Improving Hardware Transactional Memory Parallelization of Computational Geometry Algorithms Using Privatizing Transactions

Ricardo Quislant, Eladio Gutierrez, Emilio L. Zapata, Oscar Plata

Dept. Computer Architecture, University of Malaga, Spain

Abstract

Hardware transactional memory is a new parallel programming paradigm supported by current commercial multiprocessors. This paradigm provides optimistic concurrency and overcomes some of the problems associated with classical lock-based synchronization, such as deadlock and serialization. Certain algorithms of computational geometry are found to be good candidates for parallelization with this paradigm. However, hardware transactional approaches to these algorithms lead to poor performance as the resulting transactions are too large for the underlying hardware to deal with. Large transactions overflow hardware resources serializing the execution.

In this paper, we propose using privatizing transactions to parallelize two computational geometry algorithms: Lee's algorithm, which solves the shortest-route problem, and Ruppert's algorithm for Delaunay/Voronoi mesh refinement. Privatizing transactions are based on commercial hardware transactional memory extensions, and their goal is to reduce transaction footprint by means of a non-transactional private execution section. This results in effective smaller transactions. Our implementation is able to further reduce the transaction size as we propose a reduced validation set for privatizing transactions. Programming complexity of these implementations are discussed.

Results show that our privatizing transaction implementations indeed enhance performance comparing with existing hardware transactional memory versions. Experiments with Intel's transactional memory extensions yield speedups ranging from 2× to 3.5× with four threads.

Keywords: Computational geometry, Hardware transactional memory, Privatizing transactions, Lee's algorithm, Ruppert's algorithm, Delaunay triangulation

1. Introduction

In 1993, Herlihy and Moss proposed hardware transactional memory (HTM) [1, 2] with the intention of making parallel programming efficient and easy without the use of locks. They define a transaction as a finite sequence of machine instructions, executed by a single process, which satisfies the properties of serializability and atomicity. In HTM, it is the hardware which ensures the properties of a transaction by performing the following tasks: transactional updates are hidden from other transactions and made visible only on a successful commit, via transactional buffers/caches; conflicting accesses must be detected. There is a conflict when a transaction reads/writes transactionally modified data (the coherence protocol is commonly used for that purpose); conflicts must be resolved, usually by aborting one of the transactions involved in the conflict. Transactions are always executed in parallel except when a conflict is detected. In such a case, the conflicting transactions are serialized. HTM is therefore an optimistic concurrency paradigm in contrast to classical locking techniques, which always serializes the execution. The inceptive approach of Herlihy and Moss was used to protect short critical sections with short transactions, thus eliding the memory accesses to the lock and preventing the problems that come along with fine-grain locks: deadlock, convoying and priority inversion.

Email addresses: `quislant@uma.es` (Ricardo Quislant), `eladio@uma.es` (Eladio Gutierrez), `zapata@uma.es` (Emilio L. Zapata), `oplata@uma.es` (Oscar Plata)

Subsequent research has focused on providing a simplified programming model where transactions, more coarse- than fine-grained, were occasionally inserted into the code. With TCC [3] the programmer defined large transactions without worrying about their dependencies, which the underlying system was in charge of managing. The programmer then fine tuned the transactions with the feedback of the system. A problem arises when supporting coarse-grain transactions, since the hardware poses limitations both on their size (transactional buffers are finite) and duration (OS events, interrupts, etc. may abort transactions). Large transactions, such as coarse grain locks, usually imply low programming complexity, but overflow transactional resources serializing the execution. In [4, 5, 6, 7, 8] the virtualization of the HTM system is addressed so that those limitations are hidden from the user to simplify the programming model. However, virtualization often increases hardware complexity to such an extent that is believed to be better handled in software [9].

In 2012, IBM [10, 11] and Intel [12] released multiprocessors with HTM extensions. The transactions supported by these systems are limited in size and duration by hardware constraints. Support for virtualization is not provided by these processors and the user has to be aware of the hardware limitations. In fact, the user must implement a fallback handler to deal with those transactions which are unable to commit. Large and long, or conflicting transactions may lead to livelock, where the program is stuck executing the same transaction repeatedly. The transaction is aborted and retried without coming to an end. To prevent this situation, the fallback code is executed after a number of transaction retries. The fallback often executes the same code of the transaction enclosed by a global lock. Other transactions subscribe to the lock by reading it so that they are aborted when one transaction acquires the lock. Thus, transactional and non-transactional fallback codes are not allowed in parallel.

1.1. Motivation

Two different algorithms in the context of computational geometry are implemented in this paper: Lee's algorithm, which solves the shortest-route problem, and Ruppert's algorithm for Delaunay/Voronoi mesh refinement. While the first one can be used in logical drawing, optimal route finding and wiring diagramming, the second one is a well-known domain discretization algorithm used in numerical simulations, solid modeling, graphics and, also, in Biology for modeling multi-cellular systems and tissue [13].

These kind of algorithms are good candidates to be parallelized with HTM as they pose several problems when using lock-based techniques. Using a coarse-grain lock to protect the access to the global shared data structure serializes the execution, while protecting each of the elements of the data structure with fine-grain locks can lead to deadlock scenarios difficult to prevent. In fact, parallel versions of Lee's and Ruppert's algorithms are usually approached by partitioning the shared data structure and assigning each partition to a core (see Section 5). Eventually, these solutions have to deal with conflicts at the border of the partitions, which complicates the algorithm implementation.

Using HTM extensions, a transaction is assigned the work to find a route between two points in Lee's algorithm, or to refine a triangle in Ruppert's algorithm. Transactions are executed in parallel, working in the global shared data structure, and they are only serialized if they work with conflicting data: i.e. two paths that collide or two triangles very close to each other. The underlying HTM system is in charge of detecting and resolving these conflicts and the programmer is relieved from this duty.

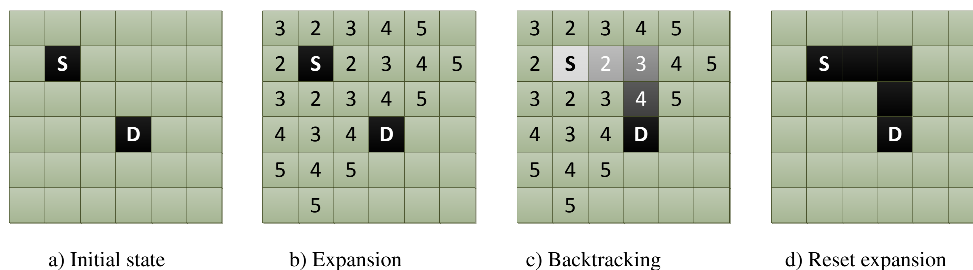
However, HTM approaches to these algorithms lead to poor performance as transactions tend to be too large for the underlying hardware to deal with. Therefore, we propose to improve the HTM performance of these algorithms, which should be examples of HTM-friendly algorithms.

1.2. Contributions

In this paper, we use privatizing transactions [14], based on commercial HTM extensions, as a tool to program coarse-grain transactions that result in effective small ones. The bulk of the transactional work is executed in private non-transactionally. Then the transactional system is used to validate all the reads done in the privatizing/execution section, and to commit the changes. Current parallel HTM implementations of Lee's and Ruppert's algorithms exhibit large transactions that yield sequential performance with commercial HTM extensions [15, 16].

The contributions of this paper are the following:

- We provide a new HTM implementation of Lee's and Ruppert's algorithms using privatizing transactions in HTM.



a) Initial state b) Expansion c) Backtracking d) Reset expansion

Figure 1: Lee's algorithm phases. 6×6 grid. S: Source, D: Destination.

- We propose reducing the validation set of privatizing transactions by finding a small subset of variables which are representative of the whole set of variables read in the private section. We aim at reducing the transactional section of privatizing transactions as much as possible to meet the hardware constraints of the HTM system.
- We discuss the programming complexity of both solutions and how the data structures involved and the validation set reduction technique may affect the complexity of the solution.

We have evaluated our proposals using the Intel Core architecture with the restricted transactional memory (RTM) extensions. Experimental results show speedups from $2\times$ to $3.5\times$ with four threads, over both the current HTM versions and the sequential.

The remainder of this paper is organized as follows. Section 2 describes the current HTM implementations of Lee's and Ruppert's algorithms. Section 3 introduces the privatizing transaction API, describes our Lee and Ruppert privatizing transaction implementation proposals, explains the validation set reduction tecnique and discusses the programming complexity. Section 4 presents the experimental methodology and discusses the results. Section 5 outlines the related work. Finally, we draw the conclusions in Section 6.

2. Background

2.1. Lee's Algorithm

C. Y. Lee proposed an algorithm for path connections [17] to solve problems of logical drawing, wiring diagramming and optimal route finding. This algorithm finds the shortest path between two points, when possible, and in the presence of obstacles such as edges, other previously calculated paths, and other constraints.

Figure 1 shows an example with the four phases of Lee's algorithm:

1. *Initial state*. One pair of source-destination coordinates is picked from a list of pairs and placed in the grid.
2. *Expansion*. The algorithm performs an expansion from the source to the destination by storing in each cell the number of steps (weight) needed to reach it.
3. *Backtracking*. If the expansion reached the destination point, the algorithm backtracks from destination to source by choosing the lowest weighted neighbor at each cell. A bending cost can be added to the weight of a cell whether taking the cell implies changing the direction of the path. If after adding the bending cost several neighboring cells have the same weight, one of them is taken indistinctly. The bending cost can be important for certain applications where a minimum number of bends is needed, e.g. bends in printed circuit board tracks may cause attenuation of certain signal frequencies.
4. *Reset expansion*. The expansion is reset by eliminating the weights from the grid. The calculated path is left marked as an obstacle.

2.1.1. Vanilla-HTM Implementation

A parallel TM implementation of this algorithm can be found in [18] by Watson et al. Actually, they propose three different parallel TM-based versions of Lee's algorithm: i) A simple version that encloses all the phases of the algorithm in one transaction that works with the global grid. This version yields poor performance as overlapping

```
 1  void threadHTMLee(gridType *globalGrid, pairType *pairList, int bendCost) {
 2    gridType *localGrid;
 3    pathType *path;
 4    while(TRUE) {  //While there is a pair
 5      pairType* pair;
 6      xbegin();
 7        pair = pickSrcDstPair(pairList);
 8      xend();
 9      if(!pair) break;  //Terminate thread work
10      xbegin();
11        if(OPEN_TRANSACT) { //If HTM supports open transactions
12          xbeginOpen();
13            gridCopyFromTo(globalGrid, localGrid);  //globalGrid read only
14          xendOpen();
15        } else if(EARLY_RELEASE) { //If HTM supports early release
16          gridCopyFromTo(globalGrid, localGrid);  //globalGrid read only
17          earlyRelease(globalGrid);
18        } else gridCopyFromTo(globalGrid, localGrid); //globalGrid read only
19        if(expansion(pair, localGrid)) {
20          path = backtracking(pair, localGrid, bendCost);
21          for(int i=1; i < length(path); i++) { //Copy path to global grid
22            if(globalGrid[path[i]] != GRID_POINT_EMPTY) //Validation
23              xabort();
24            globalGrid[path[i]] = GRID_POINT_FULL;
25          }
26        }
27      xend();
28    }
29  }
```

Figure 2: HTM implementation of Lee's algorithm as suggested by Watson et al.

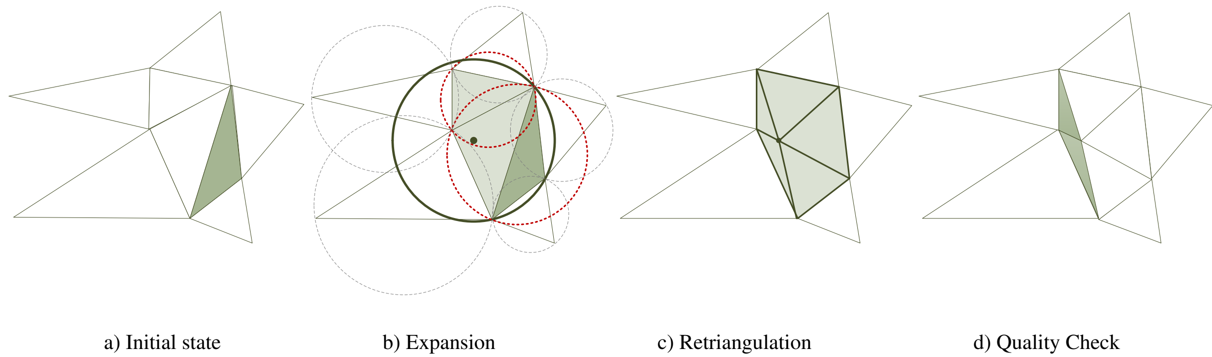
expansions abort transactions; ii) The transaction reads from the global grid and expands on a private one. Conflicts are detected when writing the path back to the global grid. If the expansion is large the probability of abort is large as well; iii) The transaction reads the global grid and expands on a private one, but the read set is cleared after the expansion. The cells belonging to the calculated path are validated before commit so that the transaction is aborted when one cell was written by other transaction. This version outperforms the others.

However, Watson et al. do not address the implementation of their proposals using vanilla HTM constructs, as they seek for an implementation independent evaluation. They only suggest that the third version of the algorithm could be implemented with the use of early release [19] or open nested transactions [20]. In fact, they provide a benchmark using early release in the Lee-TM benchmark suite [21]. The same approach is followed by the STAMP's Labyrinth benchmark [22].

Figure 2 shows the thread code of a parallel HTM implementation of Lee's algorithm following the suggestions from Watson et al. Lines 11 to 18 perform the copy of the global grid to the local grid. If the HTM system supports open transactions, lines 11 to 15, the read and write sets will be cleared when the open transaction is successfully committed. If the system supports early release, lines 15 to 18, the read set will be cleared in line 17. If neither of these features is supported, the common case in commercial HTMs (see last paragraph), the read set will not be cleared and the algorithm implemented is the second version proposed by Watson et al, that we have called Vanilla-HTM Lee. In any case, the copy of the grid is always protected by the transaction, and a concurrent write of a path from other transaction aborts the copy in order to start from an updated grid.[1]

Next, the expansion is performed in the local grid, line 19, and only if the destination was reached, the backtracking phase is executed in the local grid too, line 20. The resulting path is written to the global grid in lines 21 to 25. Before proceeding to write the path a validation is performed. All the grid cells comprising the path are checked for being empty, line 22. The transaction is aborted if not, line 23, as another transaction may have committed a path between the `earlyRelease` or `xendOpen` instructions and the copy of our path.

[1] In the figures, `xbegin`, `xend`,... are abstractions for the low-level transactional primitives, which include a fallback mechanism.



a) Initial state b) Expansion c) Retriangulation d) Quality Check

Figure 3: Rupert's algorithm phases.

Early release and open transactions are not features supported by commercial HTM systems. Consequently, the versions of the algorithm provided by Lee-TM and STAMP making use of the early release feature can be executed only in software TM (STM) systems such as DSTM2 [23] or TinySTM [24], as well as in HTM simulators. This fact makes the Vanilla-HTM implementation of the algorithm serialize the execution in commercial HTM systems [15, 16].

2.2. *Ruppert's Algorithm*

J. Ruppert proposed an algorithm for creating Delaunay triangulations [25] of geometric objects to generate quality mesh approximations to such objects. Triangulation is for use in computational geometry, graphics, solid modeling, numerical simulation, and other fields, where many applications require a quality mesh representation of an object to yield reliable results. It has been also applied in Biology for modeling multi-cellular systems and tissue [13]. The algorithm refines triangles in a mesh by eliminating those with very acute angles, also known as skinny or sliver triangles, that can lead to unreliable behavior of the applications.

The Delaunay triangulation property is enforced by the algorithm, where each triangle in the mesh fulfills that its circumcircle does not contain any point other than the circumscribed ones. Furthermore, Ruppert's algorithm checks that all triangles meet the minimum angle constraint given as parameter. Any bad triangle not complying with these quality constraints is refined by generating new triangles, which are also checked for quality. This procedure is iterative and will end when there are not bad triangles to refine.

The main phases of Ruppert's algorithm are depicted in Figure 3 and are the following:

1. *Initial State*. One triangle is taken from a queue of bad triangles. A mesh with non-quality triangles is supposed as input, and the algorithm initially traverses the mesh in the search of bad triangles to fill up the queue.
2. *Expansion*. The center of the circumference that circumscribes the triangle is taken as a new point in the mesh. The algorithm then calculates the circumcircle of the three neighboring triangles and includes them in the expansion if the new point is within their circumcircles. The same operation is performed with the neighbors of the new included triangles until no neighbor has the new point in its circumcircle.
3. *Retriangulation*. The algorithm removes the old triangles marked in the expansion and creates new ones. The new triangles are constructed using the new point calculated in the expansion phase, along with each one of the segments delimiting the expansion.
4. *Quality Check*. The algorithm checks that the new triangles comply with the quality constraints. If not, they are marked as bad and inserted into the queue of bad elements to be refined subsequently.

There is a corner case called *encroachment* that happens when the circumcircle of a boundary segment of the mesh contains the new point. In this case the algorithm must divide the segment by its midpoint and refine it. The same procedure as to refine a triangle is followed. Finally, the bad triangle, which we had to refine in the first place, is processed again.

```
1  typedef struct region {
2    coordinateType circumCenter;
3    listType *expansionList;
4    listType *borderList;
5    listType *badList;
6  } regionType;
7  void threadHTMRuppert(meshType *globalMesh, queueType *workQueue) {
8    while(TRUE) {  //While there is a bad triangle
9      triangleType* triangle;
10     xbegin();
11       triangle = queuePop(workQueue);
12     xend();
13     if(!triangle) break;  //Terminate thread work
14     regionType region;
15     xbegin();
16       regionRefine(&region, triangle, globalMesh);
17     xend();
18     xbegin();
19       regionTransferBad(&region, workQueue);
20     xend();
21   }
22 }
23 void regionRefine(regionType *region, triangleType* triangle, meshType *globalMesh) {
24   triangleType *encroach;
25   while(!triangleIsGarbage(triangle)) {
26     clearRegion(region);
27     encroach = expansion(triangle, region);
28     if(encroach) regionRefine(region, encroach, mesh);
29     else break;
30   }
31   if(!triangleIsGarbage(triangle))
32     retriangulation(region, triangle, mesh);
33 }
```

Figure 4: Vanilla-HTM Ruppert's algorithm implementation (i).

2.2.1. Vanilla-HTM Implementation

A TM parallelization of Rupert's algorithm can be found in [26], and STAMP's Yada [22] is designed upon that work. Figure 4 shows the code of the implementation. Each thread receives a pointer to the mesh and a pointer to the work queue, line 7. The global mesh is implemented as a graph of triangles. The `meshType` is a structure which holds a pointer to the root triangle, and each triangle holds a list with its three neighbors. It is, therefore, a connected graph with a path between every pair of triangles. The structure also holds a set containing the segments that form the boundary. The work queue holds all the triangles of the mesh that do not comply with the quality constraints. Lines 10 to 12 pick a bad triangle from the global work queue with the protection of a transaction. If there are not bad triangles to refine, line 13 breaks the loop and ends thread execution.

Next, line 14 defines a region to keep track of the information to perform the expansion. The `regionType`, lines 1 to 6, is a structure holding the coordinates of the circumcenter of the triangle to refine, a list of pointers to the triangles in the expansion and a list of pointers to the segments delimiting the expansion region, the `borderList`. The structure also holds a list of pointers to the new bad triangles produced in the retriangulation. Although we have put the quality check as a phase separated from the retriangulation, the quality check can be performed at the same time of creation of the new triangles. Thus, the retriangulation phase fills up the vector of bad triangles of the region for later inclusion into the work queue. This is done in lines 18 to 20 by a different transaction to avoid unnecessary aborts of the transaction that refines the triangle.

Lines 15 to 17 execute the transaction that refines the triangle. It is a function that includes the two main phases of the algorithm: expansion and retriangulation. The function, in line 23, receives the region, the bad triangle, and the global mesh. It is a recursive function to tackle the encroachment. If the expansion function, in line 27, finds a mesh boundary segment whose circumcircle contains the new point, the function returns a pointer to that segment which is stored in the `encroach` variable. The `triangleType` deals with triangles but also with segments by setting the

number of edges to one and the number of points to two. Then, `regionRefine` is executed with the segment as a parameter, and the expansion is performed on a reset region, line 26. If there is not another encroachment, the loop in line 25 is broken by line 29, and if the triangle (in this case a segment) has not been deleted by other transaction, i.e. the triangle is not garbage (line 31), the region is retriangulated in line 32. After that, the function resumes in line 25, and the loop is executed if the bad triangle to refine in the first place is not garbage. The region is cleared (line 26) and a new region is expanded and retriangulated.

The code for the expansion and retriangulation functions is in Figure 5. The expansion starts off by defining the `isBoundary` variable to perform differently depending on whether the expansion is on a bad triangle or a boundary segment. The core of the function is the loop in line 5. A queue is declared for the expansion, line 3, and the bad triangle is first inserted. The loop ends when there are not more elements to pop. In each iteration of the loop, the list of neighbors of the triangle popped out from the queue is traversed by the inner loop in line 9. If the circumcenter of the bad triangle is in the circumcircle of the neighbor, line 13, the neighbor is inserted in the queue, line 15. If the neighbor is a segment and the bad triangle is not a boundary segment we are in an encroachment scenario and the function returns the segment, line 14. In case the neighbor does not belong to the expansion the common segment is inserted into the border list for the retriangulation phase, line 17.

The retriangulation function in line 23 performs three tasks. First, it removes the old triangles found in the expansion, lines 24 to 27. Second, it splits in half the segment in case we are dealing with an encroachment, i.e. the expansion was on a boundary segment, lines 28 to 32. And finally, it inserts the new triangles built off the circumcenter and the borders of the region. As previously mentioned, the new triangles are checked for quality and inserted into the bad list of the region if they do not meet the constraints, lines 33 to 39.

2.2.2. Dynamic memory allocation and the garbage field

The algorithm works with dynamic data structures such as linked lists, graphs and queues, which are updated by transactions. That implies using dynamic memory allocation inside transactions which may lead to aborts. For instance, libc's malloc may change the size of the data segment with the system call `brk`, and either system calls or instructions that update segment registers may always cause transactional aborts, as stated in [27]. Furthermore, the multithreading support of malloc uses per-thread arenas that are not private to the thread, and locks are used to access the arenas in mutual exclusion [28]. These locks may also cause aborts when found inside transactions.

Consequently, the implementation of the algorithm uses malloc to allocate a large pool of memory in the initialization of the application out of transaction, and provides a specific function to get memory from that pool. Freeing memory from the pool is not implemented and it is left to the end of the application, where the whole pool is freed out of transaction.

As far as the garbage field is concerned, each triangle has a boolean field to mark whether it is garbage or not. A triangle is removed from the mesh in the retriangulation phase by removing its pointer from the list of neighbors of its neighbors. Figure 6 shows a situation in which removing a triangle is not detected by other thread unless we use the `isGarbage` field. Thread 1 picks a pointer to triangle A from the work queue. Then, thread 2 picks triangle B, whose expansion reaches triangle A and subsequently removes it in the retriangulation phase. Suppose that Thread 1 has not yet begun the transaction to refine triangle A because it happened to be descheduled. After thread 1 is rescheduled, it founds that triangle A is garbage and does nothing. In case the garbage field is not used, the thread would work with a deleted triangle thus wasting time. Or even worse, if the triangle had been freed, thread 1 would have dereferenced an invalid pointer with the risk of a segmentation fault.

3. Implementation Using Privatizing Transactions

Commercial HTM systems do not properly deal with the algorithms studied in this work. The main reason is that transactional data sets are constrained to the size of buffers or L1/L2 caches, and the HTM implementations of Lee and Rupert algorithms exhibit large data set transactions which often overflow the hardware resources, thus yielding a performance very close to that of the sequential versions [15, 29].

In this section we use privatizing transactions proposed in [14] to implement privatizing HTM versions of the algorithms. Our purpose is to shrink the size of the transactions to fit the HTM resources. Moreover, we discuss the different programming complexities that arise depending on the application.

```
triangleType *expansion(triangleType *triangle, regionType *region) {
  bool isBoundary = isSegment(triangle);
  queueType *expandQueue;
  queuePush(expandQueue, triangle);
  while(!isQueueEmpty(expandQueue)) {  //While there is a triangle to expand
    triangleType *current = queuePop(expandQueue);
    listInsert(region->expansionList, current);
    list *neighborList = getNeighbors(current);
    while(listHasNext(neighborList)) { //While there is a neighbor
      triangleType *neighbor = listNext(neighborList);
      isGarbage(neighbor); //To detect conflicts
      if(!listFind(region->expansionList, neighbor) {
        if(inCircumcircle(neighbor, region->circumCenter)) {
          if(!isBoundary && isSegment(neighbor)) return neighbor; //Encroachment
          else queuePush(expandQueue, neighbor);
        } else //Triangle borders region, save for retriangulation
          listInsert(region->borderList, getCommonEdge(neighbor, current));
      }
    }
  }
  return NULL;
}
void retriangulation(regionType *region, triangleType* triangle, meshType *globalMesh) {
  while(listHasNext(region->expansionList)) { //Remove old triangles
    triangleType *oldTriangle = listNext(region->expansionList);
    meshRemove(globalMesh, oldTriangle);
  }
  if(isSegment(triangle)) { //If encroachment, split in half
    meshInsert(globalMesh, split(triangle));
    meshRemoveBoundary(globalMesh, triangle);
    meshInsertBoundary(globalMesh, split(triangle));
  }
  while(listHasNext(region->borderList)) { //Insert the new triangles
    edgeType *edge = listNext(region->borderList);
    triangleType *newTriangle = newTriangle(edge, region->circumCenter);
    meshInsert(globalMesh, newTriangle);
    if(isBadTriangle(newTriangle)) listInsert(region->badList, newTriangle);
  }
}
```

Figure 5: Vanilla-HTM Ruppert's algorithm implementation (ii).

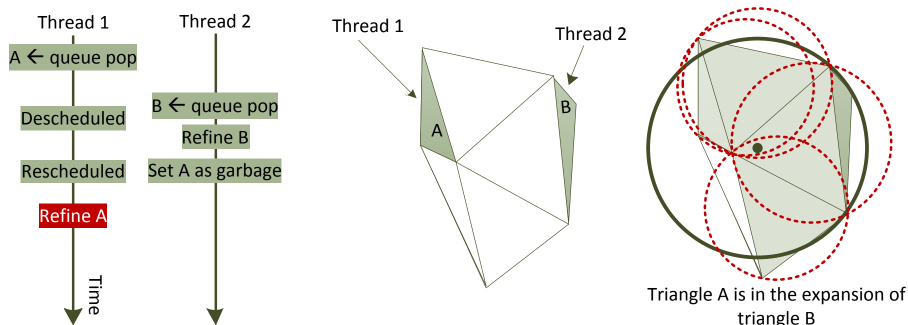


Figure 6: Garbage triangle scenario.

```
#define XBEGIN_PRIV()                                                          \
{                                                                              \
  int retries = 0, fromBegin = 1;                                              \
begin:                                                                         \
  if (retries > globalRetries) {                                               \
    while(!acquireLock(&globalLock)) ;                                         \
  }
#define XVALIDATE_PRIV()                                                       \
  if (retries <= globalRetries ) {                                             \
validate:                                                                      \
    if(!fromBegin) {                                                           \
      retries++;                                                               \
      if(VALIDATION_ERROR == xabortCause() || (retries > globalRetries)) {     \
        fromBegin = 1;                                                         \
        goto begin;                                                            \
      }                                                                        \
    }                                                                          \
    fromBegin = 0;                                                             \
    while (globalLock); /*Avoid Lemming effect*/                               \
    XBEGIN(validate);
#define XUPDATE_PRIV(isValid)                                                  \
    if (!isValid) XABORT(VALIDATION_ERROR);                                    \
  }
#define XEND_PRIV()                                                            \
  if (retries <= globalRetries) {                                              \
    if (globalLock) XABORT(); /*Lazy subscription*/                            \
    XEND();                                                                    \
  } else {                                                                     \
    releaseLock(&globalLock);                                                  \
  }                                                                            \
}
```

Figure 7: Priv-HTM API.

3.1. Privatizing transactions API

Figure 7 shows the application programming interface for the privatizing transactions (Priv-HTM API). Four statements define a privatizing transaction which divide it in three sections with the following semantics:

1. *Privatizing and execution section*: Delimited by macros `XBEGIN_PRIV()` (line 1) and `XVALIDATE_PRIV()` (line 8). In this section the user is meant to perform only reads to global memory. The programmer should privatize the data required by the algorithm and perform the execution phase of it. This section is executed non-transactionally.
2. *Validation section*: The programmer must validate the global reads performed in the previous section to ensure they have not changed. The code in this section is executed transactionally, and is enclosed by macros `XVALIDATE_PRIV()` and `XUPDATE_PRIV(isvalid)` (line 21). The result of the validation, whether true or false, has to be passed to the `XUPDATE_PRIV(isvalid)` statement as an argument. This section is not executed if the fallback is taken (line 9 shows the if clause whose scope is closed in line 23). `XBEGIN` is the low-level primitive to start a transaction. Its argument is the label from which the execution is resumed on abort.
3. *Update section*: The code to commit the results obtained in the privatizing and execution section must be placed between macros `XUPDATE_PRIV(isvalid)` and `XEND_PRIV()` (line 24). The update section is protected by the same transaction of the validation section, and it is executed only if the validation was successful. If not, the transaction is aborted (line 22). `XABORT` is the low-level primitive to explicitly abort a transaction. Its argument is an error code. `XEND` is the primitive to close a transaction.

The API makes a difference between an abort due to validation error and other aborts. If the abort cause was a validation error (line 13) or the transaction reached the retry limit, it is executed from the beginning (line 15). With this API the user has not to deal with explicit aborts.

```
 1 void threadPrivHTMLee(gridType *globalGrid, pairType *pairList, int bendCost) {
 2   gridType *localGrid;
 3   pathType *path;
 4   while(TRUE) {  //While there is a pair
 5     pairType* pair;
 6     xbegin();
 7       pair = pickSrcDstPair(pairList);
 8     xend();
 9     if(!pair) break;  //Terminate thread work
10     XBEGIN_PRIV();
11       path = NULL; //Reset variables
12       gridCopyFromTo(globalGrid, localGrid);  //globalGrid read only
13       if(expansion(pair, localGrid))
14         path = backtracking(pair, localGrid, bendCost);
15     XVALIDATE_PRIV();
16       bool isValid = TRUE;
17       if(path)
18         for(int i=1; i < length(path); i++)
19           if(globalGrid[path[i]] != GRID_POINT_EMPTY) {
20             isValid = FALSE;  //One path cell is not empty
21             break;            //Terminate validation
22           }
23     XUPDATE_PRIV(isValid);
24       if(path)
25         for(int i=1; i < length(path); i++)
26           globalGrid[path[i]] = GRID_POINT_FULL;
27     XEND_PRIV();
28   }
29 }
```

Figure 8: Priv-HTM Lee's algorithm implementation.

A fallback path with lazy global lock subscription is implemented (see Section 5). The transaction executes the fallback after a number of aborts (`globalRetries` in lines 5, 9, 13 and 25). Line 19 shows a busy wait before executing the transaction to avoid the Lemming effect [30]. The lock subscription is performed in line 26. We are using a single global lock in the figure for the sake of simplicity, but a ticket lock is more convenient as stated in [31].

3.2. Lee's Algorithm with Privatizing Transactions

Figure 8 shows Lee's algorithm using privatizing transactions. The standard transaction constructs are used to get the pair from the global list of pairs (lines 6 to 8) and the privatizing API is used for the main transaction (lines 10 to 27).

The main transaction starts off by resetting the variable `path` in line 11. This variable is used in other sections of the transaction to know if the path was found, other than to store the path. It has to be initialized at the beginning, as the first section of the privatizing transaction is non-transactional and writes are not undone on abort. Next, the majority of the work is performed: the grid privatization (line 12), the expansion (line 13) and the backtracking (line 14). If the path is successfully traced, the `path` variable will not be null, holding a pointer to a vector of grid cell indices.

Whereas in Figure 2 validation and path copy is performed by the same loop, with the Priv-HTM API those steps are performed separately. Lines 15 to 23 perform the validation, where we define the variable `isValid` in line 16. If every point in the path is empty in the global grid then the path is valid. The result is passed as a parameter to `UPDATE_PRIV()`, which is in charge of explicitly aborting the transaction or not. Eventually, lines 23 to 27 update the global grid with the path.

3.3. Ruppert's Algorithm with Privatizing Transactions

Using privatizing transactions in Ruppert's algorithm requires more changes than in Lee's algorithm. Our approach is based on performing the expansion almost the same way as it is, since it only reads the global mesh. But we perform the retriangulation in a private region, not the mesh, as shown in Figure 9. Subsequently, a new mesh update phase is

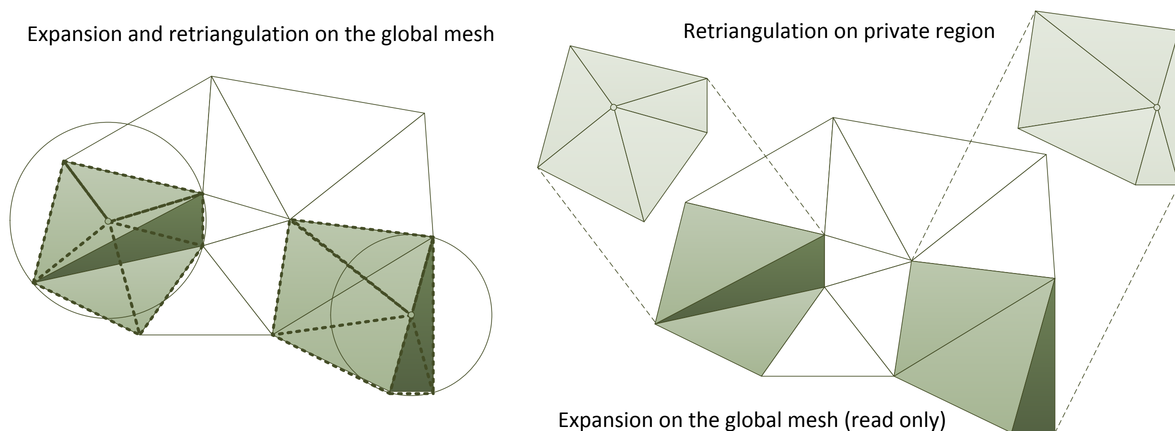


Figure 9: Vanilla-HTM Ruppert's algorithm implementation (left) versus Priv-HTM implementation (right).

executed in the `UPDATE_PRIV` section. The mesh is updated by inserting the new triangles in the private region as neighbors of the mesh triangles that border the region.

In Figure 10 we can see the changes made to `regionType`. We need a list to hold the new triangles resulting from the retriangulation, along with the pointers to their neighbors in the mesh, so that we can perform the mesh update quickly. We also need a vector to store the two new edges in case we found an encroachment. The thread function in line 6 now includes a privatizing transaction to refine the triangle. The privatizing and execution section, lines 13 to 15, executes the `regionRefinePriv` function which now returns a boolean telling whether it managed to refine the triangle. This section is non-transactional and only performs global reads. A private region is used for writes. The validation section, lines 15 to 18, transactionally executes the `regionValidation` function if the region was successfully refined. The update section, lines 18 to 20, transactionally links the neighboring triangles in the mesh with the new triangles in the private region by means of the `newList`.

The function `regionRefinePriv` is changed so that it is not recursive anymore. A recursive function to deal with the encroachment would not be effective now, as we separate the execution phase from the update one, which lays outside in line 19. The loop is now an if clause, line 29, that checks whether the triangle is garbage. The expansion function, line 31, can return two things: an `encroach` segment and if a neighbor in the expansion is garbage, `neighborIsGarbage`. In the first case, we add to the bad list both the segment and the bad triangle we had to refine in the first place, lines 32 to 35. The order is important as the segment resulting in the encroachment must be refined first for the triangle not to find the encroachment again. When a neighbor in the expansion is garbage, the triangle is added to the bad list for posterior refinement, lines 36 to 38. In both cases, the `regionRefinePriv` function returns false so that the validation and update phases are not executed. The `regionTransferBad` function in line 22 will insert these triangles into the work queue for their subsequent refinement. The retriangulation is performed in private if the expansion was successful, and the function returns true to signal a valid refinement, lines 41 to 42.

Figure 11 shows the code for the expansion and retriangulation functions. The `expansionPriv` function is almost the same as the `expansion` function of the HTM algorithm, but now it is not transactionally executed. Therefore, executing `isGarbage(neighbor)` such as in line 11, Figure 5, is not enough to detect conflicts and we have added lines 14 to 17 to detect that a neighbor is garbage. Also lines 13 and 27 are needed to control certain cases of transaction interleaving, discussed in next section. The `retriangulationPriv` function entails more changes since we have to build the new region in private. In case of an encroachment, the segment is split in half and stored in the `newEdge` vector of `regionType`, lines 32 to 35. Next, the new triangles are created and checked for quality such as the Vanilla-HTM algorithm does, but the triangles are inserted into the region instead of into the global mesh, line 39. Inserting a triangle into the private region consists in filling the neighbor list of each triangle with their neighbors in the private region. If a triangle has a neighbor in the global mesh, which is not in the expansion region but in the border, a pair of pointers to triangles (new, border) is created and inserted in the `newList` (line 3,

```
 1 typedef struct region {
 2   ...
 3   listType *newList; //List of pairs (new, border) triangles
 4   edgeType *newEdge[2];
 5 } regionType;
 6 void threadPrivHTMRupert(meshType *globalMesh, queueType *workQueue) {
 7   while(TRUE) {  //While there is a bad triangle
 8     xbegin();
 9       triangleType* triangle = queuePop(workQueue);
10     xend();
11     if(!triangle) break;  //Terminate thread work
12     regionType region;
13     XBEGIN_PRIV();
14       bool refined = regionRefinePriv(&region, triangle);
15     XVALIDATE_PRIV();
16       bool isValid = TRUE;
17       if(refined) isValid = regionValidation(&region);
18     XUPDATE_PRIV(isValid);
19       if(refined) meshUpdate(triangle, &region, globalMesh);
20     XEND_PRIV();
21     xbegin();
22       regionTransferBad(&region, workQueue);
23     xend();
24   }
25 }
26 bool regionRefinePriv(regionType *region, triangleType* triangle) {
27   triangleType *encroach;
28   bool neighborIsGarbage;
29   if(!isGarbage(triangle)) {
30     clearRegion(region);
31     encroach = expansionPriv(triangle, region, &neighborIsGarbage);
32     if(encroach) {
33       addToBadList(region, encroach); //Insert the segment first
34       addToBadList(region, triangle);
35       return FALSE;
36     } else if(neighborIsGarbage) {
37       addToBadList(region, triangle);
38       return FALSE;
39     }
40   }
41   retriangulationPriv(region, triangle)
42   return TRUE;
43 }
```

Figure 10: Priv-HTM Ruppert's algorithm implementation (i).

Figure 10) for subsequent use in the update section of the privatizing transaction.

Finally, Figure 12 shows the implementation of the validation and update functions. Both functions are executed transactionally. The `regionValidation` comprises an `isGarbage` test for all the triangles in the global mesh which are in the border of the expansion region, lines 3 to 6. All the old triangles in the expansion list are checked as well, lines 8 to 11. If at least one triangle is garbage we have been working on stale data, so the function returns false and the transaction is aborted and retried. Function `meshUpdate` removes the old triangles from the global mesh, lines 15 to 18. If an encroachment is concerned, the segment is removed from the mesh boundary, and the new edges, which result from spliting the segment in half, are inserted into the mesh boundary from the `newEdge` vector of the region, lines 19 to 23. The last step is inserting the new triangles, lines 24 to 27. The region was built privately by the `retriangulationPriv` function, hence we only have to insert the new triangles in the list of neighbors of their respective neighboring triangles of the global mesh. This information is in the `newList` we added to the `regionType`.

```
1  triangleType *expansionPriv(triangleType *triangle, regionType *region, bool *neighIsGarb) {
2    bool isBoundary = isSegment(triangle);
3    queueType *expandQueue;
4    *neighIsGarb = FALSE;
5    queuePush(expandQueue, triangle);
6    while(!isQueueEmpty(expandQueue)) {  //While there is a triangle to expand
7      triangleType *current = queuePop(expandQueue);
8      listInsert(region->expansionList, current);
9      int i=0; //Keep count of the traversed items
10     list *neighborList = getNeighbors(current);
11     while(listHasNext(neighborList)) { //While there is a neighbor
12       triangleType *neighbor = listNext(neighborList);
13       if(neighbor == NULL) break;
14       if(isGarbage(neighbor)) {
15         *neighIsGarb = TRUE;
16         return NULL;
17       }
18       if(!listFind(region->expansionList, neighbor) {
19         if(inCircumcircle(neighbor, region->circumCenter)) {
20           if(!isBoundary && isSegment(neighbor)) return neighbor; //Encroachment
21           else queuePush(expandQueue, neighbor);
22         } else //Triangle borders region, save for retriangulation
23           listInsert(region->borderList, getCommonEdge(neighbor, current));
24       }
25       i++;
26     } //Traversed different number of neighbors? Conflict
27     if(i != getNeighborListSize(current)) { *neighIsGarb = TRUE; return NULL; }
28   }
29   return NULL;
30 }
31 void retriangulationPriv(regionType *region, triangleType* triangle) {
32   if(isSegment(triangle)) { //If encroachment, split in half
33     region->newEdge[0] = split(triangle)[0];
34     region->newEdge[1] = split(triangle)[1];
35   }
36   while(listHasNext(region->borderList)) { //Create new triangles
37     edgeType *edge = listNext(region->borderList);
38     triangleType *newTriangle = newTriangle(edge, region->circumCenter);
39     regionInsert(region, newTriangle, edge);
40     if(isBadTriangle(newTriangle)) listInsert(region->badList, newTriangle);
41   }
42 }
```

Figure 11: Priv-HTM Ruppert's algorithm implementation (ii).

3.4. Reducing the validation set

Our purpose is to minimize the size of the transaction so that it complies with the HTM system restrictions. However, when using privatizing transactions we must validate all the reads to global variables issued in the privatizing/execution section to ensure that the variables have not changed. This might significantly enlarge the read set of our transactions.

We can reduce the validation set by reasoning whether it is necessary to validate all the reads or only a representative subset of them to detect conflicts.

As regards Lee's algorithm, we read the entire grid but we do not need to validate every point we read, only the points belonging to the calculated path. Actually, validating that subset of the grid encourages parallelism and we do not risk correctness. The algorithm reads the entire grid for the calculation of the path to discard occupied cells. If while the path is calculated, privately, other thread commits a path, two scenarios can be presented: (i) the new path is in the way of the calculated path, the validation detects that at least one point is full, and the transaction is aborted. (ii) the new path is not in the way of the calculated path. The validation does not detect a conflict. If we had validated every global read this situation would have caused an abort.

As far as Ruppert's algorithm is concerned, reducing the validation set may seem straightforward. If at least one

```
 1 bool regionValidation(regionType *region) {
 2   //Are mesh border region triangles garbage?
 3   while(listHasNext(region->newList)) {
 4     pairType pair = listNext(region->newList); // (new, border) triangle pair
 5     if(isGarbage(pair->border)) return FALSE;
 6   }
 7   //Are expansion region triangles garbage?
 8   while(listHasNext(region->expansionList)) {
 9     triangleType triangle = listNext(region->expansionList);
10     if(isGarbage(triangle)) return FALSE;
11   }
12   return TRUE;
13 }
14 void meshUpdate(triangleType *triangle, regionType *region, meshType *globalMesh) {
15   while(listHasNext(region->expansionList)) { //Remove old triangles
16     triangleType *oldTriangle = listNext(region->expansionList);
17     meshRemove(globalMesh, oldTriangle);
18   }
19   if(isSegment(triangle)) { //If encroachment
20     meshRemoveBoundary(globalMesh, triangle);
21     meshInsertBoundary(globalMesh, region->newEdge[0]);
22     meshInsertBoundary(globalMesh, region->newEdge[1]);
23   }
24   while(listHasNext(region->newList)) { //Insert the new triangles
25     pairType *pair = listNext(region->newList);
26     listInsert(pair->border->neighborList, pair->new);
27   }
28 }
```

Figure 12: Priv-HTM Ruppert's algorithm implementation (iii).

of the old triangles in the expansion region or one of those triangles bordering the expansion region is garbage, the transaction is aborted. However, we must reason about what happens with the list of neighbors of a triangle. This list can be accessed by two privatizing transactions at a time. The list is traversed and read, out of transaction, in the privatizing section of a transaction (lines 11 and 12, Figure 11) and is modified inside transaction in the update section (lines 17 and 26, Figure 12). A transaction can delete one item from the list and insert a new one in a different position, as the algorithm uses sorted lists with no duplicates. If there is a coincidence in time of the reader with the writer, the writer is aborted due to strong isolation (a property of the HTM by which non-transactional code may abort transactions [32]). But if there is no such a coincidence, we can be presented three different cases, as shown in Figure 13, or any combination of them:

- Case 1: transaction A has already read item 1 of the list of neighbors of the triangle. Next it reads item 2 (line 12, Figure 11) and gets stalled for some reason. Then, transaction B atomically removes item 2 and inserts a new one in the same place. As removing a triangle implies setting its `isGarbage` field to true and removing it from its neighbors, when transaction A resumes the execution it is working with a garbage triangle. If it resumes in line 14, the conflict is detected. If not, the conflict will be detected in the validation section regardless of whether the garbage triangle eventually belongs to the expansion region or its border.
- Case 2: transaction A reads item 1 and, before reading item 2, transaction B commits removing item 2 and inserting other item at the beginning of the list. In this case, transaction A fails to expand the new triangle inserted by B. We solved this with line 27, Figure 11. If the number of items checked is less than the size of the list, the variable to mark a neighbor as garbage is set to true and the function returns. The triangle will be refined later. There can be a variant for this case where we read more items than the actual size of the list. This can happen if A reads the three neighbors and B inserts a new one at the end of the list and deletes one already read by A. These are corner cases that show up a few times in a mesh refinement not causing overhead.
- Case 3: after checking item 2, transaction A executes line 11 of Figure 11 whose output is true, i.e. there is another item in the list. Subsequently, transaction B commits and removes item 3 by setting the next pointer of

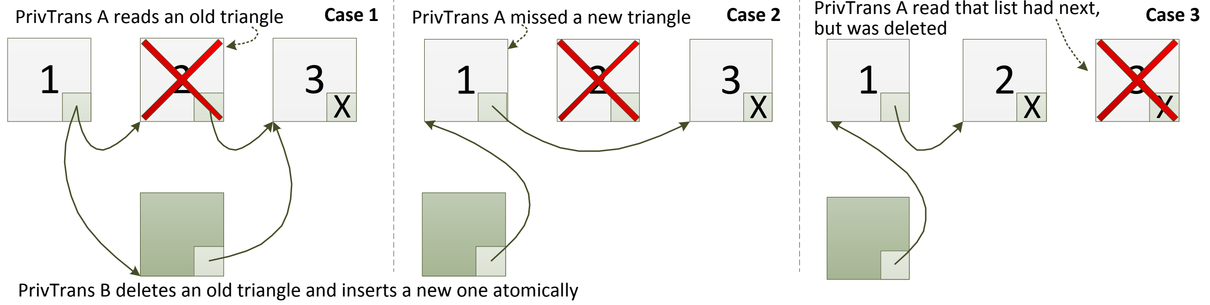


Figure 13: Access to the list of neighbors of a triangle from two different transactions. Possible cases.

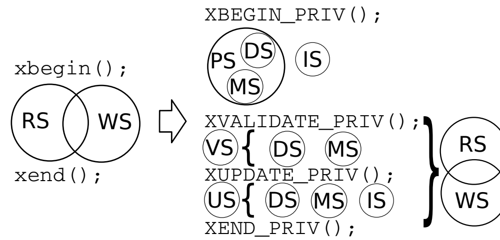


Figure 14: Definition of sets in a privatizing transaction compared with the data sets of a transaction (RS and WS).

item 2 to null. When transaction A executes line 12 it tries to dereferentiate a null pointer and the application crush. We solved this by checking the pointer before dereferentiating it (line 13, Figure 11).

3.5. Generalization

Let PS (Privatization Set) be the set of global objects read in the privatizing and execution section. After the execution of that section, we may have the following sets: let DS (Deletion Set) be the set of objects that have to be deleted from global data, as a result of the privatizing and execution section; let MS (Modification Set) be the set of global objects that have to be modified; let IS (Insertion Set) be the set of new objects that have to be inserted into the global structures. Figure 14 shows the definition of these sets and their relationship with the data set of a transaction (RS and WS).

Proposition 1. *Let be a privatizing transaction with a privatization set of global objects, PS. After the privatizing and execution phase, out of transaction, at least one set of global objects must be obtained, MS, and two other possible sets can be defined, DS and IS. With* $PS \supseteq DS \cup MS$*. Then, the validation section set is* $VS = DS \cup MS$*, and the update section set is* $US = DS \cup MS \cup IS$*.*

The read and write sets of the privatizing transaction satisfies that,

- $RS \subseteq VS \cup VAS$,
- $WS \subseteq US \cup UAS$,

as an object may not be entirely read or written by a transaction, with VAS and UAS being the auxiliary object sets needed to validate and update the transaction.

Applying Proposition 1 to the privatizing transactions of the two applications in this paper we get the following results:

- *Lee's algorithm*: PS =all the grid array cells, $DS = \emptyset$, $MS =$ array cells in the calculated path, $IS = \emptyset$. So the set to validate is $VS = DS \cup MS = MS$. And the set to update is $US = DS \cup MS \cup IS = MS$. The RS and WS are enlarged with the auxiliary variables needed to traverse the path (variables `i`, `isValid`, and function length() in Figure 8. Their cardinality is as concise as the length of the calculated path plus those variables.

- *Ruppert's algorithm*: PS = triangles in the expansion and the border of the expansion. DS = old triangles in the expansion. MS = the border triangles whose neighbors will be modified (old triangles will be replaced with new ones). IS = the new triangles obtained in the retriangulation. So the set to validate is $VS = DS \cup MS$, which are the old triangles to be removed and the border triangles. The update set is $US = DS \cup MS \cup IS$. However, RS and WS are subsets of $VS \cup VAS$ and $US \cup UAS$ since all triangle objects are not entirely read or written. Deleting a triangle consists of removing the pointer of that triangle of the list of neighbors of its neighbors, and inserting the new triangles consists of inserting the pointer of the triangles in the list of neighbors of the border triangles. Thus, the cardinality of the RS and the WS comprises a set of pointers of the neighbor lists, besides the auxiliary variables to traverse the lists. In the vanilla version, the transaction footprint is larger (see Table 4), as it comprises the expansion and retriangulation phases which access more data (e.g. the `triangleType` is a 256 byte structure whose fields are accessed in those phases, while the validation and update sections of the privatizing version only access the `isGarbage` field. The cache line granularity, 64 bytes, makes the data set reduction less significant in the privatizing version) and uses other auxiliary structures, such as the `expandQueue`, and functions, such as `meshInsert` and `newTriangle` which adds to the data set of the vanilla version.

Using Proposition 1, the data set of transactions can be reduced, being a subset of the PS plus the IS. Besides, it can be further reduced in Ruppert's algorithm with the `isGarbage` field, as stated in Section 2.2.2.

In summary, to use privatizing transactions with any application we can follow the next steps:

- Privatize memory accesses by reading from the global data structures and updating local ones.
- Identify the deletion, modification and insertion sets (DS, MS, IS).
- Enclose the validation of DS and MS and the update from local to global data structures in their respective sections (see Figure 7) to enable conflict detection.
- The validation section is dependent on the application and can be simplified by finding a representative subset of data to detect conflicts, thus further reducing capacity overflow aborts.

3.6. Programming Complexity Discussion

Programming complexity can be difficult to evaluate due its subjective nature. Here we present a quantitative measure of complexity based on the lines of code needed for the given implementations of the algorithms, and we discuss the results taking into account other factors.

Table 1 shows the number of lines of code (LOC) for our C implementation of Lee and Ruppert algorithms. Only the core functions of the algorithms are accounted (those described in this paper) and the LOC increase due to privatizing transactions is shown in percentage, both for the core functions and for the entire applications (Total).

Lee's algorithm implementation seems to need a lesser amount of work in terms of LOC. The HTM algorithm in Figure 2 includes the possibility of having open transactions or early release features, but we have deleted them for the LOC comparison as they are not yet supported by commercial HTM systems. Consequently, only line 18 is left from the if clause and the validation in lines 22 and 23 is not needed. This fact, along with the fact that privatizing transactions have more sections to deal with, make that the LOC increase of the core function amounts to 21%. However the LOC increase when it comes to the entire application is only 0.5%.

With regard to Ruppert's algorithm, the LOC increase for the core functions using privatizing transaction is noticeably greater, even for the total LOC of the application. Overall, Ruppert's algorithm is more complex than Lee's in terms of data structures and functions. The validation section is more complex, comprising the traversal of two lists while Lee's algorithm requires validating a single array. Also, although the private retriangulation needs less lines than the normal retriangulation, the mesh update has to repeat the list traversals to remove old triangles and to link the new private triangles to the global mesh.

The LOC measure can give a hint of the complexity of the implementations. However, with this metric, each line of code is accounted as costing the same to produce, which is not always true. To implement the Priv-HTM version of Lee's algorithm we start from a code that already have a privatization of the grid. Furthermore, the grid is a simple array which is easy to deal with. On the other hand, Ruppert's algorithm works with a connected graph

Table 1: Lines of code (LOC) of the C implementation of Lee and Ruppert algorithms.

Algorithm	Vanilla-HTM Core	Priv-HTM Core	LOC Increase Core	Total
Lee	53	64	21%	0.5%
Ruppert	191	268	40%	2.1%

Table 2: Characteristics of the grids for Lee. Aborts and TCR(%), 4 threads.

Dimensions (x,y,z)	Paths (n)	Max/Avg/Dev Path Length	Aborts / TCR(%) Priv-HTM	Vanilla-HTM
(128,128,3)	1024	7 / 4.1 / 1.2	9.0 / 99	5678.2 / 15
(128,128,3)	512	7 / 4.1 / 1.2	4.5 / 99	2844.3 / 15
(64,64,3)	64	94 / 42.9 / 22.1	48.7 / 72	353.8 / 16
(64,64,3)	48	91 / 39.8 / 21.6	17.2 / 85	270.7 / 17
(48,48,3)	64	71 / 33.5 / 16.6	25.7 / 84	358.4 / 16
(48,48,3)	48	69 / 31.1 / 16.1	12.2 / 89	268.7 / 17
(32,32,3)	96	59 / 30.5 / 13.0	37.2 / 84	540.8 / 15
(32,32,3)	64	58 / 27.9 / 12.1	25.5 / 84	368.9 / 15

of triangle structures with lists of neighbors. We had to implement the privatization of the retriangulation phase and, more important, we had to reason about the management of the list of triangles, which is read in the privatization section out of transaction and written atomically in the update section. The programmer must think of the cases discussed in Section 3.4. Therefore, lines 13 and 27 in Figure 11, which solve the problems discussed in cases 2 and 3, add complexity to the programming of the algorithm which is not taken into account by the LOC measure.

4. Experimental Evaluation

4.1. Methodology

For the evaluation of our proposals we have used a machine with an Intel Core i7-7700K chip, code named Kaby Lake, which provides the TSX HTM extensions. The chip includes four cores at 4.20GHz where we can expand eight threads with the hyper-threading (HT) technology.

Regarding HTM characteristics, the Intel Core architecture uses the L1 cache for conflict detection and store buffering. The write set capacity for a transaction is within the size of the L1 cache (32 KB), but the read set size can be greater, about the size of the L3 cache [15].

We have evaluated the implementations of Lee's algorithm in Figure 2 (the standard Lee with neither early release nor open transactions), referred to as Vanilla-HTM Lee, and in Figure 8, the privatizing transaction version called Priv-HTM Lee. For Ruppert's we have used the implementations shown in Figures 4 and 5 for Vanilla-HTM Ruppert, and Figures 10, 11 and 12 for Priv-HTM Ruppert.

For Lee we have evaluated the grids shown in Table 2. The first two larger grids were created randomly specifying a maximum length for the path, shorter than in the other grids to favor the small write set capacity of the HTM system used for evaluation. The rest are taken from the inputs released with the Labyrinth benchmark from the STAMP suite [22]. The maximum, average and standard deviation for path lengths are taken from the sequential application.

Table 3 shows the meshes we have used to evaluate Ruppert's algorithm. They are taken from the inputs released with the Yada benchmark from the STAMP suite. We chose the ones in [22] and tried varying the minimum angle constraint. The greater the angle the larger the amount of bad triangles to refine. We can see the number of triangles in the mesh prior to the refinement (Initial), and how many of them do not comply with the quality constraints (Bad). The final number of triangles in the mesh after their refinement is also shown (Final).

We use pthread affinity to bind each thread to a core in both Vanilla-HTM and Priv-HTM implementations of the algorithms. This way, we keep the threads from migrating from core to core to maintain a low variability in the results.

We used Intel's PIN [33] to analyse the size of the main transaction of the benchmarks. We can see the average size in cache lines in Table 4, separated into read set and write set. The values for Priv-HTM are further split into *Priv*, which stands for the size of the privatizing and execution section of the privatizing transaction, *Validate*, for the validation section, and *Update*, for the update section. Whereas the values showed for Vanilla-HTM comprise the

Table 3: Characteristics of the meshes for Ruppert. Aborts and TCR(%), 4 threads.

Mesh	Angle	Initial / Bad / Final # Triangles	Aborts / TCR(%) Priv-HTM	Aborts / TCR(%) Vanilla-HTM
633.2	20	1264 / 438 / 2678	2922.0 / 66	9251.7 / 27
ttimeu10000.2	10	19998 / 1425 / 24912	5539.8 / 75	25818.7 / 26
ttimeu10000.2	20	19998 / 4897 / 38772	33504.6 / 65	114180.9 / 27
ttimeu100000.2	10	199998 / 12689 / 238572	33296.5 / 77	194458.4 / 28
ttimeu100000.2	15	199998 / 27468 / 288866	113090.5 / 71	500371.8 / 27
ttimeu1000000.2	10	1999998 / 122250 / 2348770	242797.2 / 80	1765359.9 / 27
ttimeu1000000.2	15	1999998 / 267838 / 2840482	979465.3 / 72	4712659.5 / 27

Table 4: Size of the main transaction in cache lines (RS average / WS average).

	Input	Vanilla-HTM	Priv-HTM		
		$\|RS\|$ / $\|WS\|$	Priv	Validate	Update
Lee	x128 y128 z3 n1024	6201.0 / 6164.9	6200.5 / 6163.3	5.8 / 1.0	4.3 / 2.5
Lee	x128 y128 z3 n512	6201.3 / 6165.2	6200.8 / 6163.7	5.8 / 1.0	4.2 / 2.5
Lee	x64 y64 z3 n64	2409.7 / 1630.2	2404.4 / 1606.2	33.1 / 1.0	9.2 / 25.0
Lee	x64 y64 z3 n48	2375.7 / 1630.2	2371.5 / 1609.0	31.1 / 1.0	8.8 / 23.3
Lee	x48 y48 z3 n64	1392.5 / 933.4	1387.3 / 913.9	26.8 / 1.0	8.2 / 19.6
Lee	x48 y48 z3 n48	1375.0 / 931.1	1371.1 / 913.7	25.0 / 1.0	7.6 / 18.4
Lee	x32 y32 z3 n96	615.5 / 424.2	612.2 / 412.0	16.2 / 1.0	5.3 / 12.0
Lee	x32 y32 z3 n64	655.7 / 430.8	652.5 / 430.8	21.4 / 1.0	6.8 / 15.6
Ruppert	633.2 a20	290.2 / 132.2	206.4 / 84.7	18.3 / 1.8	86.6 / 25.3
Ruppert	ttimeu10000.2 a10	302.5 / 131.3	217.0 / 86.3	16.9 / 1.0	85.8 / 24.0
Ruppert	ttimeu10000.2 a20	298.0 / 127.8	235.3 / 92.2	20.8 / 1.9	95.7 / 28.0
Ruppert	ttimeu100000.2 a10	336.9 / 144.8	271.0 / 108.3	20.9 / 1.9	99.1 / 29.0
Ruppert	ttimeu100000.2 a15	322.3 / 138.3	265.8 / 102.8	20.8 / 1.0	101.4 / 29.1
Ruppert	ttimeu1000000.2 a10	360.6 / 156.1	313.3 / 125.5	21.9 / 1.0	105.1 / 30.1
Ruppert	ttimeu1000000.2 a15	336.0 / 140.4	286.5 / 111.0	22.1 / 1.0	105.9 / 30.0

entire transaction, only the validate and update sections comprise the size of the transaction for Priv-HTM. The transaction size reduction is significant in Lee's algorithm, as the entire grid is read and written in the privatizing/execution section out of transaction. In the case of Ruppert's, the reduction is not negligible at all, with the RS shrinking to more than a half of that of Vanilla-HTM and the WS even more.

For the breakdown of abort causes we gathered the information from the EAX register, where the abort status word is stored after an abort [27].

4.2. Results

Figure 15 shows the speedup of the Priv-HTM and the Vanilla-HTM Lee's algorithm implementations over the sequential application on the Intel Core Kabylake machine, from 1 to 8 threads. The speedup of Priv-HTM over Vanilla-HTM is also shown. The transaction-abort rate and a breakdown of the abort cause for both implementations are in Figure 16.

Priv-HTM yields around 2$\times$ speedup over the sequential and HTM versions for the grids with dimensions (32,32,3) and (48,48,3) and 4 cores. Table 2 shows a significant reduction on aborts with respect to the HTM version, as HTM Lee copies the grid inside the transaction. The hardware resources are insufficient to keep track of the entire grid. Consequently, transactions are aborted when privatizing the grid until a given number of retries is reached. Then, the execution is serialized and the performance falls to that of the sequential algorithm or lower. Other than the two transactions shown in Figure 2 there is another transaction for each thread to insert the calculated path in another global list but it rarely takes the fallback path and is low contended. Our Priv-HTM approach reduces the transaction footprint (see Table 4) to the length of the path so the hardware capacity of Intel Core can accommodate the majority of the transactions.

The grids with dimensions (64,64,3) exhibit larger paths in average and maximum. Therefore the transactions in the Priv-HTM Lee application are larger as well and there are more capacity aborts (see Figure 16). Table 2 also shows the TCR (transaction commit ratio) percentage calculated as $\frac{commits}{commits+aborts} \times 100$. For the (64,64,3) grid and 64

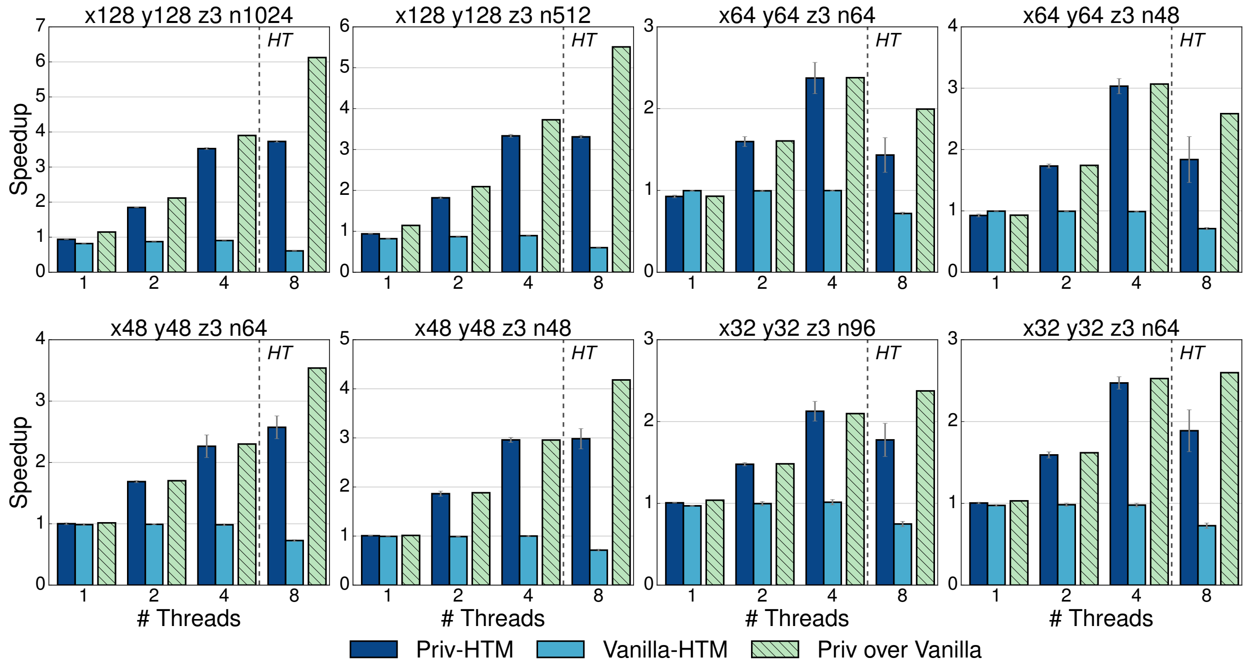


Figure 15: Lee's algorithm. Speedup of the privatizing and the vanilla HTM implementations.

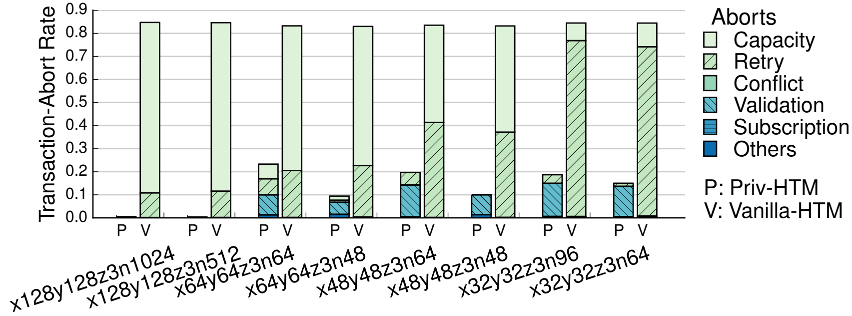


Figure 16: Lee's algorithm. Transaction-abort rate and abort breakdown with 4 threads.

paths the TCR decreases under 80%. In spite of these values the performance still is above twice the sequential with 4 threads as the number of aborts is not high.

The larger grids with dimensions (128,128,3) have shorter paths that allow our Priv-HTM version to scale up to 3.5× over sequential and even more over the HTM version with 8 threads. The TCR is almost 100% compared to a very low 14% of the HTM version which aborts the majority of transactions due to capacity overflow (see Figure 16). The few aborts of Priv-HTM are due to conflicts between paths (the explicit abort when validating the path — Validation in Figure 16), Capacity aborts because of insufficient cache capacity, and Others (debug break points, abort in nested transaction, or abort status word equal to 0). Retry abort status definition given by the TSX system hints that the transaction may succeed on a retry. We retry the transaction up to five times before taking the fallback path. It is not clear how Intel's HTM system decides to set the retry bit in the abort status word. Non-persistent events such as descheduling, page faults, conflicts or cache conflict misses may cause the system to flag the retry bit. By monitoring the other bits while the retry bit was set we found that the conflict bit was set most of the time, and also the capacity bit to a lesser extent. We found cases where both were set at the same time.

With regard to Ruppert's algorithm, Figure 17 shows the speedup of the Priv-HTM and the Vanilla-HTM versions over the sequential application, from 1 to 8 threads. The speedup of Priv-HTM over Vanilla-HTM is also depicted.

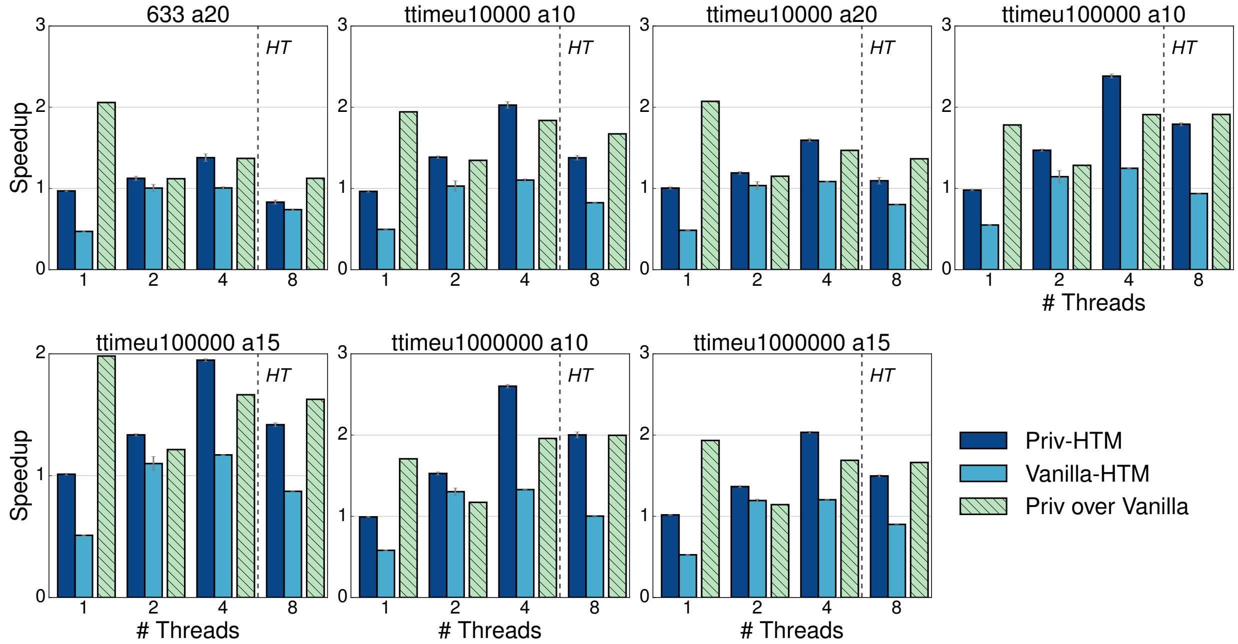


Figure 17: Ruppert's algorithm. Speedup of the privatizing and the vanilla HTM implementations.

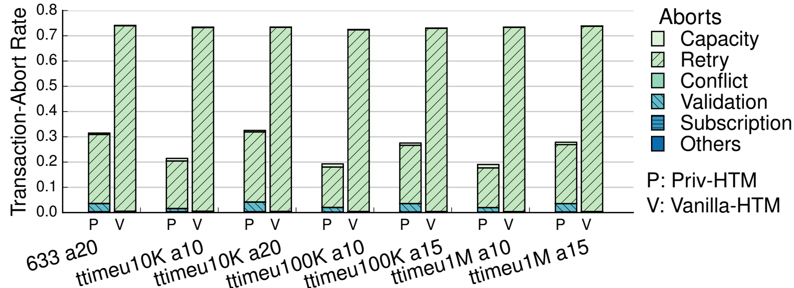


Figure 18: Ruppert's algorithm. Transaction-abort rate and abort breakdown with 4 threads.

The breakdown of the abort cause for both implementations is shown in Figure 18.

Overall, the performance is not as good as that of the Priv-HTM Lee implementation. The nature of the data structures in Ruppert's algorithm, which are dynamic (e.g. graph, lists,...), can harm cache locality thus reducing the write set capacity. Also, for small meshes with greater angle constraints (e.g. 633-a20 and ttimeu10000-a20) there are more triangles to refine within a smaller space (see Table 3), which leads to more conflicting transactions. For meshes 633-a20 and ttimeu10000-a20 the speedup is barely 1.5× for 4 threads. Even so, the TCR is twice as much as the Vanilla-HTM version, see Table 3.

On the other hand, larger meshes and smaller angles give more space between bad triangles and lower number of triangles to refine, which yields more space for performance gain. For instance, Priv-HTM with ttimeu1000000-a10 reaches the best speedup. It is more than 2.5× faster than the sequential application with four threads and has a TCR of 80%.

With only one thread, Vanilla-HTM obtains poor results whereas our Priv-HTM implementation keeps as good as the sequential. The reason for this behavior is two-fold. First, privatizing transactions reduce the transaction footprint so more transactions can commit without overflowing the HTM resources. Second, in case a transaction overflows, it is retried until a number of retries is reached. With the Priv-HTM API of Figure 7, only the validation and update sections are retried since there are not validation errors (only one thread is running). The privatizing and execution

section is not reexecuted, which is where the bulk of the work is performed. Consequently, time is not wasted such as by the Vanilla-HTM implementation.

We can see a dissimilar behavior of both algorithms when using 8 threads. As mentioned in the methodology, the machine used for evaluation has 4 physical cores which can deal with 8 threads by means of hyperthreading (HT in the figures). When using hyperthreading, the machine shares hardware resources between those threads sharing the same physical core. Consequently, the transactional footprint per thread that can be handled by the machine is smaller. This fact seems to have more impact on Ruppert's algorithm than on Lee's, which gives us an idea of the length of the transactions in each application (Priv-HTM average transaction size is lower for Lee's than for Ruppert's, as we can see in Table 4). For Lee's, 8 threads give the best speedup with grids (128,128,3). Those grids have the smallest paths and the L1 cache can store two different thread transactions. The rest of the grids show a small dip in performance with respect to the 4 thread run.

However, Ruppert's application performance can fall with the use of 8 threads because of hyperthreading hardware resource sharing. The transactions are larger and less locality-aware in Ruppert's algorithm, so there is not enough space to hold two different transactions from different threads. Our Priv-HTM implementation, though, remains better than the vanilla version in all cases.

Figure 18 shows a huge amount of retry aborts for all meshes. As we mentioned earlier, it may be caused by events that the HTM system thinks are non-persistent, such as conflicts, cache conflict misses, etc. In fact, conflict misses can be a problem in Rupperts's algorithm because of the graph structure used to hold the mesh of triangles. As stated in [13], these kind of structures result in very non-local memory accesses.

5. Related Work

Parallel versions of Lee's algorithm are difficult to find in the bibliography. Although it seems an inherently parallel algorithm where the calculation of each path is the unit of parallelism, traditional synchronization methods can lead to different problems. If we protect each path calculation with a coarse grain lock the execution is serialized. Conversely, protecting each grid cell with a fine grain lock may lead to deadlock scenarios. Actually, Won et al. [34] and Yen et al. [35] approach the parallelization of the algorithm from another point of view. They partition the grid among processors using different mapping strategies.

Similarly to Lee's algorithm, Delaunay's parallelization approaches in the bibliography make use of data parallelism with MPI. The global data structure is distributed in partitions which are locally stored in each processor [36, 37]. Communication among processors is needed whenever a triangle refinement expands upon the boundaries of the local mesh partition of a processor. The modification of these boundary triangles is challenging. Beyer et al. [36] introduce blocking status in each boundary triangle to guarantee that only one process has access rights to it. Starinshak et al. [37] propose sharing the subset of the mesh intersected by its local boundary to its neighbors. Then, a second stage of the algorithm is performed where shared edges and nodes are identified and reduced.

A different approach to Delaunay parallelization is addressed in [13]. A GPU algorithm is proposed by Sussman, where data structures are forced to be GPU-compliant. Sussman points out that CPU-convenient data structures, such as doubly connected edge lists, are not so convenient for GPUs as they result in very non-local memory accesses. Consequently, Sussman proposes initializing and maintaining many redundant data structures such as one-dimensional lists of adjacent vertices in counter-clockwise order.

As far as privatizing transactions are concerned, we can find related work in the bibliography [38, 39], in the context of data bases and search tree data structures. Their approach is to validate all the private reads performed outside transactions, for which they use application dependent metadata stored in B-trees and stacks respectively. On the contrary, privatizing transactions offer a general purpose API. Furthermore, in this paper we show how the validation set can be reduced to a small subset by reasoning on the implementation of the algorithm. Thus, the transaction size is further reduced to comply with the hardware limitations of the underlying HTM system.

A fallback path is needed in best-effort HTM systems to provide an alternative to transactional execution when transactions cannot progress. The fallback usually comprises a lock enclosing the same code of the transaction. All transactions have to subscribe to the lock by reading it before performing any computation so that they are aborted when the fallback acquires the lock. Transactional and non-transactional code are not allowed in parallel. Conversely, lazy subscription to the fallback lock is used in the implementation of improved global lock fallbacks [40, 41] and

consists in subscribing to the lock at the end of the transaction instead of at the beginning. This way, concurrency between non-transactional fallback code and transactional code is allowed. To ensure a correct execution, the HTM system must comply with strong isolation [32], by which non-transactional memory accesses can abort those transactions holding the accessed memory locations in their data sets, and sandboxing, which prevents the propagation of errors from inside a hardware transaction to other transactions. Thus, writes caused by reading inconsistent state from non-transactional fallback computation are not visible to other threads.

6. Conclusions

This paper proposes a new parallel approach to Lee's and Ruppert's algorithms for commercial HTM systems. These algorithms seem to be HTM-friendly, but the current proposals lead to sequential performance as transactions are too large and long for the underlying HTM system to deal with.

We propose an implementation using privatizing transactions, a new API implemented with commercial HTM extensions. The main phases of the algorithms are executed non-transactionally in a private execution section of the transactions. Consequently, the transaction size is reduced, thus being more suitable for the limited resources of HTM systems. Furthermore, we propose reducing even more the transaction size by validating only a representative subset of all the global reads issued in the private section. We also discuss the programming complexity involved in the resulting codes. More complex dynamic data structures are used in Ruppert's implementation which increase the programming complexity of the algorithm.

The evaluation with the Intel Core architecture and the restricted transactional memory extensions shows speedups ranging from 2$\times$ to 3.5$\times$ for our privatizing transaction implementation of Lee's and Ruppert's algorithms over the sequential and HTM versions with four threads.

Acknowledgements

This work has been supported by the Government of Spain under projects TIN2013-42253-P and TIN2016-80920-R, and Junta de Andalucía under project P12-TIC-1470.

References

[1] M. Herlihy, J. Moss, Transactional memory: Architectural support for lock-free data structures, in: 20th Ann. Int'l. Symp. on Computer Architecture (ISCA'93), 1993, pp. 289–300.

[2] Transactional memory, Journal of Parallel and Distributed Computing 70 (10) (2010) 993 – 1008, transactional Memory.

[3] L. Hammond, V. Wong, M. Chen, B. Carlstrom, J. Davis, B. Hertzberg, M. Prabhu, H. Wijaya, C. Kozyrakis, K. Olukotun, Transactional memory coherence and consistency, in: 31th Ann. Int'l. Symp. on Computer Architecture (ISCA'04), 2004, pp. 102–113.

[4] C. Ananian, K. Asanovic, B. Kuszmaul, C. Leiserson, S. Lie, Unbounded transactional memory, in: 11th Int'l. Symp. on High-Performance Computer Architecture (HPCA'05), 2005, pp. 316–327.

[5] C. Blundell, J. Devietti, E. C. Lewis, M. M. K. Martin, Making the Fast Case Common and the Uncommon Case Simple in Unbounded Transactional Memory, in: 34th Ann. Int'l. Symp. on Computer Architecture (ISCA'07), ISCA '07, 2007, pp. 24–34.

[6] J. Bobba, N. Goyal, M. D. Hill, M. M. Swift, D. A. Wood, TokenTM: Efficient Execution of Large Transactions with Hardware Transactional Memory, in: 35th Ann. Int'l. Symp. on Computer Architecture (ISCA'08), 2008, pp. 127–138.

[7] R. Rajwar, M. Herlihy, K. Lai, Virtualizing transactional memory, in: 32th Ann. Int'l. Symp. on Computer Architecture (ISCA'05), 2005, pp. 494–505.

[8] L. Yen, J. Bobba, M. R. Marty, K. E. Moore, H. Volos, M. D. Hill, M. M. Swift, D. A. Wood, LogTM-SE: Decoupling Hardware Transactional Memory from Caches, in: 13th Int'l. Symp. on High Performance Computer Architecture (HPCA'07), 2007, pp. 261–272.

[9] D. Christie, J.-w. Chung, S. Diestelhorst, M. Hohmuth, M. Pohlack, C. Fetzer, M. Nowack, T. Riegel, P. Felber, P. Marlier, E. Rivière, Evaluation of AMD's Advanced Synchronization Facility Within a Complete Transactional Memory Stack, in: 5th European Conf. on Computer Systems (EuroSys'10), EuroSys '10, 2010, pp. 27–40.

[10] H. W. Cain, M. M. Michael, B. Frey, C. May, D. Williams, H. Le, Robust Architectural Support for Transactional Memory in the Power Architecture, in: 40th Ann. Int'l. Symp. on Computer Architecture (ISCA'13), 2013, pp. 225–236.

[11] C. Jacobi, T. Slegel, D. Greiner, Transactional Memory Architecture and Implementation for IBM System z, in: 45th Ann. Int'l. Symp. on Microarchitecture (MICRO'12), 2012, pp. 25–36.

[12] R. M. Yoo, C. J. Hughes, K. Lai, R. Rajwar, Performance Evaluation of Intel Transactional Synchronization Extensions for High-performance Computing, in: Int'l Conf. on High Performance Computing, Networking, Storage and Analysis (SC'13), 2013, pp. 19:1–19:11.

[13] D. M. Sussman, cellGPU: Massively parallel simulations of dynamic vertex models, Computer Physics Communications 219 (2017) 400–406.

[14] R. Quislant, E. Gutierrez, E. L. Zapata, O. Plata, Privatizing transactions for lee's algorithm in commercial hardware transactional memory, The Journal of Supercomputing 74 (4) (2018) 1676–1694.

[15] B. Goel, R. Titos-Gil, A. Negi, S. A. Mckee, P. Stenstrom, Performance and Energy Analysis of the Restricted Transactional Memory Implementation on Haswell, in: 28th Int'l Symp. on Parallel and Distributed Processing (IPDPS'14), 2014, pp. 615–624.

[16] T. Nakaike, R. Odaira, M. Gaudet, M. M. Michael, H. Tomari, Quantitative comparison of hardware transactional memory for Blue Gene/Q, zEnterprise EC12, Intel Core, and POWER8, in: 42nd Ann. Int'l. Symp. on Computer Architecture (ISCA'15), 2015, pp. 144–157.

[17] C. Y. Lee, An Algorithm for Path Connections and Its Applications, IRE Transactions on Electronic Computers EC-10 (3) (1961) 346–365.

[18] I. Watson, C. Kirkham, M. Lujan, A study of a transactional parallel routing algorithm, in: 16th Int'l. Conf. on Parallel Architecture and Compilation Techniques (PACT '07), 2007, pp. 388–398.

[19] M. Herlihy, V. Luchangco, M. Moir, W. N. Scherer, III, Software transactional memory for dynamic-sized data structures, in: 22nd Ann. Symp. on Principles of distributed computing (PODC '03), 2003, pp. 92–101.

[20] M. J. Moravan, J. Bobba, K. E. Moore, L. Yen, M. D. Hill, B. Liblit, M. M. Swift, D. A. Wood, Supporting Nested Transactional Memory in logTM, in: 12th Int'l. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS'06), 2006, pp. 359–370.

[21] M. Ansari, C. Kotselidis, I. Watson, C. Kirkham, M. Luján, K. Jarvis, Lee-TM: A Non-trivial Benchmark Suite for Transactional Memory, in: Int'l. Conf. on Algorithms and Architectures for Parallel Processing (ICA3PP'08), 2008, pp. 196–207.

[22] C. Minh, J. Chung, C. Kozyrakis, K. Olukotun, STAMP: Stanford Transactional Applications for Multi-Processing, in: IEEE Int'l Symp. on Workload Characterization (IISWC'08), 2008, pp. 35–46.

[23] M. Herlihy, V. Luchangco, M. Moir, A Flexible Framework for Implementing Software Transactional Memory, in: Conf. on Object-oriented Programming Systems, Languages, and Applications (OOPSLA'06), 2006, pp. 253–262.

[24] P. Felber, C. Fetzer, T. Riegel, Dynamic Performance Tuning of Word-based Software Transactional Memory, in: Symp. on Principles and Practice of Parallel Programming (PPoPP'08), 2008, pp. 237–246.

[25] J. Ruppert, A Delaunay Refinement Algorithm for Quality 2-Dimensional Mesh Generation, Journal of Algorithms 18 (3) (1995) 548–585.

[26] M. Kulkarni, L. P. Chew, K. Pingali, Using transactions in Delaunay mesh generation, in: Workshop on Transactional Memory Workloads, 2006.

[27] Intel, Intel(R) Architecture Instruction Set Extensions Programming Reference, Tech. Rep. February (2012).

[28] A. Baldassin, E. Borin, G. Araujo, Performance Implications of Dynamic Memory Allocators on Transactional Memory Systems, in: Symp. on Principles and Practice of Parallel Programming (PPoPP'15), 2015, pp. 87–96.

[29] M. Machado Pereira, M. Gaudet, J. Nelson Amaral, G. Araujo, Study of hardware transactional memory characteristics and serialization policies on Haswell, Parallel Computing 54 (2016) 46–58.

[30] D. Dice, M. Herlihy, D. Lea, Y. Lev, V. Luchangco, W. Mesard, M. Moir, K. Moore, D. Nussbaum, Applications of the Adaptive Transactional Memory Test Platform, in: 3rd Workshop on Transactional Computing (TRANSACT'08), 2008.

[31] R. Quislant, E. Gutierrez, E. L. Zapata, O. Plata, Insights into the Fallback Path of Best-Effort Hardware Transactional Memory Systems, in: Int'l. Conf. on Parallel Processing (Euro-Par'16), 2016, pp. 251–263.

[32] M. M. K. Martin, C. Blundell, E. Lewis, Subtleties of Transactional Memory Atomicity Semantics, IEEE Computer Architecture Letters 5 (2) (2006) 17.

[33] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. Reddi, K. Hazelwood, PIN: Building customized program analysis tools with dynamic instrumentation, in: Conf. on Programming Language Design and Implementation (PLDI'05), 2005, pp. 190–200.

[34] Y. Won, S. Sahni, Maze routing on a hypercube multicomputer, The Journal of Supercomputing 2 (1) (1988) 55–79.

[35] I. L. Yen, R. M. Dubash, F. B. Bastani, Strategies for mapping Lee's maze routing algorithm onto parallel architectures, in: Int'l. Parallel Processing Symposium, 1993, pp. 672–679.

[36] T. Beyer, G. Schaller, A. Deutsch, M. Meyer-Hermann, Parallel dynamic and kinetic regular triangulation in three dimensions, Computer Physics Communications 172 (2) (2005) 86–108.

[37] D. P. Starinshak, J. M. Owen, J. N. Johnson, A new parallel algorithm for constructing Voronoi tessellations from distributed input data, Computer Physics Communications 185 (12) (2014) 3204–3214.

[38] Z. Wang, H. Qian, J. Li, H. Chen, Using restricted transactional memory to build a scalable in-memory database, in: European Conf. on Computer Systems (EuroSys'14), 2014, pp. 1–15.

[39] D. Siakavaras, K. Nikas, G. Goumas, N. Koziris, RCU-HTM: Combining RCU with HTM to Implement Highly Efficient Concurrent Binary Search Trees, in: Int'l. Conf. on Parallel Architectures and Compilation Techniques (PACT'17), 2017, pp. 1–13.

[40] I. Calciu, T. Shpeisman, G. Pokam, M. Herlihy, Improved Single Global Lock Fallback for Best-effort Hardware Transactional Memory, in: 9th Workshop on Transactional Computing (TRANSACT'14), 2014.

[41] Enhancing scalability in best-effort hardware transactional memory systems, Journal of Parallel and Distributed Computing 104 (2017) 73 – 87.