SSCA2 Kernel 4 Algorithm Description

Robert L. Bocchino Jr.

September 2007

This document describes my implementation of SSCA2 Kernel 4 (graph clustering), based on the reference implementation provided in version 1.0 of the benchmark. I describe both the explicit locks version and the STM version. Please note that *both versions are speculative*. However, the locks version encodes the speculation "manually" at the application level, while the STM version leverages the speculative mechanism built into the STM runtime. Similarly, the locks version uses explicit locks, while the STM version uses the locks built into the STM. (In theory, the STM implementation could also be lock-free, but mine isn't.)

Both versions of the algorithm use the following processor-local, function-global data structures, one on each processor p:

- V_p: The set of vertices local to p. The vertices adjacent to each $v \in V_p$ are stored in an adjacency list representation, also on p. The adjacency list is modified when a vertex in the list is added to a cluster ("claimed") so that subsequent users of the list will no longer "see" the link to the cluster (this effectively removes each vertex in the cluster from the graph).
- *candidateSet*: The set of candidate vertices for inclusion in the cluster we are currently computing
- *adjSet*: The set of vertices adjacent to the candidate set
- *adjCache*: A map that locally stores the adjacency list of each vertex that we have locked. The set of keys in the map also records which vertices we have locked.
- *cluster*: The actual current cluster, a subset of *candidateSet*
- The *output*, consisting of a set of clusters.

There is also an array (not explicitly shown) that associates a *state* with each vertex.

1 Explicit Locks Implementation

In the explicit locks version, the possible vertex states are CLAIMED, which means that some processor has permanently added this vertex to a cluster; AVAILABLE, which means that the vertex is available for any processor to add to a cluster; and LOCKED, which means that some processor has provisionally added the vertex to its cluster, but has not yet committed the cluster. The LOCKED state also encodes which processor owns the lock; this information is used for a simple priority-based deadlock avoidance protocol. The vertex state array is distributed across the processors in the same way as the vertices and is visible to all processors.

1.1 Main Algorithm

Pseudocode. Figure 1 shows the pseudocode for the main algorithm. Each processor iterates over its local vertices, as shown in line 4. If the vertex is claimed, then the processor moves on. If the vertex is locked, then the processor keeps trying that vertex until it is claimed or released. Once a processor locks a local vertex, it attempts to build a *candidate set* starting with that vertex, as shown in line 7. This procedure may fail, because it involves locking more vertices. If it fails, then the processor unlocks everything and tries again. Otherwise, it uses the candidate set to compute a cluster, updates

```
SSCA2K4( Graph $G = (V, E)$ )
 1   Distribute $V$ evenly over the processors $P$ into sets $V_p$
 2   for each processor $p$ in parallel
 3   do while $V_p \neq \emptyset$
 4       do adjCache $\leftarrow \emptyset$
 5          $v \leftarrow$ remove one element of $V_p$
 6          if $v$ is claimed then continue
 7          if LOCK($v$) = FAIL then add $v$ to $V_p$; continue
 8          if COMPUTE_CANDIDATE_SET($v$) = FAIL then ABORT(); add $v$ to $V_p$ ; continue
 9          cluster $\leftarrow$ COMPUTE_CLUSTER()
10          UPDATE_GRAPH()
11          add cluster to output
12   Merge output of all processors
```

Figure 1: Pseduocode for SSCA2 Kernel 4, locks implementation

the vertex data (state and adjacency lists) possibly residing on other processors, and adds the cluster to its local output, as shown in lines 8–10.

Supporting functions. The supporting functions are *lock*, *compute_candidate_set*, *update_graph*, *abort*, and *compute_cluster*. Section 1.2 explains the function *lock*, section 1.3 explains the function *compute_candidate_set*, and section 1.4 explains the function *update_graph*. The function *abort* sets the state of each vertex in *adjCache* to AVAILABLE, using the same per-processor aggregation pattern as *lock*. It is very straightforward, because it cannot fail (unlike *lock*). I omit its pseudocode. The function *compute_cluster* is a local computation: if *candidateSet* is sufficiently large (bigger than a prescribed minimum) it sets the cluster to be the first n elements of *candidateSet* such that the number of adjacencies between those elements and the whole set is minimized. If *candidateSet* is not sufficiently large, it just uses the whole candidate set. I omit the pseudocode for this function. For further details, see the SSCA2 specification.

1.2 The *lock* Function

Figure 2 shows the pseudocode for the function *lock*. We lock only the vertices not present in *adjCache*, because any vertex in *adjCache* has already been locked. We sort the vertices to lock by processor. For each processor, we send out all vertices in one message, and we perform the lock operation locally on the destination processor. We get back an array of results. We then iterate through the array, checking the status of each vertex.

If the status of a vertex is CLAIMED, then we will never be able to acquire this vertex (some other processor has claimed it), so we give up on this cluster and return FAIL. If the status is AVAILABLE, then the local lock operation has locked the vertex for us and returned its adjacency list; we put the list into *adjCache* for later use. If the status is neither CLAIMED nor AVAILABLE, then some other processor has locked the vertex, and the status tells us which processor that is. We compare the status with our own processor ID to see our relative priority to the lockholder. If we are lower priority, we abort. If we are higher priority, we put the vertex back into the vertex set and try to get it on the next pass, knowing that the lockholder will eventually succeed or abort (it will never wait for a higher priority processor, and in particular will never be connected to us by a chain of wait-for edges). We keep iterating over the vertex set until it is empty.

1.3 The *compute_candidate_set* Function

Figure 3 shows the pseudocode for the function *compute_candidate_set*. This function computes a "candidate set" (a superset of a cluster) starting with a single "seed" vertex v. We maintain an *adjSet*, which consists of all vertices adjacent to the candidate set. Initially, the candidate set contains v, and *adjSet* contains the adjacencies of v, which we look up in the *adjCache*. We try to lock the *adjSet* and abort if any of the locks fails.

```
LOCK( vertices $V$ )
 1  while $V \neq \emptyset$
 2  do sort $V \setminus$ KEYS($adjCache$) by processor into sets $V_p$
 3      $V \leftarrow \emptyset$
 4      for each processor $p$ such that $V_p$ is nonempty
 5      do $results \leftarrow$ on $p$ LOCK_LOCAL($V_p$)
 6          for each $r \in results$
 7          do if $r.status$ = CLAIMED then return FAIL
 8              else if $r.status$ = AVAILABLE then $adjCache[v] \leftarrow r.adjList$
 9              else if lockholder is higher priority then return FAIL
10              else add $v$ to $V$
11  return SUCCEED
```

Figure 2: Pseudocode for the *lock* function

```
COMPUTE_CANDIDATE_SET( vertex $v$ )
 1   $candidateSet \leftarrow \{v\}$
 2   $adjSet \leftarrow adjCache[v]$
 3   if LOCK($adjSet$) = FAIL then return FAIL
 4  while $adjSet \neq \emptyset$ and $|candidateSet| <$ MAX_CLUSTER_SIZE
 5  do $v \leftarrow$ NEXT_CANDIDATE($adjSet$)
 6      if LOCK($adjCache[v] \setminus candidateSet$) = FAIL then return FAIL
 7      add $v$ to candidateSet
 8      remove $v$ from $adjSet$
 9      add $adjCache[v] \setminus candidateSet$ to $adjSet$
10  return SUCCEED
```

Figure 3: Pseudocode for the *compute_candidate_set* function

We iteratively add points to the candidate set. In each iteration, we do the following:

1. Select a next candidate from the current *adjSet*. The selection uses a simple heuristic to look for a vertex that is most tightly connected to the cluster. I omit the pseudocode, but more detail can be found in the SSCA2 specification.
2. Try to lock the adjacencies of v that are not already in our candidate set. One of these vertices will become our next candidate. If this operation fails, abort.
3. Add v to the candidate set and remove it from *adjSet*. Add the new adjacencies of v (i.e., adjacencies that aren't already in the candidate set) to the candidate set.

We continue iterating until (1) there are no more vertices adjacent to the candidate set or (2) the candidate set is sufficiently large.

1.4 The *update_graph* Function

Figure 4 shows the pseudocode for the function *update_graph*. This function updates the state and adjacency information for three kinds of vertices: (1) vertices in the cluster; (2) vertices in the candidate set but not in the cluster; and (3) vertices adjacent to the candidate set. We sort each set of vertices by processor. Then we iterate over the processors that have any vertices to update. For each processor p, we compute the updated adjacencies of the vertices in set (3), removing any links to the cluster. This procedure effectively removes the cluster vertices from the graph.

```
UPDATE_GRAPH( cluster, adjSet, adjCache )
1   sort cluster by processor into sets $C_p$
2   sort candidateSet $\setminus$ cluster by processor into sets $C'_p$
3   sort adjSet by processor into sets $A_p$
4   for each $p$ such that $C_p \cup C'_p \cup A_p \neq \emptyset$
5   do $M \leftarrow \emptyset$
6       for each $a \in A_p$
7       do $M[a] \leftarrow$ updated adjacencies of $a$, disregarding links to cluster
8        on $p$ UPDATE_GRAPH_LOCAL$(C_p, C'_p, A_p, M)$
```

Figure 4: Pseudocode for the *update_graph* function

We then call the function *update_graph_local* on processor p. This function accepts all the update information for p in one communication. It then iterates over all the vertices that need updating, and performs the appropriate update. For vertices in set (1), this means setting the state to CLAIMED; for set (2), it means setting the state to AVAILABLE; and for set (3), it means updating the adjacencies and setting the state to AVAILABLE.

2 STM Implementation

The STM version is similar to the explicit locks version, except for the following:

1. The possible vertex states are CLAIMED and AVAILABLE. There is no LOCKED vertex state.
2. Instead of locking vertices and checking locks, the implementation just reads and writes the states and adjacency lists. The STM's conflict detection mechanism ensures that the same semantics as in the locking version will be preserved.
3. There is no explicit abort code. Instead, the STM abort mechanism provides the rollback.

The code for the main algorithm of the STM version is shown in Figure 5. Note the absence of the locking function and any abort code. Note also that we must be careful to reset local data structures at the beginning of the transaction, so that the abort works correctly. In particular, the adjacency cache must be cleared at the start of every transaction.

The supporting functions are similar to the locks version, but simpler. Instead of the *lock* function (Section 1.2), we use a *cache* function that pulls the adjacency sets for the requested vertices into the *adjCache* (Figure 6). The local function executed on each remote processor uses *stm_read* to perform the reads, so that the transaction can abort in the case of a conflict. Note that we do not need to check for LOCKED states, because the transactions guarantee serializability. Nor do we need to check for CLAIMED states, because if we ever follow an edge in the graph to a claimed vertex, a conflict will occur: when the transaction that claimed the vertex commits, it will cut the link that we followed, causing a read-write conflict. We use a *compute_candidate_set* function simlar to the locks version (Section 1.4), except that we call *cache* instead of *lock* and do not explicitly check for conflicts (Figure 7). Finally, the *update_graph* function is identical to the function shown in Figure 4. However, the *update_graph_local* function uses *stm_write* to update the state and adjacencies, so that read-write conflicts across transactions will be detected. (It would probably be sufficient to use *stm_read* and *stm_write* only on the states, and not on the adjacency lists, because the states "guard" the lists. However, it would be very difficult for a compiler to prove this fact.)

```
SSCA2K4( Graph $G = (V, E)$ )
 1  Distribute $V$ evenly over the processors $P$ into sets $V_p$
 2  for each processor $p$ in parallel
 3  do while $V_p \neq \emptyset$
 4      do STM_START
 5          adjCache $\leftarrow \emptyset$
 6          $v \leftarrow$ remove one element of $V_p$
 7          if $v$ is claimed then continue
 8          COMPUTE_CANDIDATE_SET($v$)
 9          cluster $\leftarrow$ COMPUTE_CLUSTER()
10          UPDATE_GRAPH()
11          STM_COMMIT
12          add cluster to output
13  Merge output of all processors
```

Figure 5: Pseudocode for SSCA2 Kernel 4, STM implementation

```
CACHE( vertices $V$ )
1  sort $V \setminus$ KEYS(adjCache) by processor into sets $V_p$
2  for each processor $p$ such that $V_p$ is nonempty
3  do results $\leftarrow$ on $p$ CACHE_LOCAL($V_p$)
```

Figure 6: Pseudocode for the *cache* function

```
COMPUTE_CANDIDATE_SET( vertex $v$ )
1  candidateSet $\leftarrow \{v\}$
2  adjSet $\leftarrow$ adjCache[$v$]
3  CACHE(adjSet)
4  while adjSet $\neq \emptyset$ and | candidateSet | $<$ MAX_CLUSTER_SIZE
5  do $v \leftarrow$ NEXT_CANDIDATE(adjSet)
6      CACHE(adjCache[$v$]$\setminus$candidateSet)
7      add $v$ to candidateSet
8      remove $v$ from adjSet
9      add adjCache[$v$]$\setminus$ candidateSet to adjSet
```

Figure 7: Pseudocode for the *compute_candidate_set* function in the STM implementation