Multithreaded Delaunay Triangulation

Chaman Singh Verma
Department of Computer Science
The College of William and Mary
Williamsburg, Virginia 23187
USA

October 4, 2004

Contents

Abstract

At first glance, Delaunay triangulation using Bowyer-Watson algorithm seems to be inherently sequential, but by exploiting Delaunay triangulation properties, it is possible to parallelize this kernel very efficiently. This report presents both coarse grained and fine grained multithreaded algorithm. Coarse grained multithreading uses optimistic concurrency control mechanism, whereas fine grained multithreaded algorithm exploit hardware supported multithreading being commercially available on Intel SMT processors.

Chapter 1

Introduction

The Delaunay triangulation (DT) have been extensively studied and many robust sequential implementations are available. [4] In literature, there are many theoretical parallel algorithms for DT, but their reliable implementation and availability are not very impressive. DT falls under the category of irregular applications, which pose different set of problems for both shared memory and distributed memory machines. Because of the dynamic and unpredictable nature of DT algorithm, a naive implementation may suffer from excessive synchronization in both architectures. On distributed memory machines, distributed data management becomes a major performance issue.

There are many DT algorithms such as incremental, divide-and-conquer and sweepline algorithm which have been used in both sequential and parallel environments. Perhaps the easiest among them is Bowyer-Watson incremental method [1], in which vertices are added one-by-one and at each stage Delaunay properties are maintained.

In this paper, I will present an efficient implementations of Bowyer-Watson Delaunay triangulation using threads. The first implementation is Coarse grained multithreading which is based on the heuristic that threads can work in spatially distant domains and conflicts will be rare. If any conflicts occurs, they can be resolved at later time, The second approach is based on the observation that numerically the most expensive part of DT is cavity calculation and therefore parallelization must be exploited in the kernel.

This paper is organized as follows: section II, describes the basic Bowyer-Watson algorithm, section III gives a brief overview of coarse grained approach, section IV describes about the fine grained approach and finally section V presents the conclusions.

1.1 Basic Algorithm

The basic Bowyer-Watson algorithm is an incremental algorithm in which vertices are inserted one-by-one. When a new vertex is inserted in an existing mesh, all the elements that violate the *Delaunay empty circumcircle property* form a *Cavity*. The elements in this cavity are removed and new elements are formed using new vertices as an apex of the cavity. This process is continued till all the vertices are inserted into the mesh. The pseudo code of sequential Bowyer-Watson is as follows:

Bowyer_Watson_Kernel$(\mathcal{M}_0, vertexQ)$
$i = 0$
while $(!vertexQ.empty())$ **do**
$\mathcal{V}_a \leftarrow vertexQ.pop()$ //This vertex becomes the apex of cavity
$\mathcal{E}_s \leftarrow SearchSource(\mathcal{M}_i, \mathcal{V}_a)$
$\mathcal{E}_v \leftarrow \{c, c \in \mathcal{M}_i$ that violates the Delaunay Property$\}$
$\mathcal{E}_n \leftarrow \{c, c$ a new element$c \notin \mathcal{M}_i$ and include $\mathcal{V}_a\}$
$\mathcal{M}_{i+1} \leftarrow \mathcal{M}_i - \mathcal{E}_v + \mathcal{E}_n$
$i \leftarrow i + 1$
enddo

The efficiency of the DT kernel depends on the following three factors:

1. **Point location:** Given a vertex $\mathcal{V}_a$, a point location algorithm finds the element in the existing mesh $\mathcal{M}$ that includes the vertex. This element will always violate the Delaunay empty sphere property and therefore acts as a seed for the growth of the cavity. For DT, point location is carried out using either *random walk* technique or using spatial trees such as K-D trees as background mesh. In mesh generation, a vertex is inserted at the circumcenter of the bad elements, therefore a good hint for point location is already known.
2. **Cavity Creation:** Given a mesh $\mathcal{M}_i$ and the source element $\mathcal{E}_\mathcal{S}$ of a cavity, the cavity is expanded either as breadth-first (BFS) or depth-first (DFS) strategy starting from source element. All the elements that violate the Delaunay property are included in the violation set $\mathcal{E}_v$.
3. **Data Structures:** DT algorithm require efficient *lookup, insert* and *delete* operations from its data structures. For this purpose, we use *map, hashmap* and *set* data structures from STL.

1.2 Coarse Grained Multithreading(CGM) Approach

In the CGM method, multiple cavities are expanded in parallel by multiple threads during the transaction phase in which the global data structures are used only for identifying the cavities. Since most of the operations on data are *read-only* in this phase, we avoid using locks which is the main reason behind performance degradation in multi-threaded implementation. Conflict resolution phase and commit phase constitutes a very small fraction of the overall computation execution time and are carried out by a single thread. Experiments show that it is possible to achieve more than 95% efficiency for both 2D and 3D kernels.

1.2.1 Optimistic Concurrency Control

Optimistic Concurrency Control (OCC) have been widely used in databases and operating systems. A good and brief information can be found in [3]. This method works on the following assumptions

- Each task is a transaction with the computational resources.
- The number of tasks are quite huge and any conflicts among them are rare during transactions.
- If the conflicts occurs, the task can be recomputed.

- If the conflicts are rare, the speculative execution without locking pay very high dividends.

This model of concurrency control is well suited for DT using incremental algorithm. In DT, calculation of cavity for each new vertex is considered as one task, number of vertices for many practical mesh generation is very large $(10^9 - 10^{12})$, many good spatial heuristic strategies can reduce the conflicts and strict order of vertex insertion is not very important. With these ideal conditions, we apply OCC method for mesh generation.

The OCC method have three distinct phases *Transaction Phase, Validation Phase* and *Commit Phase*. During the *transaction phase*, concurrent tasks are executed without locking. In the *validation phase*, all the tasks are scanned for possible conflicts identified and in the *commit phase*, all the successful transactions are made persistent. These three phases are shown in the figure 1.1.

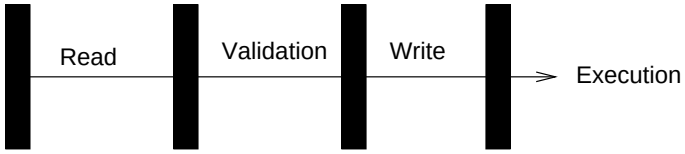

Figure 1.1: Three phases of Optimistic Transactions

1.2.2 Multithread Delaunay Triangulation Kernel (MDTK)

In this section, I describe all the three phases of OCC method that are applied to DT kernel. The figure 1.2 shows both sequential and multi-threaded flow chart of the basic algorithm.

- *Transaction Phase:* The main executing thread, creates a fixed number of threads and passes three arguments to each thread (1) A pointer to the mesh object (2) set of vertices that are assigned to the thread (3) starting sequenceID. Each thread searches for the source and expand the cavity for each of the vertices assigned to it. An expanded cavity includes three element set *violationSet, newElementSet* and *protectionSet. violationSet* contains the elements of the mesh that violate the Delaunay property, *newElementSet* contains new elements that will be created if the cavity is committed in the phase III and *protectionSet* are the elements that are protecting the cavity (described later). Each thread runs concurrently even in the presence of possible conflicts that are resolved later.
- *Validation Phase:* When all the threads have completed their work, they are joined and, the validation process starts. This phase require sequential execution and therefore executed by a single main thread. This phase identifies conflicts and assign valid/invalid flag to each cavity. Each cavity in the order of increasing *sequenceID* assign its *sequenceID* to all its elements i.e *violatioSet* and *protectionSet* if it is not marked. If all the cavity elements possess the same sequenceID, which signifies that no conflict has taken place. If any one of the element in the cavity has lower sequenceID, means that conflict has occurred with the previous cavities. A conflicting cavities releases all its sequenceIDs and marked invalid. Only the valid cavities proceed towards committing phase.
- *Commit Phase:* This phase is also performed by a single thread as (1) it require global data structure updates (2) and the execution time is very small compared to creating new threads for this phase. The *violationSet* elements are marked removed and appended to the objectpool, *newElementSet* elements are made persistent, all the sequenceID of *protectionSet* elements are set to default value and memory pool of elements is checked and refilled. The apex vertices of invalid cavities and threads are returned back to their respective pools for reuse in the next round of transactions.

When the conflicts occurs, the vertex can be inserted at the front or back of the vertex queue.

- *Insert conflicting vertex in the end of the queue:* This is very simple to implement and this strategy forgets everything about the cavity and all the computations done with it.
- *Insert conflicting vertex in the front of the queue:* Many cavities fail because they overlap with *protectionSet* of other cavity. If this cavity is inserted in the front of the vertexQ, then at least we can take advantage of searching the source of the cavity next time based on its previous search, hoping that at least one of the elements from the previous *protectionSet* still survive in the mesh. Also, the cavity expansion could be minimized, if we can skip Delaunay test properties on the elements, which still exist in the mesh. From the experiments and intuitions, I feel that the optimization gained from this approach may not be significant, if good KD trees for point location is used. Although it might make significant improvement, if the random walk algorithm is used.

The C++ implementation of the core MDTK is as follows:

```
1    while( !vertexQ.empty() ) {
2         numSpawned = 0;
3         //
4         // Expansion phase: It involves mostly "Read only" operation, therefore
5         // many cavities can be expanded concurrently.
6         //
7         for(int tid = 0; tid < numThreads; tid++) {
8            vertex_assigned.clear();
9
10            for( int i = 0; i < numCavitiesPerThread; i++){
11                 if( vertexQ.empty() ) break;
12                 vertex_assigned.push_back( vertexQ.front() );
13                 vertexQ.pop_front();
14            }
15
16            start_id = tid*numCavitiesPerThread;
17            BWKernel *newkernel = new BWKernel(outGrid, vertex_assigned, start_id);
18            if( newkernel ) {
19                bwkernel[numSpawned] = newkernel;
20                bwkernel[numSpawned]->setBoundingBox( &rootBox );
21                bwkernel[numSpawned]->start();
22                numSpawned++;
23                numCavities += vertex_assigned.size();
24            }
25        }
26        //
27        // Validation Phase:  Check for conflicts among the cavities ..
28        //
29        vector<Cavity*>  vcavity;
30        for(int tid = 0; tid < numSpawned; tid++) {
31             bwkernel[tid]->join();
32             vcavity = bwkernel[tid]->getCavities();
33             for( int j = 0; j < vcavity.size(); j++)
34                 vcavity[j]->resolve_conflicts();
```

```
35        }
36        //
37        // Commit Phase : All the valid cavities are committed now.
38        //
39        conflicts.clear();
40        for(int tid = 0; tid < numSpawned; tid++) {
41             vcavity = bwkernel[tid]->getCavities();
42             for( int j = 0; j < vcavity.size(); j++){
43                 vcavity[j]->release();
44                 if( vcavity[j]->getStatus() == Cavity::BLOCKED ) {
45                     vertexQ.push_back( vcavity[j]->getApex() );
46                     numSetbacks++;
47                 } else
48                     vcavity[j]->commit();
49                 delete vcavity[j];
50             }
51             delete bwkernel[tid];
52        }
53    }
```

1.2.3 Changes from Sequential Code

In the section, I describe the changes that I performed, to obtain scalable MDTK algorithm. Some changes are essential and some changes which were required for MT also improved the sequential non-multithreaded version.

Identification of Conflict Region

The main issue in moving from sequential code to parallel code is to identify the conflicts and then resolve them. To identify the *Conflict region* we look for DT properties and data structure modification requirements (Figure 1.3)

According to DT properties, if each edge (or face) is Delaunay, then the entire mesh is Delaunay or vice-versa. Therefore, all the elements that share an edge(or face) with the cavity elements should be restricted and no new point insertion are allowed in those elements.These elements are strictly protected in order to maintain the Delaunay property and therefore grouped in *strictSet*.

The current data structures store vertex-element adjacency information. When a cavity is committed, all the vertices that form the outer boundary of the cavity, require changes in their adjacency information. Since, we are not allowed to use locks, all the elements that share the cavity vertices, become part of conflicting region. Elements which are in this set but not in the *strictSet*, are grouped in *relaxSet*. These are relaxed because any conflict with this set does not require recalculation of the cavity, but can be delayed for data structures changes.

Scalable multiprocessor memory allocation

Standard memory allocation functions such as *malloc* and *calloc* are suitable only for single-threaded applications. These functions utilize single heap and thus only one thread can allocate/deallocate memory at a time. The perfor-

mance of multithreaded application is severely affected by this contention of the resource. *Hoard* is a fast, highly scalable memory allocator library for multithreaded applications which maintains per-processor heaps and one global heap. Each thread can access only its heap and the global heap. When a per-processor heap's usage drop below a certain fraction, *Hoard* transfer a large fixed-size chunk of its memory from the per-processor heap to the global heap where it is then available for reuse by another processors. Hoard also reduces false sharing and memory fragmentation. We experimented with other multithreaded memory allocation library such as *ptmalloc* and *mtmalloc*. The results shows that *Hoard* significantly outperforms other libraries as much as 5-10 times. For more information, please refer to Hoard [2].

Object Reuse through Memory pool

DT is dynamic algorithm in which many objects are created and destroyed dynamically. For large size applications, recycling can effectively reduce new/delete operations. A simple, single-threaded templated memory pool class provide a good solution for this application.

```
template<class T>
class MemoryPool
{
  public:
      MemoryPool ( size_t size = EXPANSION_SIZE );
     ~MemoryPool();

      inline void* alloc( size_t size);
      inline void  free( void *elem);
  private:
      MemoryPool<T> *next;
      enum { EXPANSION_SIZE = 512};
      void expand_freelist( int nsize = EXPANSION_SIZE );
}
```

New objects are requested from the pool. Whenever a new object is requested, the pool returns the head of the list and whenever an object is deleted the object is inserted at the end of the pool. The size of the pool can be changed by the user.

In this application, we apply this technique at various places. Although, it improves the sequential algorithm also, but it is important for MT implementation for retriangualtion to go in parallel as described below.

- A thread object is instantiated whenever a set of vertices are inserted in the mesh. Once the cavity for each vertex is identified, the thread object is deleted. For large number of vertices, this incur cost of object creation and deletion, which can be minimized by using the pool of threads.
- During re-triangulation of the cavity, old simplexes are deleted from the data structure and new simplexes are created and inserted into the data structures as described earlier. In multithreaded implementation, whenever possible, we try to avoid using *new* and *delete* by recycling the objects that have been marked deleted. In the present application, the objectpool is expanded in the III stage by single thread. It is highly unlikely that objectpool will run out of objects when required in the stage I (they can still be created at the expense of performance slight performance loss). With this approach, the re-triangulation stage also involve mostly *Read only* operations.

1.2.4 Standard Template Library

We use Standard Template Library as an extremely useful blackbox. In our application, STL is used extensively, therefore its performance in multithread application is critical. STL was primarily designed for single-threaded applications, therefore we must consider the following issues

- STL use reference counting for performance considerations. Reference counting may uses *mutex* or read-writer locks to lock the counter value.
- For each data container, STL provide an argument for *allocator* which is customizable by the users. Different implementations of STL uses different default allocators. For example, SGI STL uses *alloc* as default allocator, which uses standard memory functions for allocation and deallocation purpose. *Hoard* library solves this problem, but different implementation may choose other allocator, therefore portability becomes an issue.
- During the transaction phase, we want to avoid *locking* as much as possible because this phase mostly involves *read-only* operations from global memory. Using *const iterators* for container traversal, and using *const* qualifier whenever possible help compiler in optimizing the application.

1.2.5 Experiments and Results

The entire application is written in C++ and the results are obtained using Sun Ultra2, 450MHz, 4096MB RAM with quad CPU configuration available at Sciclone Cluster at the College of William and Mary. The compilation was done with Sun CC with -O5 optimization flag.

The standard Pthread library is written in C and our entire data structures have been written in C++. We adopted CommonC++ which provide very high level pthreads function in C++.

For experiments I generated random points using *drand48()* of different size $(10 - 10^5)$. At present, the user need to provide an input for number of cavities that is assigned to each thread. This parameter amortizes the cost of threads creation cost, but as this number goes up, it also creates more conflicts and therefore will require more recalculations. Since, in general, this number depends on the input, a good number can be found by performing some experiments. For all the test cases, I found that four cavities per thread produced the best execution time. In 3D, this number may be less, as each thread have more work to do than in 2D.

- **Is optimism good ?** In this experiment, we wanted to know what fraction of cavity creation suffer setbacks as the number of threads grow. The figure 1.5 shows that the conflicts are dominant in the start, but as the number of elements in the mesh increases, they start decreasing. For reasonably large mesh, we found that number of setbacks are less than 5%, which make the initial assumption correct that conflicts are rare.
- **Memory allocator performance** In this experiment, we used standard malloc, mtmalloc and hoard library functions. Although, we have changed our application, we did not want to change STL and other libraries. The figure in 1.6 shows that *hoard* is the best memory allocator for MT applications.
- **Effect of Granularity** This test was performed to know the number of cavities which should be assigned to each thread(See figure 1.7). On quad CPU machine, the first set contains the execution time using 1-5 cavities per thread, and the second set shows the execution time using 8 threads and assigning 1-5 cavities per thread.

1.3 Fine Grained Multithreading (FGM) Approach

With the FGM approach, parallelization is exploited at the level of cavity creation. It is based on the observation that in Bowyer-Watson DT algorithm, numerically, the most expensive computation is the cavity identification and refilling. Cavity identification step involves searching the triangle rooted at a given element $\mathcal{E}_s$ that violate the delaunay properties. Because of the local properties of the DT algorithm, searching is carried out using depth first search (DFS) or breadth first search (BFS). Both BFS and DFS searching can be efficiently parallelized in the presence of efficient hardware or software multithreading support.

- *Factory Class* In a given DT algorithm, for each newly inserted vertices, a new cavity has to be formed and retriangulated, therefore, a pool of cavities are instantiated at the beginning and reused. The scheduling of threads and context switching are expensive with standard Posix thread library. therefore, a customized Factory class have been used which manages scheduling of the task transparently to the users.
- *Spin locks* In the present implementation, at least six locks to protect the shared data. We use efficient *spin locks* instead of standard Posix mutex.
- *Multiprocessor memory allocator* Standard memory allocators require locks which are expensive. In the present algorithm, each thread allocate memory during the search operation, therefore we used *Hoard* library for memory allocation purpose.

```
 Cavity :: expansion ( Triangle *source, Point *newpoint )
{
   if( !cavity_expansion_factory.initialized() )
        cavity_expansion_factory =  create_new_factory();
        cavity_expansion_factory->start_working();

    cavity_expansion_factory->wakeup();
    factory_breadth_first_search(source, newpoint);
    cavity_expansion_factory->sleep();
 }

 Cavity :: factory_atomic_work_bfs( Triangle *tri, int side,
                                    cavity_expansion *parent, Point *apexCoords )
 {
    Triangle *neighbor = tri->neis[side];

    if(neighbor) {
        if (!isSelected(neighbor)) {
            if( isRejected(neighbor) ) {
                addItem(tri, neighbor, side, apexCoords );
            } else {
                double ori = in_circle(neighbor->points[0]->coord.xy,
                                       neighbor->points[1]->coord.xy,
                                       neighbor->points[2]->coord.xy,
                                       apexCoords);
                if(ori >= 0) {
                    setSelected(neighbor);
```

```
                    cavity_expansion *child = new cavity_expansion(this, neighbor, paren
                                                                   apexCoords);
                    cavity_expansion_factory->add_work( child );
                } else {
                    setRejected( neighbor );
                    addItem(tri, neighbor, side, apexCoords );
                }
            }
        }
    } else {
        addItem(tri, neighbor, side, apexCoords);
    }
}

Cavity :: factory_breadth_first_search( Triangle *source, Point *p)
{
    apexCoords = p;
    setSelected(source);
    cavity_expansion *root = new cavity_expansion(this, base, NULL, xy, areaBound, angle
    cavity_expansion_factory->add_work( root );
    cavity_expansion_factory->parent_barrier( root );
}
```

With both BFS and DFS searching methods, it is likely that a single element may be reached by two different threads. If we do not allow two threads to access the elements then we need to protect the element with efficient mutex and we do, then when all threads have completed their searching, we have to remove the duplicate elements to finalize the cavity. The performance of the first method depends on the cost of mutexes whereas the performance of later depends on the cost of sorting the element.

Once the cavity expansion start, the factory is woken up and become ready to accept work. The factory accepts work unit which is an object and include all the work performed by the thread. Each work unit can create more work and add to the factory object. This way a factory work unit are organized into tree structure. The factory does not return until all the work from its children is complete.

The performance of the above module depends on mutexes which protect the data and the memory allocation for the new work unit. In *isSelected, isRejected, addItem, setSelected* and *setRejected* protect the data using locks.

1.4 Conclusions

Optimistic Concurrency Control is well suited for mesh generation algorithm such as Bowyer Watson incremental algorithm. The experiments show that this method is scalable, reliable and most importantly requires minor modifications in the existing sequential code. Efficient implementation of memory allocators, thread safe STL and good heuristic are essential for obtaining good performance of this algorithm in multithreaded environments.

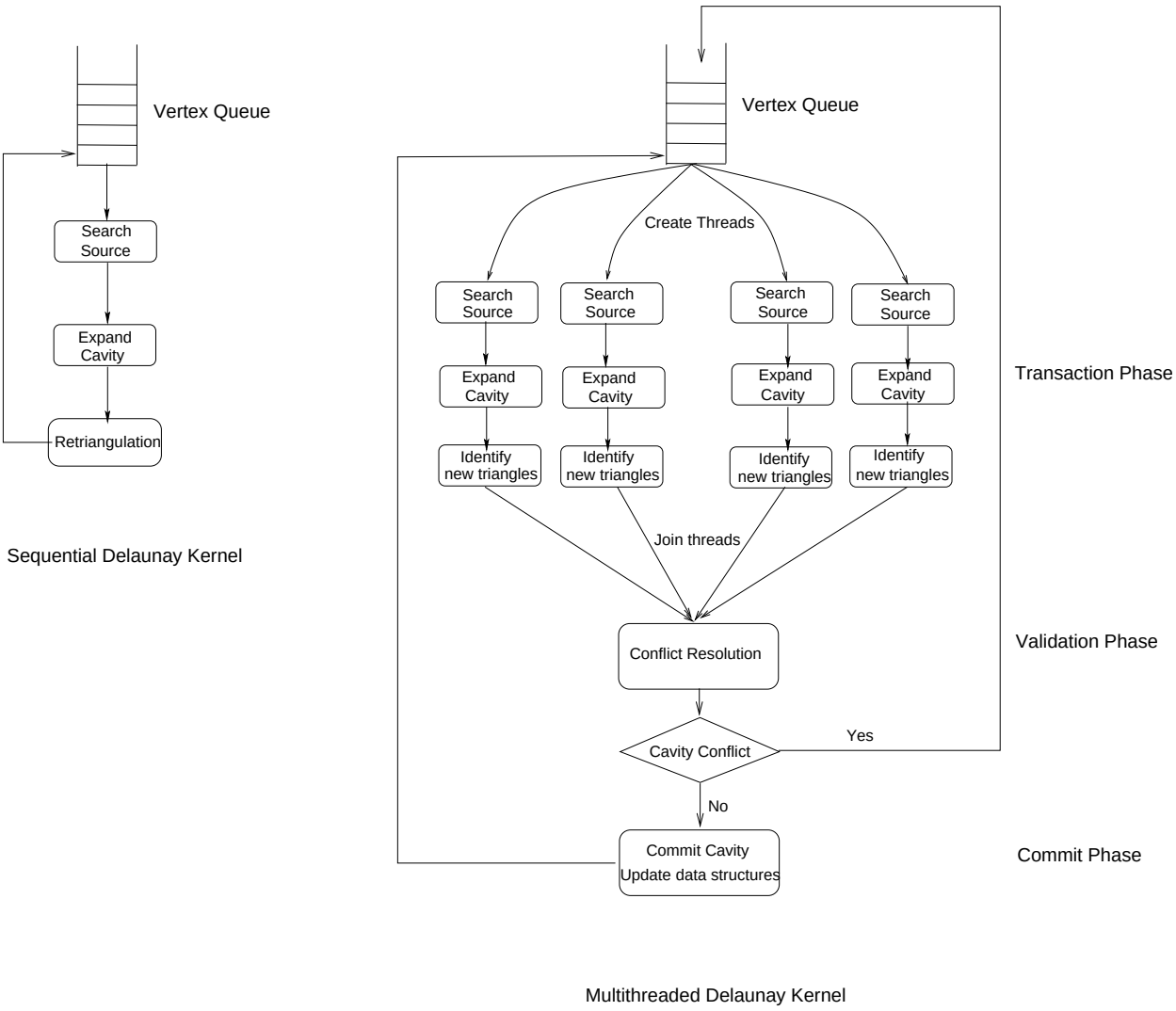

Figure 1.2: Flow chart of Delaunay triangulation

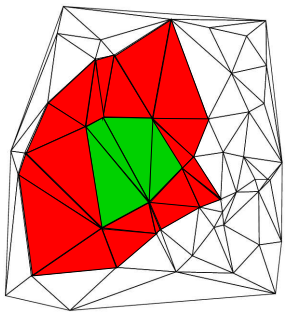

Figure 1.3: Conflict region in Parallel Delaunay Triangulation

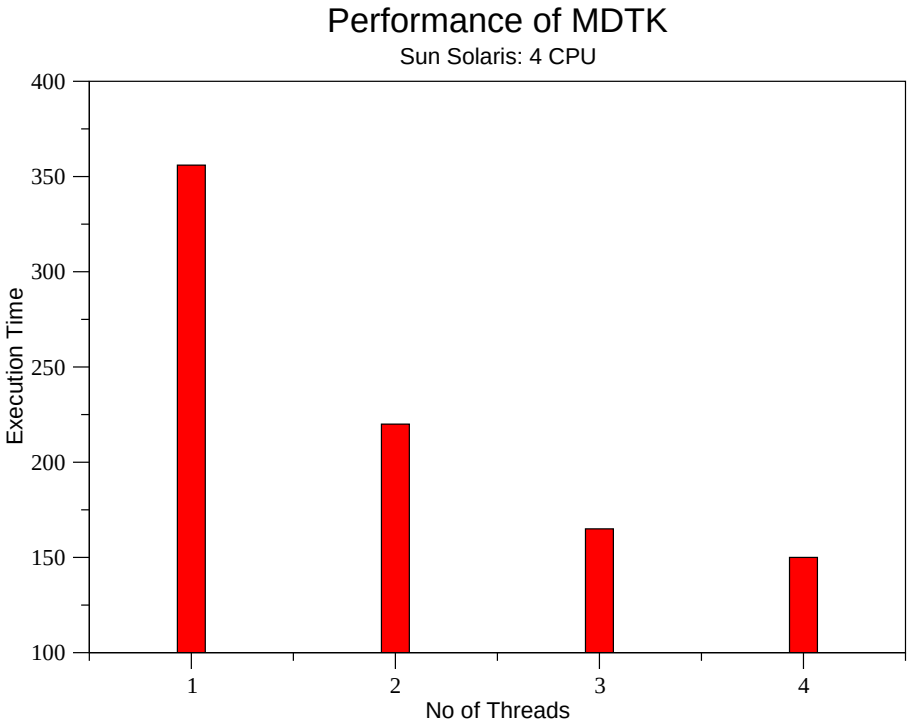

Figure 1.4: Performance of MDTK on Solaris

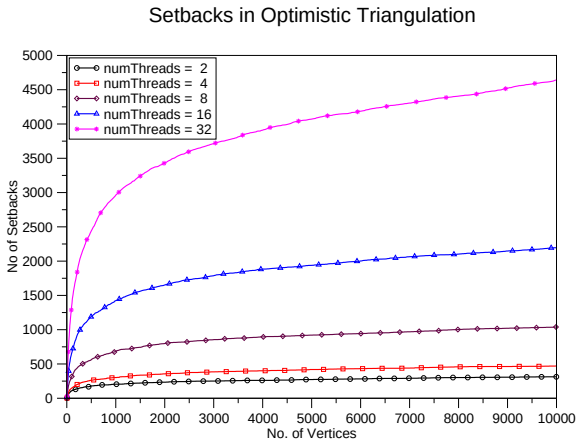

Figure 1.5: Conflicts grows as number of threads increases

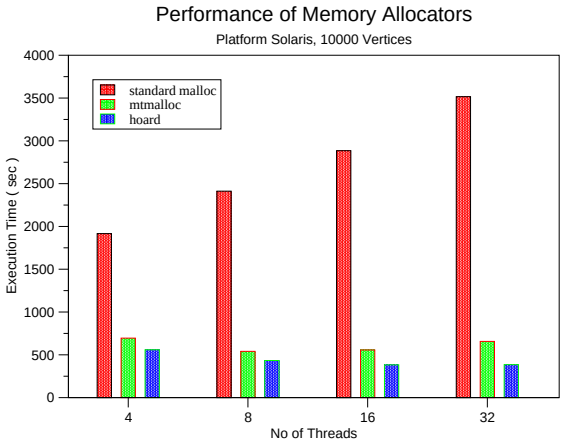

Figure 1.6: Performance of Memory allocators on Solaris

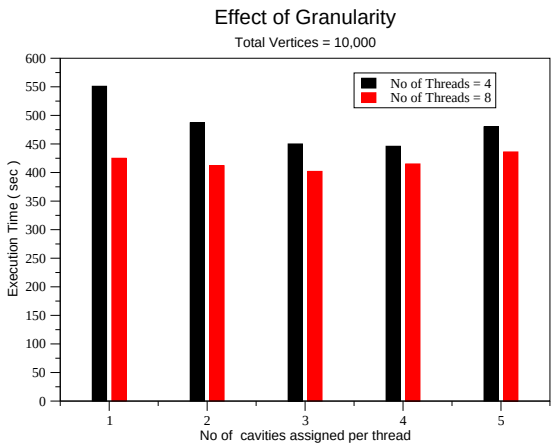

Figure 1.7: The execution time depends on number of cavities assigned to each thread

Bibliography

[1] Nikos Chrisochoides and Demian Nave. Parallel delunay mesh generation kernel. *Int. J. Num. Methods in Engineering*, 58(No 2):161–176, 2003.

[2] Robert D. Blumofe Emery D. Berger. Hoard: A fast, scalable and memory-efficient allocator for shared-memory multiprocessors. Technical report, University of Texas at Austin, September 1999.

[3] H. T. Kung and John T. Robinson. On optimistic methods for concurrency control. *ACM Trans. Database Syst.*, 6(2):213–226, 1981.

[4] Jonathan Richard Shewchuk. *Delaunay Refinement Algorithms For Triangular Mesh Generation*. PhD thesis, Department of Electrical Engineering and Computer Science, May 2001.