Manual Speculative Graph Algorithm

Shule Li

1. Introduction

Parallel operations on traditional data structures often require threads to "own" the entire data so that atomicity can be ensured during the operation. The ownership can be achieved by implementing locks in the said data structure so that once a method is invoked, the exclusivity is granted to the caller thread. However, not all operations on a given data structure requires locking an entire data structure. For instance, consider an algorithm where a new node needs to be inserted into a sorted list. In such algorithm, once we have found the desired place for the new node, we only have to lock its predecessor and successor to make sure while inserting, the two neighboring nodes are not moved or deleted. Imagine a long list with multiple threads trying to insert new nodes into it. The majority of each threads are traversing the list for most of the time. This traversing operation does not have to be mutually exclusive. Naturally, a speculative algorithm can be designed so that during the traversing phase, threads do not ask for exclusivity to the data object. It is once they have found out what they want to do, i.e. where to insert, that they try to obtain the locks of needed objects stored in said data structure.
This speculative mechanics is a design pattern that is suitable for other parallel algorithms that require a similar "search and do something" operation. The pattern can be extended to cases where multiple objects are needed to be modified. Using the sorted list example, a thread may want to insert more than one object into the list. To achieve this, it can search and secure the first location by locking the first two neighbors, finish inserting the first object, and continue on. However, if one requires that the insertion of these three objects to be "all or none", then if this thread finds out later that it cannot insert the second object, it has to go back to delete the first insertion. Conversely, one can imagine the thread to traverse the list once, marking all the locations it needs to perform the insertion, and only after that does it try to acquire the locks of all the needed nodes. The "do first, commit later" idiom can be seen as similar to the lazy acquire mechanism used in transactional memory systems, with locking performed instead of Compare and Swap. Another example is in a graph algorithm, a thread may need to insert a vertex into the data structure with several desired neighbors. The thread needs to first perform a graph search to allocate all the neighbors, and lock them after the search is done. In this paper, we apply this speculation technique to graph data structure, and rationalize the performance. In Section 2, we describe the data abstraction as well

as the speculation algorithm. In Section 3, we discuss the performance results. In Section 4, we summarize and propose possible future directions.
There are many approaches towards achieving better parallelism in this context. Each approach has its merits and shortcomings. For instance, nonblocking data structures have seen great success in the past years. Its inherent optimism can be seen as an extention of the speculation technique just described, and by avoiding using any locking mechanism, it also avoids traditional weaknesses of locks such as deadlocking and convoying. However, creating nonblocking data structures that exactly suite one's need can be difficult. The easiest to create are probably queues and stacks where there is an inherent ordering inferred in the data structure. The speculation technique discussed in this paper is not suitable for such data structure since it requires a section to speculate on and therefore can be moved outside of the critical section. Such section usually involves traversing through the data structure or other similar "preparation" phase that can be executed concurrently without affecting atomicity.

2. Speculative Implementation

C++0x provides threading, atomic operations, locking and user specific memory consistency model. We base our study on these methods provided by this standard, but the results can be easily extended to other language implementations.
Vectors The undirected graph used in this paper is implemented using adjacency list: each vertex store a header pointer to a list (neighbor list hereafter) of edges. This list is sorted by the weight of each edge so that the closest neighbor always appear first. The key value can be anything such as a string as persons' names in a graph describing a social relationship network once a way of comparing or assigning "values" to the vertices is provided, however, in this paper, we assume the key values are integers. Vertex contains a mutual exclusive lock that can be granted to an outside caller once the provided lock method is invoked. Vertex also contains a version number (*vn* hereafter). Whenever changes are made to the vertex, its *vn* is atomically incremented so that caller threads can know whether the vertex has been modified. Methods defined in a vertex class includes locking, which checks if the current version number matches the input and lock the object, adding or deleting neighbors from neighbor list, checking if a certain vertex is neighbor, getting its key values, print out all neighbors, etc. We only show pseudo code of several of these methods here.
Edges Edge classes store an integer *weight* and a vertex pointer pointing to the neighbor. *next* and *prev* operators are needed for list operation. Note that depending on the granularity of locking, it is possible that one may be able to simplify the implementation. If the granularity is at the vertex level, then all the edges are automatically locked once the vertex is locked. This requires one to have all the read-write operations performed on the vertex level. This is indeed what happens here: the only methods edge class can invoke are returning the pointer to the neighbor or returning its weight. Edge modifications are done

by methods in vertex class. The data abstraction of vertex class are shown in the following code section (only several important methods are displayed, with constructor and destructor omitted).

```
class vertex {
int key; //key value
int nl; //length of adjacency list
public:
atomic_int vn; //version number
mutex m; //mutex
edgenode *headlink, *taillink; //pointers to adjacency list
edge* findneighbor(vertex* v) const; //return the edge pointing to v
bool isisolated() const; //check if there is any neighbor
void addlink(edge* n); //connect edge n with this vertex
void removeneighbor(vertex* v); //remove the edge connecting this to v
void isolate(); //remove all edges in the adjacency list
bool locking(int i); //lock this vertex if version number equals to the input
void unlocking(); //unlock this vertex
};
```

```
edge* vertex::findneighbor(veretex* n){
if vertex is isolated:
     return NULL; //return if no neighbors found
edgenode* probe = headlink;
repeat{
     if(v == probe->getneighbor()) //return neighbor pointed to by this edge
     return probe;
     probe = probe->next;
} while probe is not NULL
     return NULL;
};
```

```
void vertex::addlink(edgenode* n){
if empty list:
     headlink = taillink = n
if already in list:
     return;
int temp = n->getlength();
edgenode* probe = this->headlink;
repeat{ //find the right place to insert
     if(probe->getlength() > temp):
          break;
     probe = probe->next;
} while probe is not NULL
insert n before probe;
};


void vertex::removeneighbor(vertex* v){
if empty list:
     return;
edgenode* probe = this->findneighbor(v); //check if the neighbor is in this list
if(probe!=NULL){
     if(headlink == taillink):
          headlink = taillink = NULL;
     else if probe == headlink:
          probe->next->prev = NULL;
          headlink = probe->next;
     else if probe == taillink:
          probe->prev->next = NULL;
          taillink = probe->prev;
     else:
          probe->prev->next = probe->next;
          probe->next->prev = probe->prev;
}
delete probe;
nl--;
vn++;
};


void vertex::isolate(){
edgenode *probe, *tmp;
probe = headlink;
while(probe!=NULL){
     if probe->next == NULL:
          headlink = taillink = NULL;
          delete probe;
          nl--;
          vn++;
          break;
     tmp = probe;
     probe->next->prev = NULL;
     probe = probe->next;
     headlink = probe;
     delete tmp;
     nl--;
     vn++;
}
```

```
};


bool vertex::locking(int i){
if atomic_load(vn)!=i:
     return false;
m.lock();
if atomic_load(vn)==i:
     return true;
else:
     m.unlock();
     return false;
}
```

Graph Next we describe methods defined in our graph class. We consider four types of operations: inserting, deleting, transplanting and reducing. Each of them requires a graph search.
Insertion Insertion is defined as an operation that adds a vertex into the graph with desired neighbors. During insertion, we first search the graph to find desired neighbors and record their location and version number into a pair and insert this pair into a set. After the traversal is finished, we try to obtain the locks to the locations located in the set so that we can safely insert the new vertex. At the end of the speculative search section, we traverse the set to invoke the locking method provided in vertex class and lock the desired neighbors one by one. Upon finishing, the set is unlocked and erased.
Removal Removal of a vertex is implemented in a similar fashion as insertion.
Transplanting Imagine a graph describing relationships in an organization. Once a personnel is moved from one department to another, he or she may form a complete new working relationship at the new place. Transplanting allows us to move a vertex from one part of the graph to another, forming new connections there while cutting the old connections. The implementation requires us to cut the vertex from its old position, search the graph for a desired new position and insert it again. Intuitively, this method is a combination of cutting and inserting a vertex. Cutting does not require speculation: we just get the locks to the neighbors and delete the related edges.

Reducing Graph reduction has important role in real life applications that require us to extract information from a graph. Such algorithm is usually composed of detecting reoccurring patterns in a graph and merging vertices while contracting the edges connecting them. In this study, we consider a simplification. Since edge weights can be used to denote the type of relationship, graph reducing can be seen as a contraction operation that affects the entire graph. The procedure consists of search the graph to find the relations one wants to reduce, lock the related vertices, and then merge them by contracting the edges. A diagram of vertex merging operation is shown in Fig.2. During contraction, we search through the graph to find out the vertices needing merge, and record

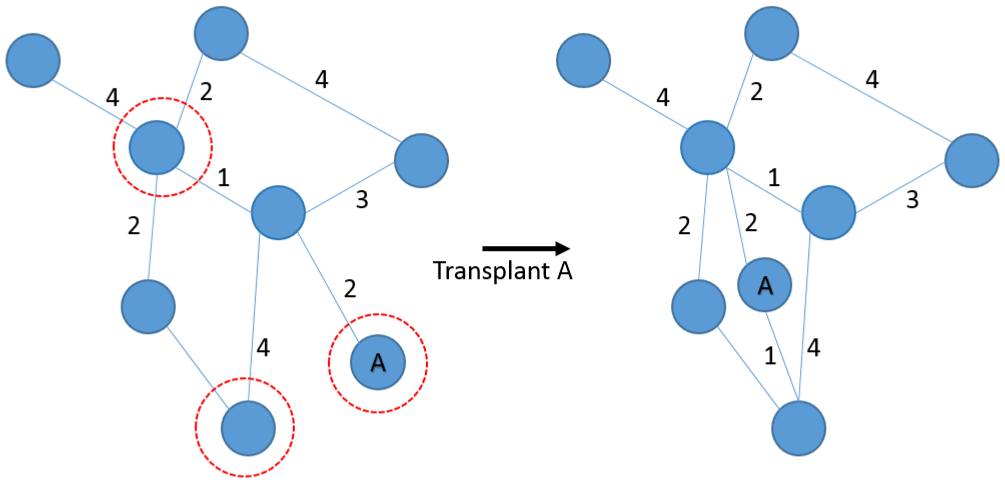


Figure 1: Transplanting of vertex A

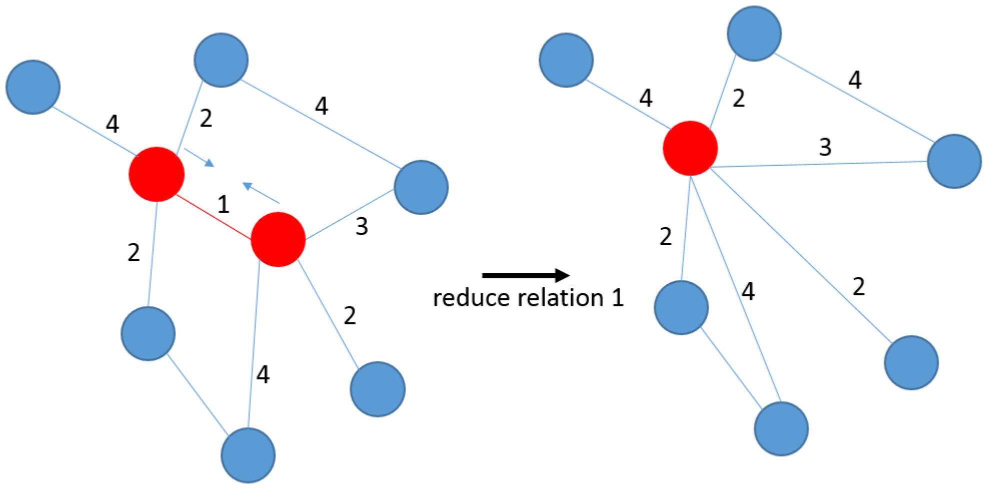


Figure 2: Reducing of edges of weight 1.

their information as well as their neighbors' information in pairs of vertex pointers and version numbers, much like what we did with transplanting. Whenever we find a new vertex-version pair to record, we push it into a set. Notice that here, using a set is crucial since it automatically rejects duplicates: it is possible when traversing the graph, one may record a certain vertex A and its neighbor B as needed, and later find out B in itself satisfies the search condition and should be pushed into the set as well as its neighbor A. Since the set only contains vertices that need to be locked but not vertices that need to be operated on, we use another task list to record those vertices where we need to invoke the *mergevertex* method. After traversing the graph, we go through the set to lock all the required vertices, if failed, meaning some vertices has been modified along the way and thus the search has to be redone. The pseudo-code for graph class methods are shown bellow.

```
void graph::insert(vertex* newv){ //insert a new vertex v
vertex* probe;
std::set<vertex*, int> myneighbor;
repeat{
     repeat{
          assign probe to first vertex;
          if this vertex is desired neighbor:
               myneighbor.insert(make_pair(probe, atomic_load(probe->vn)));
          assign probe to next vertex;
      } till graph is traversed
     for(iterator it = myneighbor.begin(); it !=myneighbor.end(); ++it):
         it->first->locking(it->second); //lock all required neighbors
} while failed to grab all the locks

for(iterator it = myneighbor.begin(); it !=myneighbor.end(); ++it:
     add an edge between newv and it->first;

for(iterator it = myneighbor.begin(); it !=myneighbor.end(); ++it):
     it->first->unlocking();
};


void graph::transplant(int i){ //transplant all vertices with key i
vertex* probe;
std::set<vertex*, int> myneighbor;
std::set<vertex*> todo;
repeat{
     repeat{
          assign probe to first vertex;
          if this vertex needs to be transplanted
               myneighbor.insert(make_pair(probe, atomic_load(probe->vn)));
```

```
                for all the vertices v in the adjacency list:
                      myneighbor.insert(make_pair(v, atomic_load(v->vn)));
                todo.insert(probe);
           assign probe to next vertex;
       } till graph is traversed

     for(iterator it = myneighbor.begin(); it !=myneighbor.end(); ++it):
         it->first->locking(it->second); //lock all required neighbors
} while failed to grab all the locks

for(iterator it = todo.begin(); it !=todo.end(); ++it:
     cut *it isolate;

for(iterator it = myneighbor.begin(); it !=myneighbor.end(); ++it):
     it->first->unlocking(); //free up old neighbors

for(iterator it = todo.begin(); it !=todo.end(); ++it):
     reinsert *it; //similar to addvertex routine
};


void graph::contractedge(int i){ //contract all edges with weight i
vertex* probe;
bool found;
std::set<vertex*, int> myneighbor;
std::set<vertex*> todo;
repeat{
     repeat{
          assign probe to first vertex;
          if this vertex needs to be transplanted
                myneighbor.insert(make_pair(probe, atomic_load(probe->vn)));
                for all the vertices v in the adjacency list:
                      myneighbor.insert(make_pair(v, atomic_load(v->vn)));
                todo.insert(probe);
           assign probe to next vertex;
       } till graph is traversed

     for(iterator it = myneighbor.begin(); it !=myneighbor.end(); ++it):
         it->first->locking(it->second); //lock all required neighbors
} while failed to grab all the locks

for(iterator it = todo.begin(); it !=todo.end(); ++it:
     found = true;
     repeat{
          find a neighbor v with edge i;
          if no such neighbor:
                found = false;
          else:
                 merge vertices *it and v

     } till found = false
}

for(iterator it = myneighbor.begin(); it !=myneighbor.end(); ++it):
     it->first->unlocking(); //free up old neighbors
};
```

3. Performance Results

Results of test on niagara1 machine: a Sun T1000 multiprocessor with 8 4-way multithreaded cores, are presented in Fig.3 and Fig.4. During the test, we first construct a graph with 5000 vertices. We then assign each threads (total number p) a fixed amount of tasks, for instance, perform 100 graph operations as defined in the class specification. We also compare our result with the case where no speculation is used: whenever a graph operation is performed, the entire graph is locked beforehand (shown in the figures by the "graph lock" curve). The performance is measured by assigning a given amount of task to each thread, and use a fine resolution timer to measure the time it takes for all tasks to run to completion. For example, the transplanting 100 vertices per thread on a graph with 5000 total number of vertices, is equivalent to moving 64% of the vertices. Each thread works on 100 vertices. As we reduce the number of threads to 8, each thread still works on 100 vertices, but the total amount of vertices changed is reduced by a factor of 4. We calculate the throughput as the ratio of number of graph manipulations done to the computation time. Because we keep the total number of vertices constant, the speculative searching time when multiple threads are running can be shortened (by roughly the same factor regardless of the number of threads) greatly compared to the "graph lock" method. However, because of the upper bound of the number of vertices, the more threads running (each thread always performs a fixed amount of task), the more likely threads will run into each other causing redos of the speculative search. Therefore intuitively, we should expect a initial growth in the throughput due to exploited parallelism by speculation, and a decrease in performance when the number of vertices affected becomes large. The $100 ops/thread$ case with 32 threads has more than 3200 vertices locked since in some operations, we do not only lock the vertices themselves but also their neighbors.
In Fig.3.(a), we see that the non-speculative algorithm does not give good parallel performance, with no boost to the throughput. The reason is that in most of the methods, locking are required almost from the start of the operation. Therefore most threads end up waiting for the current owner of the lock to finish its operation before they can continue. On the contrary, the speculative version gives nice performance boost for up to 6 processes. The majority of

the operations performed here are searching to locate correct neighbors. This operation is not guarded by the locks but rather by an atomic load operation to check the current version of each needed vertex. If this check fails, the search has to be done all over again since the originally found vertex might not be needed anymore. Note that suppose one thread tries to remove all vertices with key value 1 and another thread tries to insert a new vertex of value 1, the end result of the graph may or may not contain a vertex with value 1. This is because although modifications to the existing vertices is done by making sure they are not changed after searching, there is no guarantee that the search remains complete after grabbing the lock. This is no different from locking the entire graph for invoking insertion or removing operations since depending on the actual order of the two calls, the result can be either with or without the newly inserted vertex. For more than 8 processors, the parallel performance drops rapidly. There is no performance boost at $p = 24$, which is when about 50% of the vertices are operated on. Since each vertex by default takes three or more neighbors when inserted, this means that the majority of the vertices has to be locked by one of the threads for operation, leading to a good chance of conflicting.

In Fig.3.(b), we see that although the peak performance at 6 threads is slightly suppressed compared to the previous case, the speculative runs do scales better for more than 8 threads. This is because with the added operations of vertex removal, the graph itself is gradually shrinking in size over time. As stated before, the overhead incurred on the cases when a large number of vertices getting locked is that the potential risk of running into conflict (failed speculation): threads may record a vertex to lock but later find another thread has modified it (The possibility roughly scales as p^2). But with a smaller number of vertices due to removal, the redo of searching takes less time therefore reducing the overhead of failed speculation. The "graph lock" run as expected gives almost no speedup.

In Fig.4.(a), we study the situation when each thread carries out 100 transplants, 100 removals and 100 reductions (each thread only reduces a certain edge weight). Notice that in our experiments reduction can be redone since transplanting can introduce new edges with the required edge weight. We see that Fig.4.(a) shows much better scaling compared to the previous test, which is understandable since consecutive removal and reduction can lead to a graph with many disconnected components therefore greatly reduce the likelihood of speculation failure. Recall that failure occurs when one thread tries to speculatively lock a vertex and finds it has been modified by another thread. Usually, the locking is performed not on a target vertex (the vertex where the reduction or removal is performed on), but on the neighbor vertex of a target vertex. If the average number of neighbors is reduced in a graph, less conflicts will happen, leading to boosted performance. In Fig.4.(b), we see that this is the same case even without the removal operations. In Fig.4.(b), each thread performs 100 insertions, 100 replaces (replace all of certain key values by another key value) and 100 reductions. The replace operation does not need neighbors to be locked, leading to much less failure rate compared to transplanting or removal;

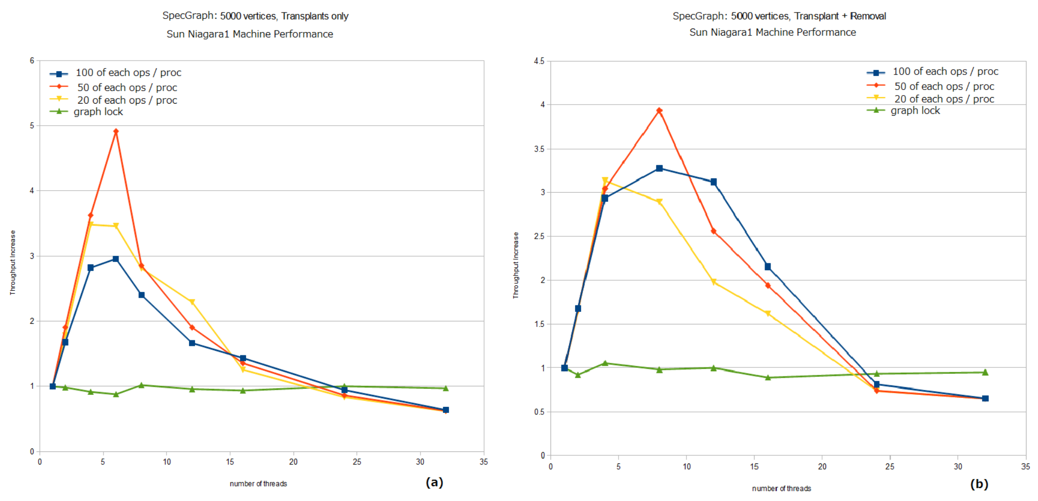
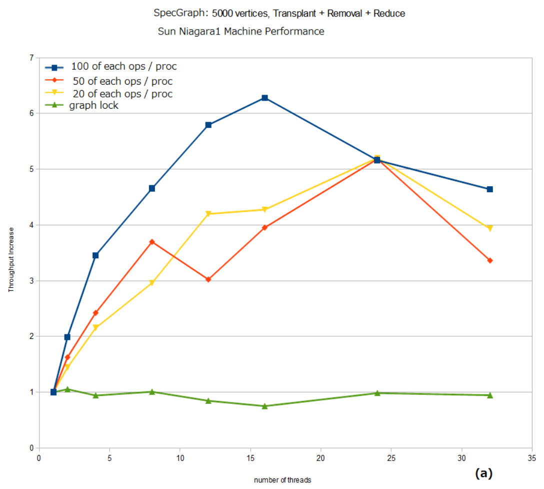


Figure 3: (a) Throughput increase on Sun T1000 Niagara1 machine for a speculative 5000-nodes graph to perform transplant operations: 100, 50 and 20 operations per thread; (b) Throughput increase for a speculative 5000-node graph to perform transplant and removal operations: 100, 50 and 20 of each operation per thread.

insertions only has to lock three or four neighbor vertices. Therefore the only operation that requires a large number of vertex locking is the reduction operation. The newly inserted vertices also increase the advantage of speculation by increasing the amount of computation time on searching. Expectedly, this case performs much better compared to the cases demonstrated in Fig.3. Also notice that in Fig.4.(b), the 20 operations per thread case (yellow) performs lower than expected. When the number of edges affected by reduction is low, the graph cannot shrink as fast as the other two cases in the same diagram, therefore resulting in longer search time as well as failed speculation overhead.

Next, we look at the performance on x86 cycle machines (maximum 8 hardware threads). Fig.5 shows the throughput comparison for a 5000-vertex and a 20000-vertex graph. One thing we notice is that when the graph is relatively small (e.g. 5000 vertices), the performance is no better than non-speculative execution. This is because the saved searching time by speculation does not justify the parallel overhead as well as failure penalty on this machine. One way to boost the performance is to increase the size of the searching phase: by either increasing the size of the graph or by letting each thread inserting a large number of vertices beforehand. We see that the 2000-vertices case does give better performance initially, but drop fastly for more than 4 threads. This is a similar behavior as observed in Fig.3.(a). When 6 threads each transplanting 1000 vertices, the total number of locked vertices can be close to 50% of that of the entire graph, causing greater chance of failed speculation.



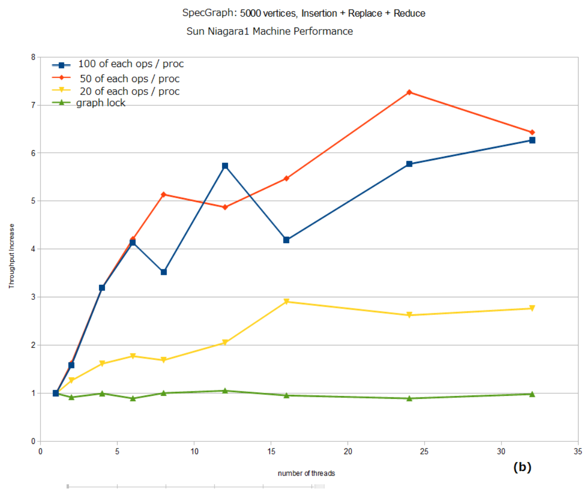


Figure 4: (a) Throughput increase on Sun T1000 Niagara1 machine for a speculative 5000-nodes graph to perform transplant, removal and reduction operations: 100, 50 and 20 of each operation per thread; (b) Throughput increase for a speculative 5000-node graph to perform insertion, replace and reduction: 100, 50 and 20 of each operation per thread.

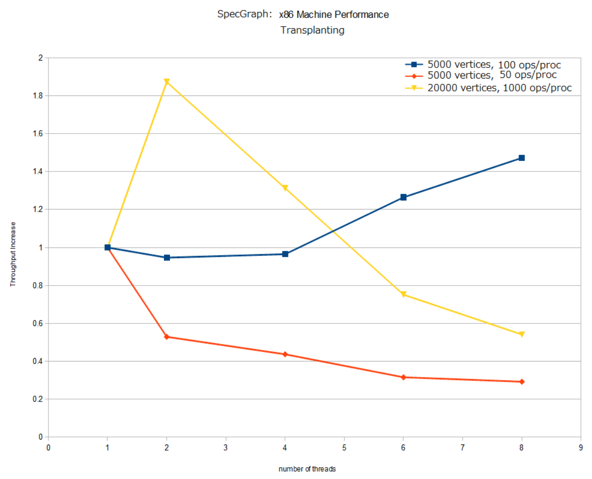


Figure 5: Throughput on x86 Cycle machine for a speculative graph, with 5000 or 20000 vertices.

4. Future Works

Considering the test results being inherently dependent on how operations are planned out, more tests are needed to benchmark the fine grained speculative approach. Instead of reducing certain relationships characterized by edge weights, graph reduction can have another type of "preparation" phase where structural features are searched to identify mergers. However, such search can still be performed speculatively since no modifications are done during the process. Intuitively, this approach can benefit from speculation even more since more resources are needed in the search phase. We plan to test more sophisticated applications using the described approach. The speculation can also be automated using Compiler Aided Speculation (CSpec) as described in Xiang2013. To compare the performance difference between such an approach with the current speculation implementation is a future interest.