MapReduce (MR) is a popular framework for parallel data processing in cluster environments providing scalability while largely hiding the complexity of a parallel system. We study the problem of efficiently computing the CCs of very large graphs with MR. Finding connected vertices is an inherently iterative process so that a MR-based implementation results in the repeated execution of an MR program where the output of an iteration serves as input for the next iteration. Although the MR framework might not be the best choice for iterative graph algorithms, the huge popularity of the freely available Apache Hadoop implementation makes it important to find efficient MR-based implementations. We implemented our approaches as part of our existing MR-based entity resolution framework [14,13].

The computation of the transitive closure of a binary relation, expressed as a graph, is closely related to the CC problem. For entity resolution, it is a typical post-processing step to compute the transitive closure

Lars Kolb · Ziad Sehili · Erhard Rahm

Institut für Informatik
Universität Leipzig
PF 100920
04009 Leipzig
Germany

E-mail: {kolb,sehili,rahm}@informatik.uni-leipzig.de

of matching entity pairs to find additional, indirectly matching entities. For an efficient MR-based computation of the transitive closure it is important to not explicitly determine all matching pairs which would result in an enormous amount of intermediate data to store and exchange between iterations. For instance, the left CC of Figure 1 with nine entities results in a transitive closure consisting of 36 pairs $(A, B), (B, C), ..., (H, I)$. Instead, we only need to determine the set or cluster of matching entities, i.e. $\{A, B, C, D, E, F, G, H, I\}$, which all individual pairs can be easily derived from, if necessary. Determining these clusters is analogous to determining the CCs with the minimal number of edges, e.g. by choosing a CC/cluster representative, say $A$, and having an edge between $A$ and any other element of the CC, resulting in the eight pairs $(A, B), (A, C), ...(A, I)$ for our example. The algorithms we consider determine such star-like connected components.

For the efficient use of MapReduce for iterative algorithms, it is of critical importance to keep the number of iterations as small as possible because each iteration implies a significant amount of task scheduling and I/O overhead. Second, the amount of generated (intermediate) data should be minimized to reduce the I/O overhead per iteration. Furthermore, one may cache data needed in different iterations (similar to [4]). To this end, we propose and evaluate several enhancements over previous MR-based algorithms. Our specific contributions are as follows:

- We review existing approaches to determine the CCs of an undirected graph (Section 2) and present an efficient MR-based algorithm, CC-MR, (Section 3) as a basis for comparison.
- We propose several algorithmic extensions over CC-MR to reduce the number of iterations, the amount of intermediate data, and, ultimately, execution time (Section 4). A major improvement is to connect edges early during the map phase, so that the remaining work for the reduce phase and further iterations is lowered.
- We perform a comprehensive evaluation for several large datasets to analyze the proposed techniques in comparison with CC-MR (Section 5).

## 2 Related work

### 2.1 Parallel transitive closure computation

An early iterative algorithm, TCPO, to compute the transitive closure of a binary relation was proposed in the context of parallel database systems [22]. It relies on a distributed computation of join and union operations. An improved version of TCPO proposed in [6] uses a double hashing technique to reduce the amount of data repartitioning in each round. Both approaches are parallel implementations of sequential iterative algorithms [3] which terminate after $d$ iterations where $d$ is the depth of the graph. The Smart algorithm [11] improves these sequential approaches by limiting the number of required iterations to $\log d + 1$. Although not evaluated, a possible parallel MR implementation of the Smart algorithm was discussed in [1]. The proposed approach translates *each* iteration of the Smart algorithm into *multiple* MR jobs that must be executed sequentially. However, because MR relies on materializing (intermediate) results, the proposed approach is not feasible for large graphs.

### 2.2 Detection of Connected Components

Finding the connected components of a graph is a well studied problem. Traditional approaches have a linear runtime complexity and traverse the graph using depth first (or breadth first) search to discover connected components [21]. For the efficient handling of large graphs, parallel algorithms with logarithmic time complexity were proposed [10, 20, 2] (see [9] for a comparison). Those approaches rely on a shared memory system and are not applicable for the MR programming model which relies on shared nothing clusters. The authors of [5] proposed an algorithm for distributed memory cluster environments in which nodes communicate with each other to access remote memory. However, the MR framework is designed for independent parallel batch processing of disjoint data partitions.

An MR algorithm to detect CC in graphs was proposed in [7]. The main drawback of this algorithm is that it needs three MapReduce jobs for each iteration. There are further approaches that strive to minimize the number of required iterations [12, 15]. All approaches are clearly outperformed by the CC-MR algorithm proposed in [19]. For a graph with depth $d$, CC-MR requires $d$ iterations in the worst case but needs in practice only a logarithmic number of iterations according to [19]. This algorithm will be described in more detail in the following section. A very similar approach was independently proposed at the same time in [18]. The authors suggest four different algorithms of which the one with the best performance for large graphs corresponds to CC-MR.

Recent distributed graph processing frameworks like Google Pregel [16] rely on the Bulk Synchronous Parallel (BSP) paradigm which segments distributed computations into a sequence of supersteps consisting of a

parallel computation phase followed by a data exchange phase and a synchronization barrier. BSP algorithms are generally considered to be more efficient for iterative graph algorithms than MR, mainly due to the significantly smaller overhead per iteration. However, [18] showed that in a congested cluster, MR algorithms can outperform BSP algorithms for large graphs.

### 2.3 MapReduce

MapReduce (MR) is a programming model designed for parallelizing data-intensive computing in clusters [8]. MR implementations such as Hadoop rely on a distributed file system (DFS) that can be accessed by all nodes. Data is represented by key-value pairs and a computation is expressed employing two user-defined functions, map and reduce, which are processed by a fixed number of map and reduce tasks.

$$map : (key_{in}, val_{in}) \rightarrow list(key_{tmp}, val_{tmp})$$
$$reduce : (key_{tmp}, list(val_{tmp})) \rightarrow list(key_{out}, val_{out})$$

For each intermediate key-value pair produced in the map phase, a target reduce task is determined by applying a partitioning function that operates on the pair's key. The reduce tasks first sort incoming pairs by their intermediate keys. The sorted pairs are then grouped and the reduce function is invoked on all adjacent pairs of the same group. This simple processing model supports an automatic parallel processing on partitioned data for many resource-intensive tasks.

## 3 The CC-MR algorithm

The input of the CC-MR algorithm [19] is a graph $(V, E)$ with a set of vertices $V$ and a set of edges $E \subseteq V \times V$. The goal of CC-MR is to transform the input graph into a set of star-like subgraphs by iteratively assigning each vertex to its "smallest" neighbor, using a total ordering of the vertices such as the lexicographic order of the vertex labels. During the computation of the CCs, CC-MR checks for each vertex $v$ and its (current) adjacent vertices $adj(v)$ whether $v$ is the smallest of these vertices. If this is already the case (*local max state*), all $u \in adj(v)$ are assigned to $v$. A subgraph in the local max state does already constitute a CC but there may be further vertices that belong to this CC but still need to be discovered. If $v$ ist not smaller than all its adjacent vertices, then there is a vertex $u \in adj(v)$ with $u < v$. In this *merge case*, $v$ and $adj(v) \backslash \{u\}$ are assigned to $u$ so that the component of $v$ becomes a component of $u$. The described steps are applied iteratively until no more merges occur.

**Algorithm 1:** CC-MR (reduce)

```
1  reduce(Vertex source, Iterator<Vertex> values)
2      locMaxState ← false;
3      first ← values.next();
4      if source.id < first.id then
5          locMaxState ← true;
6          output(source, first);                    // Forward edge
7      lastId ← first.id;
8      while values.hasNext() do
9          cur ← values.next();
10         if cur.id = lastId then
11             continue ;                        // Remove duplicates
12         if locMaxState then
13             output(source, cur);                  // Forward edge
14         else
15             output(first, cur);                   // Forward edge
16             output(cur, first);                  // Backward edge
17         lastId ← cur.id;
18     if ¬locMaxState ∧ (source.id < lastId) then
19         output(source, first);                    // Backward edge
```

In the following we sketch the MR-based implementation of this approach. We also discuss CC-MR's load balancing to deal with skewed component sizes. Figure 2 illustrates how CC-MR finds the two CCs for the graph of Figure 1. There are three iterations necessary resulting in the two star-like components shown in the lower right corner of Figure 2 with the component centers $A$ and $J$.

### 3.1 MapReduce processing

An edge $v-u$ of the graph is represented by a key-value pair $(v, u)$. To decide for each vertex $v$, whether it is in *local max state* or in *merge state*, $v$ and each $u \in adj(v)$ are redistributed to the same reduce task. As illustrated in Figure 2, for each edge $(v, u)$ of the input graph, the map function of the first iteration outputs a key-value pair $(v.u, u)$ as well as an inversed pair $(u.v, v)$. The output pairs are redistributed to the reduce tasks by applying a partitioning function which utilizes only the first component of the composite map output keys so that all vertices $u$ that are connected to vertex $v$ will be sent to the same reduce task and vice versa. CC-MR makes use of the secondary sorting technique. The reduce tasks sort the incoming key-value pairs by the entire key and group adjacent pairs by the first key component only. A group $v : [val_1, ..., val_n]$ consists of a key for vertex $v$ and a sorted list of values $val_1, ..., val_n$ corresponding to $v's$ neighbors $adj(v)$. For example, the first reduce task in Figure 2 receives group $A : [B, C, D]$

The pseudo-code of the reduce function is shown in Algorithm 1. The reduce function compares each vertex $v$, with its smallest neighbor $first \in adj(v)$. If $v < first$ (*local max state*), then $v$ is already the smallest vertex in the (sub)component and a key-value pair $(v, u)$ is

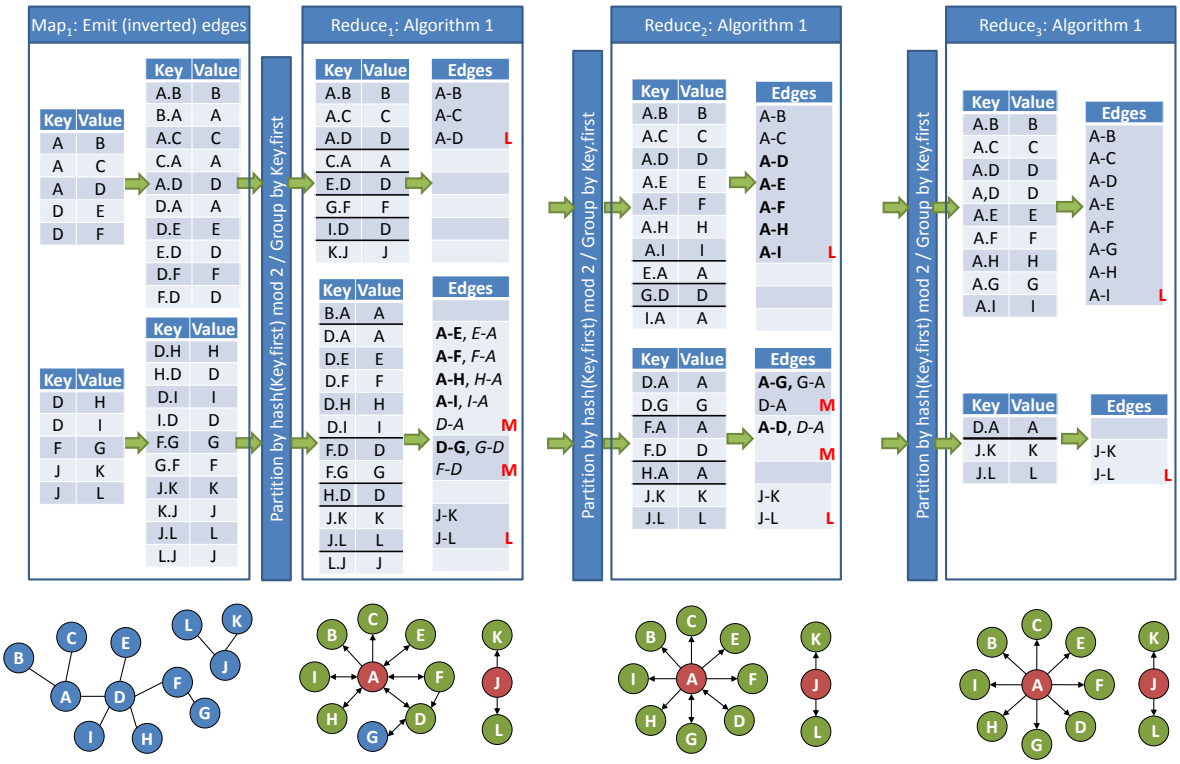



**Fig. 2** Example dataflow (upper part) of the CC-MR algorithm for the example graph of Figure 1. A red $L$ indicates a *local max state*, whereas relevant *merge states* are indicated by a red $M$. Newly discovered edges are highlighted in boldface. The algorithm terminates after three iterations since no new backward edges (italic) are generated. The map phase of each iteration $i > 1$ emits each output edge of the previous iteration unchanged and is omitted to save space. The lower part of the figure shows the resulting graph after each iteration. Red vertices indicate the smallest vertices of the components. Green vertices are already assigned correctly whereas blue vertices still need to be reassigned.

outputted for each value $u \in adj(v)$ (see Lines 4-6 and 12-13 of Algorithm 1). For example in the first iteration of Figure 2, the (sub)components $A : [B, C, D]$ and $J : [K, L]$ are in the *LocMaxState* (marked with an $L$).

For the merge case, i.e. $v > first$, all $u \in adj(v) \setminus \{first\}$ in the value list are assigned to vertex $first$. To this end, the reduce function emits a key-value pair $(first, u)$ for each such vertex $u$ (Line 15). Additionally, reduce outputs the inverse key-value pair $(u, first)$ for each such $u$ (Line 16) as well as a final pair $(v, first)$ if $v$ is not the largest vertex in $adj(first)$ (Line 19). The latter two kinds of reduce output pairs represent so-called *backward edges* that are temporarily added to the graph. Backward edges serve as bridges to connect $first$ with further neighbors of $u$ and $v$ (that might even be smaller than $first$) in the following iterations.

In the example of Figure 2, relevant merge states are marked with an $M$. The reduce input group $F : [D, G]$ of the first iteration results in a newly discovered edge $(D, G)$ and backward edges $(G, D), (F, D)$. Group $D : [A, E, F, H, I]$ also reaches the merge state and generates (amongst others) the backward edge $(D, A)$. In the second iteration, the first reduce task extends the component for vertex $A$ by adding the newly generated neighbors of $A$. The second reduce task merges $A$ and $G$ (for group $D : [A, G]$) as well as $A$ and $D$. Note, that edges might be detected multiple times, e.g. the edge $(A, D)$ is generated by both reduce tasks in the second iteration. Such duplicates are removed in the next iteration (by the first reduce task of the third iteration in the example) according to Line 11 of Algorithm 1.

The output of iteration $i$ serves as input for iteration $i + 1$. The map phase of the following iterations outputs each input edge unchanged (aside from the construction of composite keys), i.e., no reverse edges are generated as in the first iteration. The algorithm terminates when no further backward edges are generated. This can be determined by a driver program which repeatably executes the same MR job by analyzing the job counter values that are collected by the slave nodes and are aggregated by the master node at the end of the job execution. As iteration $i > 1$ *only* depends on the output of iteration $i - 1$, the driver program can replace the input directory of iteration $i - 1$ with the output directory of iteration $i - 1$.

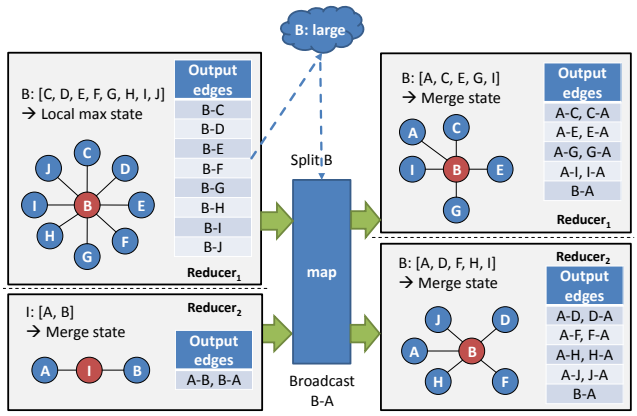



**Fig. 3** Load balancing mechanism of CC-MR.

### 3.2 Load balancing

In the basic approach, all vertices of a component are processed by a single reduce task. This deteriorates the scalability and runtime efficiency of CC-MR if the graph has many small but also a few large components containing the majority of all vertices. CC-MR therefore provides a simple load balancing approach to deal with large (sub)components. Such components are identified in the reduce phase by counting the number of neighbors per group. For large groups whose size exceed a certain threshold, the reduce task records the smallest group vertex $v$ in a separate output file used by the map tasks in subsequent iterations. For a *forward* edge $(v, u)$ of such large components, the map task now applies a different redistribution of the generated key-value pair $(v.u, u)$ by applying the partitioning function on the *second* instead of the first part of the key. This evenly distributes all $u \in adj(v)$ across the available reduce tasks and, thus, achieves load balancing. *Backward* edges $(v, u)$, with $v$ being the smallest element of a large component, are broadcast to all reduce tasks to ensure that all $w \in adj(v)$ can be connected to $u$. Apart from these modifications, the algorithm remains unchanged.

The load balancing approach is illustrated in Figure 3. In the example, the component $B : [C, D, E, F, G, H]$ is identified as a candidate for load balancing. In the following map phase, the neighbors of $B$ are distributed across all (two) reduce tasks based on the neighbor label. This evenly balances the further processing of this large component. The *backward* edge $(B, A)$ is broadcast to both reduce tasks and ensures that all neighbors of $B$ can become $A$'s neighbors in merge states of the following iteration.

## 4 Optimizing CC-MR

Despite its efficiency, CC-MR can be further optimized to reduce the number of iterations and the amount of intermediate data. To this end, we present three extensions in this section. First, we propose to select a CC center based on the number of neighbors rather than simply choosing the vertex with the smallest label. Second, we extend the map phase to already connect vertices there to lower the amount of remaining work for the reduce phase and further iterations. Finally, we propose to identify stable components that do not grow anymore (e.g., $J : [K, L]$ in the example) to avoid their further processing. The proposed enhancements do not affect the general complexity of the original algorithm, CC-MR, so that we further expect a logarithmic number of iterations w.r.t to the depth of the input graph.

### 4.1 Selecting CC centers

CC-MR assigns all interconnected vertices to the smallest vertex of the component, based on the lexical ordering of vertex labels. This is a straight-forward and natural approach that also exploits the built-in sorting of records in the reduce phase. However, this approach does not consider the existing graph structure so that a high number of iterations may become necessary to find all CCs. For example, consider the left part of the initial example of Figure 1 in which the five vertices $E, F, G, H, I$ have to be assigned to vertex $A$ (vertices $B, C, D$ are already neighbors of $A$). If we would use vertex $D$ as the CC center instead, only the three vertices $B, C, G$ need to be reassigned. In the worst case, the input graph consists of a long chain with a smallest vertex located at the head of the chain. In this case, a vertex located in the middle of the chain would be a better component center.

We propose the use of a simple heuristic called CC-MR-VD to select the vertex with the highest degree, i.e. the highest number of direct neighbors, as a component center. While this might not be an optimal solution, it promises to reduce the overall number of vertex reassignments, and, thus, the number of edges generated per iteration and possibly the number of required iterations. If the vertex degrees are known, an additional optimization can be applied. The output of backward edges $(v, u)$ in the reduce phase can entirely be saved if $v$ has vertex degree 1 because $v$ has no further neighbors to be connected with $u$. This simple idea reduces the number of edges to be processed further.

To determine the vertex degree for each graph vertex, CC-MR-VD requires an additional, light-weight MR job as a pre-processing step. This job exploits Hadoop's

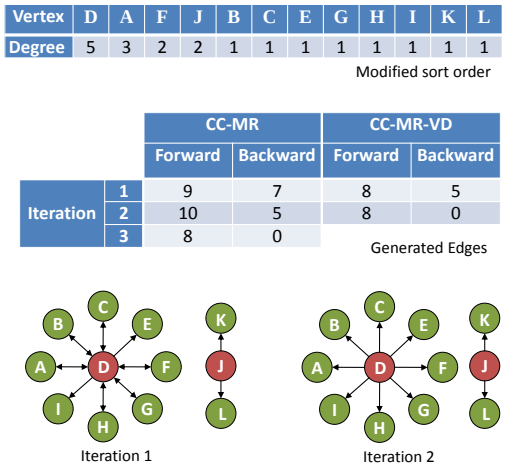



**Fig. 4** Edges generated when taking the vertex degree into account (lower part) and comparison of the number of generated edges (upper part).

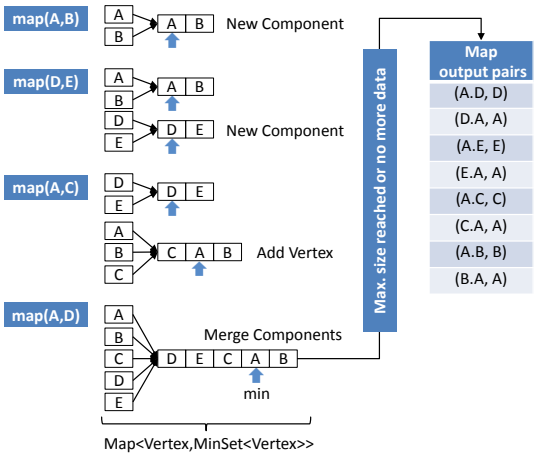



**Fig. 5** Computation of local CCs in the map phase.

*MapFileOutputFormat* to produce indexed files supporting an on-disk lookup of the vertex degree for a given vertex label. The resulting data structure is distributed to all cluster nodes using Hadoop's Distributed Cache mechanism. In the first iteration, the map tasks of the adapted CC-MR computation look up (and cache) vertex degrees of their input edges. Throughout the algorithm, each vertex is then annotated with the vertex degree and vertices are not solely sorted by their labels but first by vertex degree in descending order and second by label in ascending order. The first vertex in this sort order, thus, becomes the one with the highest vertex degree.

For the running example, Figure 4 shows the resulting graph after each iteration when applying CC-MR-VD. Based on the changed sort order, we choose vertex $D$ as the center of the largest component which saves one iteration. Furthermore, the number of generated edges is almost reduced by half.

Unfortunately, CC-MR-VD also has some drawbacks. First, it needs an additional MR job to determine the vertex degrees. Second, the map and reduce output records are larger due the augmentation by the vertex degree. Furthermore, the disk-based random access of vertex degrees in the first map phase introduces additional overhead proportional to the graph size. Our evaluation will show whether the expected savings in the number of edges and iterations can outweigh these negative effects.

## 4.2 Computing local CCs in the map phase

CC-MR applies a stateless processing of the map function where a map task redistributes each key-value pair (i.e. edge) of its input partition to one of the reduce tasks. We propose an extension CC-MR-Mem where a map task buffers a predetermined number of input edges in memory to find already (sub)components among the buffered edges. The determination of such local components uses the same "assign-to-smallest-vertex" strategy as in the reduce tasks. The generated edges are emitted and the buffer is cleared for the next set of input records. Overlapping local CCs that are computed in different rounds or by different map tasks are merged in the reduce phase, as before.

An important aspect of CC-MR-Mem is an efficient computation of local components. We organize sets of connected vertices as hash tables and maintain the smallest vertex per set. Each vertex is mapped to the component set it belongs to as illustrated in Figure 5. A set and its minimal vertex are updated when a new vertex is added or a merge with another set (component) occurs. In the example of Figure 5, the first two edges result in different sets (components) while the third edge $(A, C)$ leads to the addition of vertex $C$ to the first set. The fourth edge $(A, D)$ connects the two components so that the sets are merged. Merging is realized by adding the elements of the smaller set to the larger set and updating the pointers of the vertices of the smaller set to the larger set. Once all edges in the input buffer are processed, the determined sets are used to generate the output pairs for distribution among the reduce tasks (see right part of Figure 5).

CC-MR-Mem thus finds already some components in the map phase so that the amount of work for the reduce tasks is reduced. Furthermore, the amount of intermediate data (number of edges) to be exchanged via the distributed file system as well as the required number of iterations can be reduced. This comes at the cost of increased memory and processing requirements in the

**Algorithm 2:** CC-MR-Mem (map phase)

```
1  map_configure(JobConf job)
2      max ← job.getBufferSize();
3      components ← new HashMap<Vertex,MinSet>(max);
4  map(Vertex u, Vertex v)
5      if components.size() ≥ max then
6          generateOutput();
7      comp1 ← components.get(u);
8      comp2 ← components.get(v);
9      if (comp1 ≠null) ∧ (comp2 ≠null) then
10         if comp1 ≠comp2 then                          // Merge
11             if comp1.size() ≥ comp2.size() then
12                 comp1.addAll(comp2);
13                 foreach Vertex v ∈ comp2 do
14                     components.put(v, comp1);
15             else
16                 comp2.addAll(comp1);
17                 foreach Vertex v ∈ comp1 do
18                     components.put(v, comp2);
19     else if comp1 ≠null then                          // Add Vertex
20         comp1.add(v);
21         components.put(v, comp1);
22     else if comp2 ≠null then                          // Add Vertex
23         comp2.add(u);
24         components.put(u, comp2);
25     else                                              // New component
26         MinSet component= new MinSet(u, v);
27         components.put(u, component);
28         components.put(v, component);
29 map_close()
30     generateOutput();
31 generateOutput()
32     foreach component ∈ components.values() do
33         if ¬component.isMarkedAsProcessed() then
34             component.markAsProcessed();
35             min ← component.min;
36             foreach Vertex v ∈ component do
37                 if v≠min then
38                     output(min.v, v);      // Forward edge
39                     output(v.min, min);  // Backward edge
40     components.clear();
```

map phase. The size of the map input buffer is a configuration parameter that allows tuning the trade-off between additional map overhead and achievable savings. CC-MR-Mem can be combined with the CC-MR-VD approach.

Algorithm 2 shows the pseudo-code of the map function of CC-MR-Mem. Input edges are added to the components map as described above. If its size exceeds a threshold or if there are no further input edges in the map task's input partition, the computed local CCs will be outputted as follows. For each vertex $v \neq c.min$ of a local CC $c$, two key-value pairs $(min.v, v)$ (forward edge) and $(v.min, min)$ (backward edge) are emitted. The partitioning, sorting, and grouping behavior as well as the reduce function is the same as as in Algorithm 1. The only exception is that no backward edges need to be generated by the reduce function, since this is already done in the map phase. Therefore, Lines 16 and 19 of Algorithm 1 are omitted. As for CC-MR, backward edges of large components are broadcast to all reduce tasks to achieve load balancing.

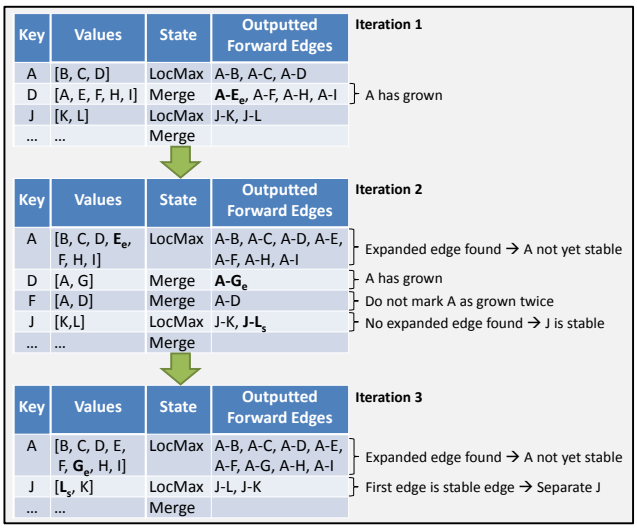



**Fig. 6** Detection and separation of stable components in the reduce phase of the CC-MR algorithm.

## 4.3 Separation of stable components

Large graphs can be composed of many CCs of largely varying sizes. Typically, small and medium-sized CCs are completely discovered much earlier than large components. When a CC does neither disappear (due to a merge with another component) nor grows during an iteration, it will not grow any further and, thus, can be considered as stable. For example, component $J : [K, L]$ is identified during the first iteration of CC-MR and does not change in the second iteration so that no further edges are generated. Hence, this component remains stable until the algorithm terminates. We efficiently want to identify such stable components and separate them from unstable components to avoid their unnecessary processing in further iterations. Stable components are written to different output files which are not read by map tasks of the following iterations.

First, we describe the approach for the original CC-MR (and CC-MR-VD) algorithm. Figure 6 illustrates the approach for the running example. A component with the smallest vertex $v$ grows if there is at least one merge case $u : [v, w, ...]$ with $v < u$. In this case, the reduce function generates new forward edges $(v, w), ...$ as well as corresponding backward edges $(w, v), ...$ . Due to the latter ones, the component may grow in the next iteration. To notify for the next iteration that the component with center $v$ is not yet stable, we augment its *first*

forward edge with a special *expanded flag*, e.g. $(v, w_e)$. In Figure 6, we, thus, augment the first forward edge for the merge case $D : [A, E, F, H, I]$ (resulting in a component with the smallest vertex $A$) with an expanded flag (edge $(A, E_e)$). If there are several merge cases in a reduce task with the same smallest vertex, we set the expanded flag only for one to limit the further overhead for processing components marked as expanded. For example, in the second iteration of Figure 6 there are two merge states $D : [A, G]$ and $F : [A, D]$ processed by the same reduce task (see Figure 2) but only edge $(A, G_e)$ is flagged.

In the reduce function of iteration $i > 1$, we check for each component in *local max state* whether it has some expanded vertex $w_e$ indicating that some vertex was newly assigned in the previous iteration. If such a vertex is *not* found, we consider the component as stable. In the second iteration of Figure 6, the expanded vertex $E_e$ is found and, thus, $A$ can not be separated. Determining whether a component is stable is only known after the last edge for the component has been processed. Since components can be very large, it is generally not possible to keep all its edges in memory at a reduce task. Therefore, the reduce tasks continuously generate the output edges as usual but augment the last output edge with a *stable flag*, e.g. $(v, z_s)$, if $v$ did not grow.

A stable component is then separated in the following iteration. When a map task of the following iteration reads a stable edge $(v, z_s)$, it outputs a $(v.\perp, z_s)$ instead of a $(v.z, z_s)$ pair (alternatively the empty string or Int.MinValue could be used instead of $\perp$). This causes $z_s$ to be the first vertex in the list of $v$'s reduce input values and, thus, the stable component can be found immediately and separated from the regular reduce output. The final result of the algorithm consists of the regular output files generated in the final iteration *and* the additional stable files of *each* iteration. The described approach introduces nearly no additional overhead in terms of data volume and memory requirements.

In Figure 6, there are no expanded vertices for component $J : [K, L]$ in the second iteration so that this component is identified as stable. The *last* output edge, i.e. $J - L_s$, is, thus, augmented by a *stable flag*. Component $J$ is then separated during the third iteration (which is the last one for our small example).

For large components that are split across several reduce tasks for load balancing reasons, *expanded* and *stable* edges need to be broadcast to all reduce tasks. To avoid duplicates in the final output, expanded and stable edges of large components in the *local max state* are outputted only by *one* reduce task, e.g. the reduce task with index zero.

The described approach can also be used for CC-MR-Mem. If a component is known to be stable, the output of backward edges in Line 39 of Algorithm 2 can be saved. An important difference is that components can grow in the map phase as well so that expanded edges $(v.w, w_{e\_map})$ are also generated in this phase. In reduce, a component in *local max state* is considered as stable if there is neither a vertex $w_e$ nor a vertex $w_{emap}$ in the value list. Furthermore, in contrast to expanded edges generated in the reduce phase of the previous iteration, expanded edges that were generated in the map phase of the current iteration need to be forwarded to the next iteration.

## 5 Evaluation

### 5.1 Experimental setup

In our first experiment, we compare the original CC-MR algorithm with our extensions for the first three graph datasets[1] shown in Figure 7(a). The Google Web dataset is the smallest graph containing hyperlinks between web pages. The Patent citations dataset is four times larger and contains citations between granted patents. The Live Journal graph represents friendship relationships between users of an online community. It has a similar number of vertices but about four times as many edges than the Patent graph. The experiments are conducted on Amazon EC2 using 20 worker instances of type c1.medium (providing two virtual cores) and a dedicated master instance of type m1.small. Each node is set up with Hadoop 0.20.2 and a capacity of two map and reduce tasks. The overall number of reduce tasks scheduled per iteration is set to 40.

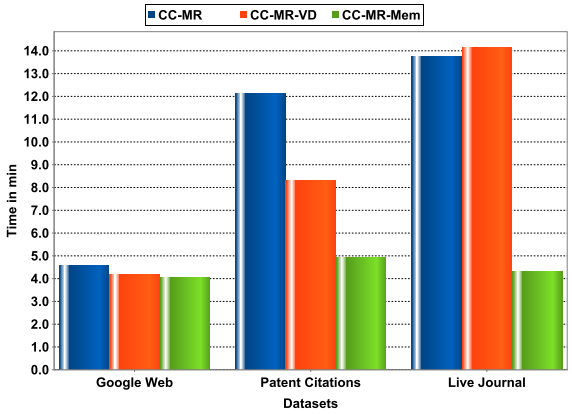
In a second experiment, we evaluate the effect of the early separation of stable components. For this purpose, we use a fourth graph from the Memetracker dataset which tracks web documents (along with their link structure) containing certain frequent quotes or phrases. In contrast to the first three graphs, the Memetracker graph is a sparse graph containing many small CCs and isolated vertices. Due to the large size of the input graph, we increase the cluster size to 40 EC2 worker instances of type m1.xlarge which again can run two map and reduce tasks in parallel.

The third experiment analyzes the scalability of the CC-MR and CC-MR-Mem algorithms for cluster sizes of up to 100 nodes. For this experiment, we again use the Memetracker dataset and EC2 worker nodes of type m1.xlarge. Again each node runs at most two map and reduce tasks in parallel. Thus, for $n$ nodes, the cluster's

[1] `http://snap.stanford.edu/data/`

| Dataset | Nodes | Input edges | #CCs |
|---|---|---|---|
| Google Web | 875,713 | 5,105,039 | 2,746 |
| Patent citations | 3,774,768 | 16,518,948 | 3,627 |
| Live Journal | 4,847,571 | 68.993.773 | 1,876 |
| Memetracker | 185,050,250 | 453,192,771 | 47,021,622 |

(a) Datasets used for the evaluation.




(b) Overall execution times for the three different datasets.

| | CC-MR | | | | CC-MR-VD | | | | CC-MR-MEM | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | Iterations | Overall edges | HDFS Out (GB) | Time (min) | Iterations | Overall edges | HDFS Out (GB) | Time (min) | Iterations | Overall edges | HDFS Out (GB) | Time (min) |
| Google Web | 8 | $\approx 41 \cdot 10^6$ | 0.64 | 4.6 | 7 | $\approx 27 \cdot 10^6$ | 0.78 | 4.2 | 6 | $\approx 9 \cdot 10^6$ | 0.12 | 4.0 |
| Patent Citations | 8 | $\approx 648 \cdot 10^6$ | 11.25 | 12.1 | 8 | $\approx 311 \cdot 10^6$ | 8.98 | 8.3 | 6 | $\approx 112 \cdot 10^6$ | 1.87 | 4.9 |
| Live Journal | 6 | $\approx 640.10^6$ | 9.97 | 13.8 | 6 | $\approx 441 \cdot 10^6$ | 12.33 | 14.1 | 4 | $\approx 87 \cdot 10^6$ | 1.13 | 4.3 |

(c) Summary of results regarding different criteria.

**Fig. 7** Comparison of CC-MR, CC-MR-VD, and CC-MR-Mem for the Google Web, Patent Citations, and Live Journal datasets. The execution times and DFS output volumes of CC-MR-VD include the additional overhead of the vertex degree computation.

map and reduce task capacity is $2 \cdot n$, i.e. adding new nodes leads to additional map and reduce tasks.

All experiments are conducted *with* load balancing turned on. As in [19], we consider a component as large if its size exceeds a threshold of 1% of the number of forward edges generated in the previous iteration. For CC-MR-Mem, the maximum number of vertices that are buffered in memory is set to $20,000$.

## 5.2 Comparison with CC-MR

We first evaluate the three algorithms CC-MR, CC-MR-VD (consideration of vertex degrees) and CC-MR-Mem (computation of local CCs in the map phase) for the first three datasets. We consider the following four criteria: number of iterations, execution time, overall number of edges written to the DFS (across all iterations) as well as the corresponding overall data volume. The results are listed in Figure 7(c) and illustrated in Figure 7(b).

The results show that CC-MR-Mem outperforms CC-MR for all cases. The improvements in execution time increase with larger datasets up to a factor of 3.2 for the Live Journal graph. Furthermore, CC-MR-Mem significantly reduces the overall amount of data written to the DFS by up to a factor of 8.8 for Live Journal. It strongly profits from the density of the input graphs and is able to already connect overlapping subcomponents in the map input partitions of each iteration that otherwise would have been connected in the reduce phase of later iterations. This causes fewer generated forward and backward edges in the reduce phase which increases the probability of an earlier termination of the computation. For each dataset, CC-MR-Mem needs two iterations less than CC-MR.

For all datasets, CC-MR-VD's consideration of vertex degrees leads to a significant reduction in the number of generated edges (at the cost of an additional analysis job). The number of merge cases could be reduced which in turn led to improved execution times for Google Web and the Patent Citations graph. However, the resulting data volume is mostly larger than for CC-MR since all vertices (which are represented by an integer in the datasets) are augmented with their vertex degree (an additional integer). Note, that for larger vertex representations (e.g. string-valued URLs of websites) this might not hold. For the large Live Journal graph, CC-MR-VD suffered from the overhead of pre-computing the vertex degrees before and reading them from the distributed cache during the first iteration, which is its main drawback for large graphs. Hence, CC-MR-VD is in its current form not a viable extension for large graphs.

| | CC-MR | | CC-MR-Mem | |
|---|---|---|---|---|
| | regular | stable | regular | stable |
| Iterations | 11 | 11 | **9** | **8** |
| Overall Edges | ≈4.71·10⁹ | ≈4.15·10⁹ | ≈1.87 ·10⁹ | ≈1.30 ·10⁹ |
| HDFS out (GB) | 552.4 | 467.3 | 212.0 | 133.7 |
| Time (min) | 74.7 | 71.6 | 28.8 | 21.6 |

**Fig. 8** Results of CC-MR and CC-MR-Mem for the Memetracker dataset with and without separation of stable components.

### 5.3 Separation of stable components

In our second experiment, we study the effects of the early separation of stable components that do not grow further in the following iterations. To this end, we utilized a graph derived from the Memetracker dataset which has a significantly lower degree of connectivity compared to the other datasets. Without separating stable components, the execution time of CC-MR-Mem to find all CCs is by a factor of 2.6 lower than for CC-MR (Figure 8).

With separation turned on, we observe only a small improvement of 4% of the execution time for CC-MR. Apparently, due to the MR overhead for job and task submission, the amount of $85GB$ which is saved across 11 iterations has only a small influence on the overall runtime for a cluster consisting of 40 nodes. However, CC-MR-Mem strongly benefits from the separation of stable components which leads to an improvement of 25%. Compared to the regular CC-MR algorithm, the execution time is improved by a factor of 3.5 (21.6 vs 74.7 minutes). An interesting observation is that the number of iterations could be reduced from 9 to 8 for CC-MR-Mem. This is caused by the fact that due to the separation of stable components from the regular reduce output, the input data of a map task does no longer contain stable components. This in turn leads to a higher probability that two components that would be separated by a stable components otherwise are merged in the map phase already. Compared to the regular CC-MR-Mem, the amount of data written to HDFS could be reduced by 37% which improved the execution time by 25%.

### 5.4 Scalability

The third experiment focuses on the scalability of the CC-MR algorithm and the CC-MR-Mem extension. We therefore use the large Memetracker dataset and vary the cluster size $n$ from 1 up to 100 nodes. Figure 9 shows the resulting execution times and speedup values.

The execution of the CC-MR algorithm did not succeed for $n = 1$. This is because in some iteration, the sizes of the input graph, the output of the previous iteration, the map output (containing replicated backward edges for load balancing purposes), and the reduce output exceeded the available disk capacity. To this end, CC-MR's execution time for $n = 1$ was estimated as twice as long as for $n = 2$. By contrast, due to its reduced amount of generated data, CC-MR-Mem was able to compute all CCs of the input graph on a single node and did even complete earlier than CC-MR running on two nodes.

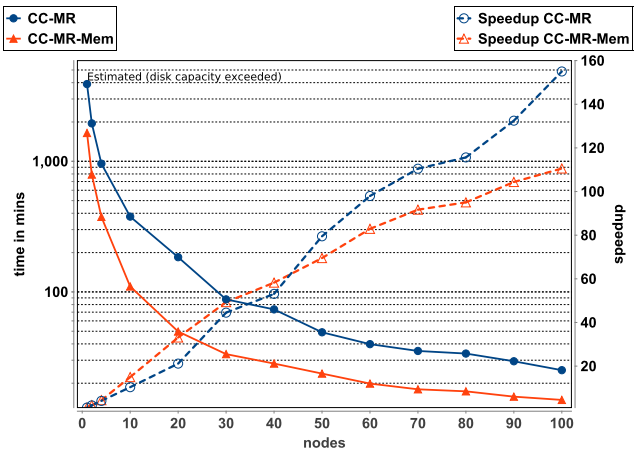



**Fig. 9** Execution times and speedup values using CC-MR and CC-MR-Mem for the Memetracker dataset (without separation of stable components).

Overall, CC-MR-Mem clearly outperformed CC-MR for all cluster sizes. The super-linear speedup values in Figure 9 are caused by the execution times of the single node case with only two parallel map and reduce tasks. Here, both approaches heavily suffer from load balancing problems in early iterations. These load balancing issues are resolved with larger configurations leading to a better utilization of the available nodes. Up to 60 nodes, the execution time for CC-MR-Mem was a factor of 2 to 4 better than with CC-MR. For example, an execution time of 100 minutes is achieved with ten nodes for CC-MR-Mem vs. about 30 nodes for CC-MR. The results show that CC-MR-Mem keeps its effectiveness even for an increasing number of map tasks which in principle reduces the local optimization potential per map task. This is influenced by the fact, that CC-MR-Mem determines local components within fixed-sized (20.000) groups of edges and *not* within whole map input partitions. Still, the relative improvement of CC-MR-Mem over CC-MR decreases somewhat when increasing the number of nodes beyond a certain level. In our experiment, the relative improvement was a factor of 2 for 60 nodes and a factor of 1.7 for 100 nodes (14.9 vs. 25.2 minutes).

## 6 Summary

The computation of connected components (CC) for large graphs requires a parallel processing, e.g. on the popular Hadoop platform. We proposed and evaluated three extensions over the previously proposed MapReduce-based implementation CC-MR to reduce both, the amount of intermediate data and the number of required iterations. The best results are achieved by CC-MR-Mem which connects sub-components in both the reduce *and* in the map phase of each iteration. The evaluation showed that CC-MR-Mem significantly outperforms CC-MR for all considered graph datasets, especially for larger graphs. Furthermore, we proposed a strategy to early separate stable components from further processing. This approach introduces nearly no additional overhead but can significantly improve the performance of CC-MR-Mem for sparse graphs.

## References


1. Afrati, F.N., Borkar, V.R., Carey, M.J., Polyzotis, N., Ullman, J.D.: Map-Reduce Extensions and Recursive Queries. In: Proc. of Intl. Conference on Extending Database Technology, pp. 1–8 (2011)
2. Awerbuch, B., Shiloach, Y.: New Connectivity and MSF Algorithms for Shuffle-Exchange Network and PRAM. IEEE Transactions on Computers **36**(10), 1258–1263 (1987)
3. Bancilhon, F., Maier, D., Sagiv, Y., Ullman, J.D.: Magic Sets and Other Strange Ways to Implement Logic Programs. In: Proc. of Symposium on Principles of Database Systems, pp. 1–15 (1986)
4. Bu, Y., Howe, B., Balazinska, M., Ernst, M.D.: The HaLoop approach to large-scale iterative data analysis. VLDB Journal **21**(2), 169–190 (2012)
5. Bus, L., Tvrdík, P.: A Parallel Algorithm for Connected Components on Distributed Memory Machines. In: Proc. of European PVM/MPI Users' Group Meeting, pp. 280–287 (2001)
6. Cheiney, J.P., de Maindreville, C.: A Parallel Transitive Closure Algorithm Using Hash-Based Clustering. In: Proc. of Intl. Workshop on Database Machines, pp. 301–316 (1989)
7. Cohen, J.: Graph Twiddling in a MapReduce World. Computing in Science and Engineering **11**(4), 29–41 (2009)
8. Dean, J., Ghemawat, S.: MapReduce: Simplified Data Processing on Large Clusters. In: Proc. of Symposium on Operating System Design and Implementation, pp. 137–150 (2004)
9. Greiner, J.: A Comparison of Parallel Algorithms for Connected Components. In: Proc. of Symposium on Parallelism in Algorithms and Architectures, pp. 16–25 (1994)
10. Hirschberg, D.S., Chandra, A.K., Sarwate, D.V.: Computing Connected Components on Parallel Computers. Communications of the ACM **22**(8), 461–464 (1979)
11. Ioannidis, Y.E.: On the Computation of the Transitive Closure of Relational Operators. In: Proc. of Intl. Conference on Very Large Databases, pp. 403–411 (1986)
12. Kang, U., Tsourakakis, C.E., Faloutsos, C.: PEGASUS: A Peta-Scale Graph Mining System. In: Proc. of Intl. Conference on Data Mining, pp. 229–238 (2009)
13. Kolb, L., Rahm, E.: Parallel Entity Resolution with Dedoop. Datenbank-Spektrum **13**(1), 23–32 (2013)
14. Kolb, L., Thor, A., Rahm, E.: Dedoop: Efficient Deduplication with Hadoop. Proceedings of the VLDB Endowment **5**(12), 1878–1881 (2012)
15. Lattanzi, S., Moseley, B., Suri, S., Vassilvitskii, S.: Filtering: A Method for Solving Graph Problems in MapReduce. In: Proc. of Symposium on Parallelism in Algorithms and Architectures, pp. 85–94 (2011)
16. Malewicz, G., Austern, M.H., Bik, A.J.C., Dehnert, J.C., Horn, I., Leiser, N., Czajkowski, G.: Pregel: A System for Large-Scale Graph Processing. In: Proc. of the Intl. Conference on Management of Data, pp. 135–146 (2010)
17. Petermann, A., Junghanns, M., Mueller, R., Rahm, E.: BIIIG: Enabling Business Intelligence with Integrated Instance Graphs. In: Proc. of Intl. Workshop on Graph Data Management (GDM) (2014)
18. Rastogi, V., Machanavajjhala, A., Chitnis, L., Sarma, A.D.: Finding connected components in map-reduce in logarithmic rounds. In: Proc. of Intl. Conference on Data Engineering, pp. 50–61 (2013)
19. Seidl, T., Boden, B., Fries, S.: CC-MR - Finding Connected Components in Huge Graphs with MapReduce. In: Proc. of Machine Learning and Knowledge Discovery in Databases, pp. 458–473 (2012)
20. Shiloach, Y., Vishkin, U.: An O(log n) Parallel Connectivity Algorithm. J. Algorithms **3**(1), 57–67 (1982)
21. Tarjan, R.E.: Depth-First Search and Linear Graph Algorithms. SIAM Journal on Computing **1**(2), 146–160 (1972)
22. Valduriez, P., Khoshafian, S.: Parallel Evaluation of the Transitive Closure of a Database Relation. Intl. Journal of Parallel Programming **17**(1), 19–37 (1988)