Efficient Homology Computations on Multicore and Manycore Systems

N. Anurag Murty[1], Vijay Natarajan[1,2], Sathish Vadhiyar[1]
[1]Supercomputer Education and Research Centre
[2]Department of Computer Science and Automation
Indian Institute of Science, Bangalore, India
anrgmrty@gmail.com, vijayn@csa.iisc.ernet.in, vss@serc.iisc.in

***Abstract*—Homology computations form an important step in topological data analysis that helps to identify connected components, holes, and voids in multi-dimensional data. Our work focuses on algorithms for homology computations of large simplicial complexes on multicore machines and on GPUs. This paper presents two parallel algorithms to compute homology. A core component of both algorithms is the algebraic reduction of a cell with respect to one of its faces while preserving the homology of the original simplicial complex. The first algorithm is a parallel version of an existing sequential implementation using OpenMP. The algorithm processes and reduces cells within each partition of the complex in parallel while minimizing sequential reductions on the partition boundaries. Cache misses are reduced by ensuring data locality for data in the same partition. We observe a linear speedup on algebraic reductions and an overall speedup of up to 4.9× with 16 cores over sequential reductions. The second algorithm is based on a novel approach for homology computations on manycore/GPU architectures. This GPU algorithm is memory efficient and capable of extremely fast computation of homology for simplicial complexes with millions of simplices. We observe up to 40× speedup in runtime over sequential reductions and up to 4.5× speedup over REDHOM library, which includes the sequential algebraic reductions together with other advanced homology engines supported in the software.**

I. Introduction

Topology is the study of the connectivity of space and provides useful tools for analyzing datasets by enabling the abstract representation of features in the data. Topological data analysis finds numerous applications in neuroscience, astrophysics, image analysis, and nonlinear dynamics [1–6]. All of these applications are characterised by very large data sizes from which topological data analysis reveals underlying patterns and structure. This structure is extracted in the form of connected components, holes, and voids of higher dimensions along which the data aligns itself in space. The characterization of these connected components, holes, voids, and their higher dimensional equivalents is more formally described by the notion of homology. Computing homology requires the construction of a combinatorial representation of the space such as a simplicial complex.

An interesting application of homology computations is the detection of holes in the coverage of a sensor network[7]. Hole detection is useful in cell-phone communications, beacon navigation and some problems in security and defense. These type of applications require real-time computation of homology. The requirement for real-time computations and increasingly large datasets highlight the need for fast and memory-efficient algorithms for homology computations. This serves as our primary motivation for developing parallel algorithms for homology computations.

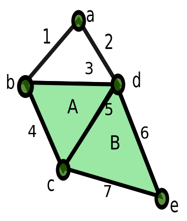
We present parallelization strategies for fast computation of homology on multicore and manycore GPU systems. The algorithm we consider for parallelization uses the method of algebraic reductions to reduce the size of the input space while maintaining its homology[8]. For implementation on multicore architectures, the algebraic reduction step in REDHOM is parallelized using OpenMP[9]. We decompose the complex and perform parallel reductions on the different partitions while keeping the boundaries between partitions intact. The next step involves algebraic reduction of the unreduced boundary cells sequentially to compute homology. We obtain up to 4.9× improvements in performance over sequential algebraic reductions.

The above idea does not scale well for higher degrees of parallelism as in the case of GPU architectures. So we describe a different algorithm amenable to massively parallel architectures. Each GPU thread attempts to perform an algebraic reduction but this is possible only when certain conditions involving its neighbours are met. Moreover, it is observed that construction of the entire simplicial complex is not necessary for performing algebraic reductions. This observation speeds up homology computations and leads to a memory-efficient algorithm. Finally, we define a cost function that enables us to perform reductions in a load-balanced manner. Implementation of this algorithm gives up to 40× speedup over sequential algebraic reductions. We also obtain up to 4.5× speedup over REDHOM, which implements among the fastest sequential algorithms for homology computations.

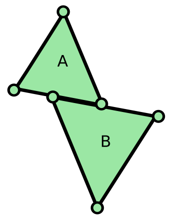
Primary contributions of this work are:

1) A multicore algorithm for fast homology computations.
2) Modifications to sequential algebraic reductions using OpenMP that improve the performance by up to 4.9×.
3) A memory-efficient GPU algorithm based on algebraic reductions that gives up to 40x speedup over the sequential algorithm and up to 4.5× speedup over homology computations in REDHOM library.
4) A novel cost assignment scheme to ensure load-balanced execution and to ensure that only low-cost reductions are performed in a given iteration.

978-1-4799-0730-4/13/$31.00 ©2013 IEEE



(a)



(b)

Fig. 1. (a) A valid simplicial complex of dimension 2 (b) An invalid simplicial complex since A and B do not intersect on an edge or a vertex.

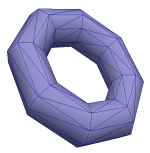


Fig. 2. The torus has one connected component, two tunnels and one void

Although we focus on computing the rank of homology groups in our description, the algorithms presented could optionally be used to output an incidence matrix of the reduced complex. The Smith normal form algorithm can be applied to compute torsion coefficients also. This extension is relevant only in complexes of dimensions higher than three.

Section II provides the required background and definitions especially focusing on algebraic reductions. Section III is a literature survey of prior research in this area. Sections IV and V provide detailed descriptions of the proposed algorithms for homology computations on multicore systems and GPUs respectively. Experimental results are presented in Section VI and Section VII presents possible directions for future research.

II. Background

In topology, we study the properties of spaces that are invariant under continuous deformations or more formally, homeomorphisms. A finite representation of topological spaces is required to compute these topological invariants. An example of such a finite representation is a simplicial complex. We present below a few definitions that are required to describe our methodology. For a more mathematical treatment, we refer the reader to the texts by Zomorodian[10] and Munkres[11].

A. Simplicial complexes and simplicial homology

A k-$simplex$ σ is the convex hull of a set A of $k+1$ independent points in $\mathbb{R}^d$, for $0 \leq k \leq d$. We use the terms $vertex$ for 0-simplex, $edge$ for 1-simplex, $triangle$ for 2-simplex and $tetrahedron$ for 3-simplex. A simplex σ' is a $face$ of a simplex σ if σ' is contained in σ. A $simplicial\ complex$, K, is a finite set of simplices satisfying two properties : (i) if $\sigma \in K$ and τ is a face of σ then $\tau \in K$ and (ii) if $\sigma \in K$ and $\sigma' \in K$, then $\sigma \cap \sigma'$ is either ϕ or a face of both σ and σ'. The dimension of K, $d(K)$ is defined as the maximum dimension of a simplex in K. Fig 1(a) shows a valid simplicial complex whereas the collection of simplices in Fig 1(b) does not satisfy property (ii) and is thus not a simplicial complex.

A k-simplex σ can be represented as the set of its vertices $[v_0, v_1, \ldots, v_{k-1}]$. For instance, a triangle with vertices A, B, and C can be represented as $[A, B, C]$. The boundary of a k-simplex is formed by the $(k-1)$-simplices bounding it. In our example, the boundary of $[A, B, C]$ are the three edges $[A, B]$, $[B, C]$ and $[C, A]$. The $boundary$ of a k-simplex $\sigma = [v_0, v_1, \ldots, v_{k-1}]$ is defined as the formal sum $\partial\sigma = \sum_i -(1)^i[v_0, v_1, \ldots, \hat{v}_i, \ldots, v_{k-1}]$. A minus sign in this sum basically means including the same simplex but with the opposite orientation i.e., with any two of the vertices interchanged. Simplices with opposite orientations cancel each other out. The coboundary of a k-$simplex$ σ' is the set of all $k+1$-$simplices$ that have σ' as a face. If a simplex σ' lies in the boundary of σ, then σ' lies in the $coboundary$ of σ'. For instance, in Figure 1(a), $\partial A = 3+4+5$ and $\partial B = 5+6+7$. The coboundary of edges 1 and 2 is $\{\phi\}$. The coboundary of edges 3 and 4 is $\{A\}$, and that of 6 and 7 have coboundary $\{B\}$. Edge 5 has $\{A, B\}$ as its coboundary.

A fundamental property of boundaries is that the boundary function applied twice is zero. In the above example, $\partial\partial[A, B, C] = \partial([B, C] + [C, A] + [A, B]) = [B] - [C] + [C] - [A] + [A] - [B] = 0$. We define a k-cycle as any formal sum of simplices whose boundary is zero. Due to property of boundaries, all boundaries are cycles. However, not all cycles bound a higher dimensional simplex. For instance, if our original simplicial complex had the edges $[B, C]$, $[C, D]$ and $[D, B]$ but not the triangle $[B, C, D]$, the boundary of $[B, C] + [C, D] + [D, B]$ is $\partial([B, C] + [C, D] + [D, B]) = [B] - [C] + [C] - [D] + [D] - [B] = 0$. The edges $[B, C]$, $[C, D]$ and $[D, B]$ form a cycle that is not a boundary of any triangle.

The $homology$ of a simplicial complex deals with counting the number of independent cycles that do not bound any set of simplices in a higher dimension. The homology in orders 0, 1 and 2 represent the number of connected components, tunnels, and voids respectively, and are represented as algebraic groups. In this paper, we are interested in computing the rank of these groups and we refer to these computations as $homology\ computations$. For example, homology computations identify one connected component, two independent tunnels and one void in the simplicial complex that represents a torus in Figure 2. For ease of description, computations are performed modulo 2 which gives us the $\mathbb{Z}_2$ homology[10].

B. Algebraic reduction

Consider the simplicial complex in Figure 1(a) . It consists of one connected component and contains one tunnel. Clearly, we can construct a smaller sized complex representing one component and containing one tunnel. Reduction algorithms reduce the size of a simplicial complex in a way such that homology remains unchanged.

We focus on algebraic reductions to reduce the size of the complex[8]. Initially, each dimension d of the simplicial complex consists of the set of all the d-simplices. During the reduction procedure, d-simplices can merge to form d-*cells*, which can be thought of as more general versions of simplices. For example, vertices are 0-cells, edges are 1-cells, polygons are 2-cells and 3-D polytopes are 3-cells. In any intermediate step of the procedure, dimension d consists of the set of all the d-cells.

For two cells u,v of the same dimension, we define $\langle u, v\rangle$ to be 1 when $u = v$ and 0 otherwise. After the algebraic reduction of cell b of dimension m with respect to its face a in dimension m-1, the new boundary maps are given by Equation 1, where addition is performed modulo 2.

$$\partial v = \begin{cases} \partial v, & \text{if } d(v) \notin \{m, m+1\}, \\ \partial v + \langle \partial v, a\rangle \partial b, & \text{if } d(v) = m, \\ \partial v + \langle \partial v, b\rangle b, & \text{if } d(v) = m+1. \end{cases} \tag{1}$$

After the reduction, b and a are removed from the complex. This reduction operation is guaranteed to preserve the homology of the complex. In the end, the number of cells in dimension d that are irreducible is equal to the homology of order d.

Figure 3 illustrates an example algebraic reduction. The input is a simplicial complex with one connected component and one tunnel. Initially, the cells in various dimensions just consist of the simplices. The cells are as follows:
$dimension - 0 : \{a, b, c, d, e\}$
$dimension - 1 : \{1, 2, 3, 4, 5, 6, 7\}$
$dimension - 2 : \{A, B\}$
First, face B is reduced with respect to edge 5. After this reduction, face A no longer remains a simplex. It becomes a cell with boundary $\{3, 4, 6, 7\}$. Subsequent reductions reduce face A with respect to edge 7, and the edges $6, 2$, and 1 with respect to the vertices e, d, and b respectively. In each of these steps, the homology of the initial simplicial complex is preserved. Finally, the edge 3 and vertex a remain. There are no remaining cells of dimension 2. This implies that the homology of order 2 is zero. In dimension 1, edge 3 is irreducible as it is incident on vertex a twice which means that it has no boundary. So homology of order 1 is one. Similarly, the vertex a is irreducible as a 0-dimensional cell has a zero boundary. Thus the homology of order 0 is also one.

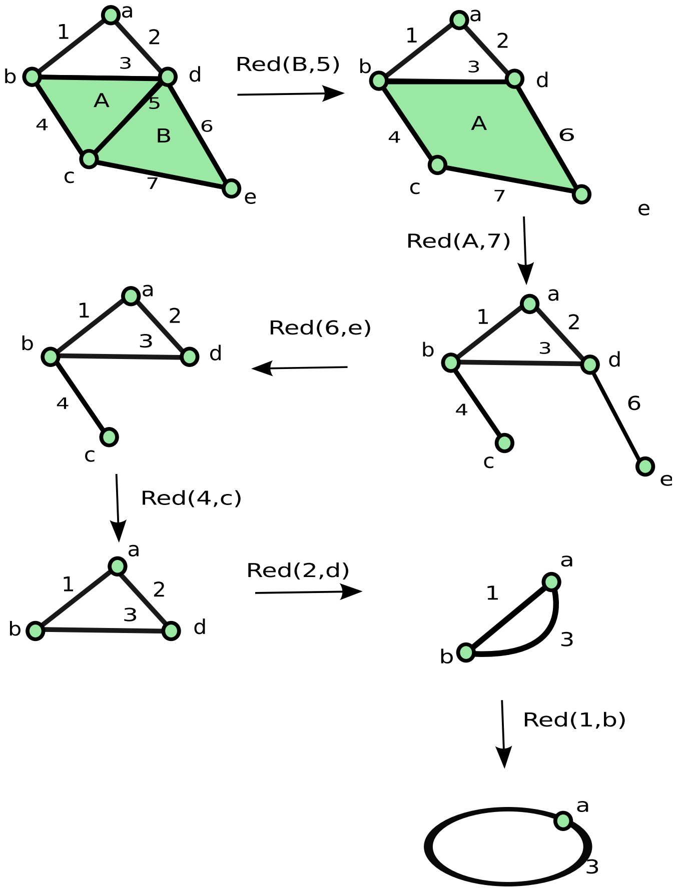


Fig. 3. Step-by-step algebraic reductions of a simplicial complex [Red(b, a) means the reduction of cell b with respect to face a]

III. Related Work

Homology computations via the classical method given by Munkres[11] has exponential bounds. The Smith normal form used to compute homology requires a polynomial number of steps but the numbers in the intermediate computations can get very large thus yielding exponential bounds. Kannan et al.[12] gave the first polynomial time algorithm to compute the Smith normal form of matrices. A probabilistic analysis based on the fact that the boundary matrices are sparse further improved the expected running time to O(n^2)[13]. The quadratic complexity is undesirable for very large datasets.

For finite simplicial complexes embedded in $\mathbb{R}^3$, Delfinado et al.[14] describe a near linear time algorithm to compute homology. However, this algorithm does not extend to complexes in dimensions greater than three.

Another class of algorithms are the algebraic reduction algorithms that reduce the size of the complex while maintaining its homology. Kaczynski et al.[8] propose a reduction algorithm to compute homology of a finitely generated chain complex. This is the first in a number of algorithms based on algebraic and geometric reductions which are implemented in REDHOM, a software library for efficient computation of homology of sets[15]. REDHOM includes algorithms for computing homology based on coreductions[16], acyclic subspace methods[17] and discrete Morse theory[18]. All these methods postpone the actual homology computations using Smith normal form until the complex size is much smaller. The implementation in REDHOM is sequential and there is scope for parallelization in many of its algorithms.

Lewis et al.[19] have implemented a framework for parallel computation of homology on multicore computers by dividing an input complex into local pieces and performing parallel computations on these. After the parallel computations, the pieces are merged and homology is calculated again to give the final result. The method relies on the property of the initial division that the homology is equal to the sum of homology of the individual pieces. However, this method does not scale to the level of parallelism offered by manycore GPU platforms, an issue we address in our work.

IV. Homology Computations on Multicore Systems

We now propose two approaches to parallelizing the homology computation algorithm. The first approach is suitable for multicore computations and is based on the sequential algorithm implemented in the REDHOM library[15]. The library has efficient implementations of algorithms based on reduction methods such as acyclic subspace construction, elementary reductions, and discrete Morse theory. Each of these techniques is applied in sequence on the input complex to reduce its size and hence compute the homology efficiently. In this work, our focus is only on parallelizing algebraic reductions. We disable all the other steps of REDHOM and restrict our attention to algebraic reductions.

First, we discuss the steps of the sequential algebraic reductions which are profiled for various datasets in Figure 4. We discuss the datasets mentioned in the figure in more detail in Section VI.

A. *Sequential algorithm for algebraic reductions*

Read and construct simplicial complex. In this step, maximally induced simplices of a simplicial complex are taken as input and the simplices of all dimensions are generated. This step forms the pre-processing step for all the algorithms implemented in REDHOM, including algebraic reductions. As seen in Figure 4, this step has a very low contribution to the execution times of sequential algebraic reductions.

Codes assignment and construction of reducible complex. The simplicial complex constructed in the previous step has to be algebraically reduced for homology computations using Equation 1. Since each step of a reduction modifies the boundaries and coboundaries of simplices, we need a data structure that provides fast access to boundary and coboundary data. For the purpose of creating a map from simplices to their boundaries and coboundaries, integer codes are assigned to all the simplices. Then boundary and coboundary maps which assign a chain to each code are constructed. This set of maps constitutes a reducible complex on which algebraic reductions are performed. This step takes up the highest percentage of the total execution time.

Algebraic reductions. This step performs the actual reductions on the reducible complex that represents the input simplicial complex. Starting from the highest dimension, the cells are reduced with respect to their faces and their boundary maps are modified. For each dimension, the number of remaining irreducible simplices is the homology of that dimension. The modification of these boundaries and coboundaries to

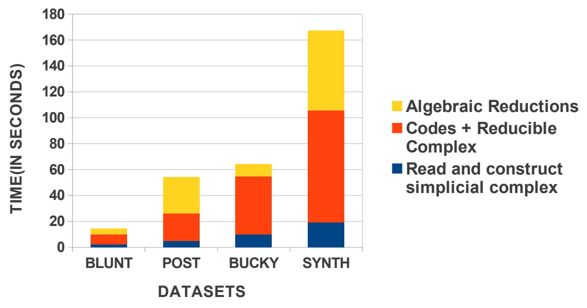


Fig. 4. Timings for various functions in homology computations using sequential algebraic reductions

compute homology is also a time consuming step in sequential reductions.

Algorithm 1 Algorithm For Multicore Homology

```
Input: Maximal simplices of simplicial complex $K$
Output: Homology : $\beta[0]$,$\beta[1]$,...,$\beta[d(K)]$
1: Partition the simplicial complex $K$ ($P_0$,$P_1$,...,$P_{k-1}$)
2: Mark boundary vertices
3: Spawn k threads and assign thread t to $P_t$
4: (In Parallel) Threads construct reducible complex for their partition
5: (In Parallel) Threads reduce non-boundary cells in their partition
6: Merge unreduced partitions to get a single reducible chain complex $K'$
7: Perform algebraic reductions on all reducible cells of $K'$
8: $\beta[d]$ is cardinality of irreducible cells in dimension d
   return $\beta$
```

B. *Multicore algorithm for algebraic reductions*

We attempt to parallelize the construction of the reducible complex and the algebraic reductions as both of these are the major contributors to the execution time of the homology computations using algebraic reductions. Algorithm 1 explains the steps for computation of homology on multicore machines. The steps are : decomposition of the complex into partitions, parallel reductions of these partitions, merging of the reduced partitions and sequential reduction of the merged complex.

Decomposition into partitions. Parallelization depends on an initial partition of the input mesh into near-equal sized meshes whose boundaries are as small as possible. These partitions are generated using METIS, a graph partitioning software[20]. Minimizing the number of boundary cells helps in reducing the time spent in the sequential part of our algorithm while similar sized partitions help in maintaining load balance during the parallel phase. Simplices with all their vertices occurring in two or more partitions are marked as boundary simplices. During the serial read phase, we preallocate contiguous memory for non-boundary simplices from each partitions. This ensures spatial locality of simplices from the same partition.

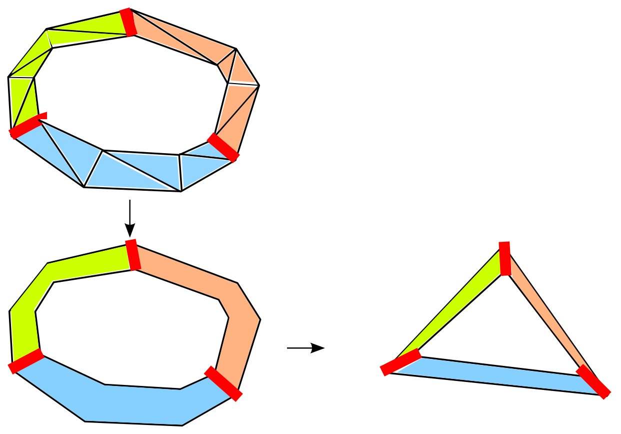


Fig. 5. Intermediate steps in reductions of the partitions by different threads. Different colours represent the partitions reduced by the threads. The boundary elements are shown in red and are not reduced in the parallel phase.

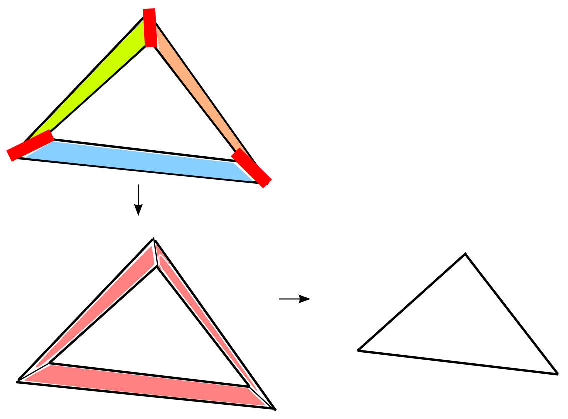


Fig. 6. Intermediate steps of the sequential reduction phase. The partitions are merged in this case and sequential reductions are performed subsequently.

Also, a separate memory space is allocated beforehand for the boundary simplices.

Parallel construction of the reducible complex. After boundary vertices are marked, we spawn one thread per partition. Firstly, the threads split the simplicial complex obtained from the read operation to construct one reducible complex per partition. We allocate memory pools for each partition and ensure that simplices which belong to the same partition are assigned to the same memory pool. As data locality is maintained for the non-boundary simplices of a partition, cache misses incurred in the iterations over these simplices to construct the reducible complex in parallel are greatly reduced. This step is crucial in improving the performance of the multicore algorithm.

For the set of boundary simplices, all the threads have to iterate over the same memory space but this does not greatly affect performance due to the small size of the boundary compared to the size of the partitions.

The reducible complex represents the boundary and coboundary information of the partition assigned to the thread. Codes are assigned to cells while ensuring that boundary cells, i.e., cells with all their vertices on the boundary, are assigned the same code over all partitions. Using METIS for graph partitioning ensures that we obtain balanced partitions of the mesh, thus keeping these parallel constructions load-balanced.

Parallel algebraic reductions. Algebraic reductions are then performed in parallel on each of these reducible complexes. The reductions are done on all cells with the exception of boundary cells. Boundary cells are shared by two or more partitions and are not reduced in the parallel phase. Our partitions should thus have very small boundary sizes to maximize the parallel reductions. Figure 5 shows some of the intermediate steps in the parallel reduction phase. The threads reduce the different coloured partitions in parallel and leave the boundaries unreduced.

Merge and sequential reductions. Now each thread has a partially reduced chain complex consisting primarily of unreduced boundary cells. All of these complexes are then merged together to form a new reducible complex. As reduction operations preserve homology, the homology of this new complex is the same as the homology of the input mesh. Algebraic reductions are applied to all reducible cells of the merged chain complex. After this step is completed, the homology in each dimension is given by the number of irreducible cells of that dimension. Figure 6 shows the different partitions being merged after which the complex is reduced sequentially.

V. Homology Computations on Manycore/GPU Systems

As mentioned, the methodology used for multicore homology cannot be directly extended to GPU architectures. We propose an algorithm to compute homology on GPUs. The discussion considers $\mathbb{Z}_2$ homology i.e., addition modulo 2, but can be easily extended to arbitrary fields, albeit with increased space requirements. We assume that the input is in the form of maximal simplices of a simplicial complex and is stored as a set of vertex arrays in the GPU global memory. Algorithm 2 is the homology computation algorithm for GPUs. The important steps in the algorithm are explained below.

Reducing memory requirements for GPU algorithm. In Equation 1, we observe that reducing a cell in dimension m can only modify the boundaries in dimensions m and $m+1$. So, if we start from the highest dimension and work our way downwards, the boundaries in only the highest dimension are modified[8]. An algebraic reduction of a cell in dimension m is performed with respect to one of its faces in dimension $m-1$. Thus, given the list of all cells we can generate all the faces with which the cells can be paired and reduced. This implies that we just need to transfer the list of simplices of the highest unreduced dimension m to perform algebraic reductions. This crucial observation helps in improving performance of the algorithm as all the intermediate data structures are generated on the GPU so that data transfers between the host and device are minimized. Intermediate data structures include the boundary data and the coboundary data of the cells and faces respectively. For lower dimensions, we only need to carry forward the unreduced faces from this dimension. In comparison to the space required for storing the entire simplicial complex, GPU memory requirements are very low when we adopt this approach of constructing the complex per-dimension starting from the highest dimension. In contrast with this, the entire simplicial complex with simplices in all dimensions is constructed in REDHOM as a pre-processing step.

Algorithm 2 Algorithm For GPU Homology

```
Input: Maximal simplices of simplicial complex $K$
Output: Homology : $\beta[0]$,$\beta[1]$,...,$\beta[d(K)]$
1: for $dim$ = d($K$) downto 1 do
2:     Transfer cells of dimension $dim$ to GPU(struct-cells)
3:     Allocate space for faces on GPU(struct-faces)
4:     $\beta[dim]$ = Reduce-dimension(struct-cells,struct-faces)
5:     Merge unreduced faces in struct-faces with cells of
         dimension $dim - 1$
6: end for
7: return $\beta$
```

Data structure for reductions. Algebraic reductions of a cell-face pair require a data structure which allows fast access to cell boundaries and face coboundaries[1]. In REDHOM, this reducible chain complex is generated from the simplicial complex[15]. For our purposes, we never construct the entire simplicial complex. In each iteration in Algorithm 2, we only transfer the unreduced simplices of the highest unreduced dimension to the GPU.

On the device, the reducible complex is generated in the procedure given in Algorithm 3. Initially, all the cells to be reduced are simplices. So, it is straightforward to generate the faces. It is trivial for cells to access the faces on their boundaries and also to compute the boundary-sizes which are equal in the beginning. However, the list of faces has many repeated faces that belong to the boundary of more than one cell. A sort procedure, say lexicographical sort, helps to collect the repeated faces and in obtaining their coboundary cells and coboundary sizes. The indices of the faces change after the sorting step. So, a remapping from old face indices to new ones is performed on the cell boundaries. After this step, all cells and faces are organised into a data structure that enables efficient algebraic reductions, by supporting fast access to cell boundaries and face coboundaries. Cells can access their boundary by using the new maps and faces can directly access their coboundary by using the values of their old locations. For instance, if the face generated from a tetrahedron had an initial index i, it is straightforward to see that the index of the parent tetrahedron of this face is $\lfloor i/4 \rfloor$. Initially we ensure that the boundary face indices and coboundary cell indices are stored in sorted order. This helps in ensuring that the symmetric difference operation carried out for algebraic reductions can be executed in time linear in the size of the boundary/coboundary.

Algorithm 3 Procedure Reduce-dimension

```
1: Reduce-dimension(struct-cells,struct-faces){
2: (GPU) Generate faces from cells
3: (GPU) Assign values to boundary, boundary-size vectors
4: (GPU) Sort faces in lexicographic order and mark repeated
     faces
5: (GPU) Assign values to coboundary, coboundary-size vec-
     tors
6: (GPU) Remap to get newIDs in boundary vectors
7: (GPU) Remove repeats from face vectors
8: Initialize variables $irreducible, reduced$ to 0
9: while ($irreducible + reduced < number\_of\_cells$) do
10:     (GPU) Each cell finds face with min. cost of reduction
11:     (GPU) Cells with min. costs within fixed margin do a
          $race\text{-}prioritycheck\text{-}check$ to lock required
          boundaries and coboundaries
12:     (GPU) Invoke Kernel Reduce-pair
13:     (GPU) Update values oF $reduced$ and $irreducible$
14: end while
15: return $irreducible$
16: }
```

Cost of a reduce-pair operation. When reducing the cells of a particular dimension, only one of the many faces of a cell has to be chosen for reduction. In addition to this, not all cells can be reduced simultaneously. We introduce a novel cost function that helps in choosing unique reduction pairs without conflicts. We only allow reduction pairs with costs within a small margin of the smallest cost to proceed with reductions in a given iteration. As cost of a reduction pair reflects the time taken for that particular cell-face reduction, avoiding high reduction costs ensures that we limit the time spent in a particular iteration.

To derive the cost function and to describe the intuition behind its design, let us consider Equation 1 again. As reductions are performed on the highest unreduced dimension, only the boundaries of dimension m are modified. The value $\langle \partial v, a \rangle$ is non-zero only if v is on the coboundary of face a. So, reduction with respect to a only modifies the boundaries of cells on the coboundary of a. Similarly, if cell b is reduced, only the coboundaries of faces on its boundary are modified.

When working with $\mathbb{Z}_2$ homology, the boundaries and coboundaries are essentially sets of faces and cells respectively. For cell a, $Cbdy(a)$ and $Bdy(a)$ are used to denote the set of cells in the coboundary and boundary respectively. $\#Cbdy(a)$ and $\#Bdy(a)$ are the cardinality of these sets. For $\mathbb{Z}_2$ homology, merging two boundaries or coboundaries is equivalent to a symmetric set difference operation, as described in Algorithm 4. We define the cost of a pair reduction as the work done in performing the symmetric set difference operations. As the order of complexity of computing set difference of two sorted arrays is linear in the sum of number of elements in these arrays, the cost of reducing cell b with face a is:

$$\begin{aligned} reduction\_cost(a,b) = (\#Bdy(b)-1)\times(\#Cbdy(a)) \\ + \sum_{g\in Bdy(b)\setminus\{a\}} (\#Cbdy(g)) \\ +(\#Cbdy(a)-1)\times(\#Bdy(b)) \\ + \sum_{u\in Cbdy(a)\setminus\{b\}} (\#Bdy(u)) \end{aligned} \tag{2}$$

In Algorithm 3, the cost function is used to find the face with minimum cost of reduction for each face. This cost helps in deciding the maximum cost we are willing to incur for reductions in a particular iteration.

Algorithm 4 Kernel Reduce-pair

```
1: Reduce-pair(cell $b$,face $a$){
2: for all cells $t$ on coboundary of $a$ except $b$ do
3:     $Bdy(t)=(Bdy(t)\cup Bdy(b))\setminus(Bdy(t)\cap Bdy(b))$
4:     if $(\#Bdy(t) == 0)$ then
5:         Mark $t$ as irreducible
6:     end if
7: end for
8: for all faces $f$ on boundary of $b$ except $a$ do
9:     $Cbdy(f)=(Cbdy(f)\cup Cbdy(a))\setminus(Cbdy(f)\cap Cbdy(a))$
10: end for
11: Mark $b, a$ as reduced
12: }
```

Race-prioritycheck-check and homology computations. The reduction of a cell-face pair needs to modify certain boundaries and coboundaries as described in Algorithm 4. Thus we need to ensure that these boundaries and coboundaries are not modified by more than one thread performing a cell-face reduction. We use the three-phase *race-prioritycheck-check* technique to ensure that modification of a particular boundary/coboundary is done only by a single thread[21]. In the *race* step, threads assigned to each unreduced cell use their IDs to lock the required boundaries and coboundaries. In the priority check step, all threads read the lock ID value of the boundaries and coboundaries and modify the lock value, if and only if they have a higher priority than the ID assigned in the *race* phase. Finally, in the *check* phase all cells check if they have ownership of all the required boundaries and coboundaries. If the result of the check phase is TRUE, then the thread proceed with the reduction.

Assignment of priorities in the *prioritycheck* phase is not directly based on thread IDs. Priorities to reduction pairs are assigned on the basis of costs defined in Equation 2. The reduction pair with lower reduction costs is given a higher priority. Ties are broken based on a random number assigned to each reduction pair. We do not directly use thread IDs to break ties between reduction pairs with equal costs. This is because the lexicographic ordering of the cells results in a series of cascading conflicts when we break ties using

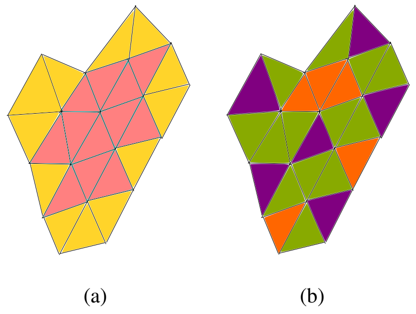
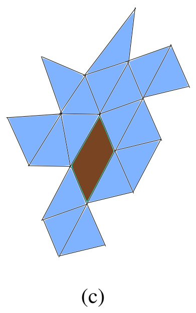



Fig. 7. Algebraic reductions in a single iteration of the GPU algorithm: (a) Yellow cells have low reduction costs and pink cells denotes high reduction costs. (b) In this iteration, purple cells will be reduced and the orange cells will not be reduced as they are unable to lock their neighbours. Green cells are the neighbours of cells being reduced. (c) The complex after the reductions in this iteration. The brown cell is formed by the reduction of the interior cell.

thread IDs. As a result, very few pairs are able to obtain the required locks and proceed with the reduction, thus reducing the number of parallel reductions in an iteration.

In some iterations, there is a possibility that the sequence of priorities is such that very few pairs are declared reducible. In the worst case, it is possible that no pairs are reducible in this iteration. In both these cases, random priorities are reassigned to reduction pairs to increase the number of parallel reductions.

Unreduced cells in dimension m are marked *irreducible* when their boundary size becomes zero. This effectively means there is no face with respect to which these cells can be reduced. When there are no reducible m-cells left, the number of irreducible m-cells is the homology of order m. After the homology for dimension m is computed, the maximal simplices of dimension $m-1$ are merged with the unreduced faces of this iteration. The same procedure is repeated until the homology of all orders has been computed.

An illustration for the GPU algorithm is shown in Figure 7. The complex in Figure 7(a) has to be reduced and one GPU thread is assigned to each cell. The minimum reduction cost of each cell with respect to any of its faces is computed on GPU using Equation 1. Yellow cells represent a minimum reduction cost of 6 with respect to their free face and pink cells have a minimum reduction cost of 20. Random priorities are assigned to each of these cells and the race-prioritycheck-check step identifies the cells which will be reduced in this iteration. In Figure 7(b), purple cells denote the cells being reduced in this iteration as they have locks on their neighbours denoted by the green cells. Orange cells are the ones which were unable

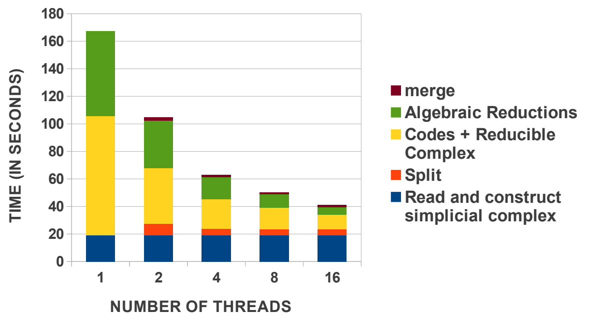


Fig. 8. Parallelization results for dataset SYNTH using multicore reductions

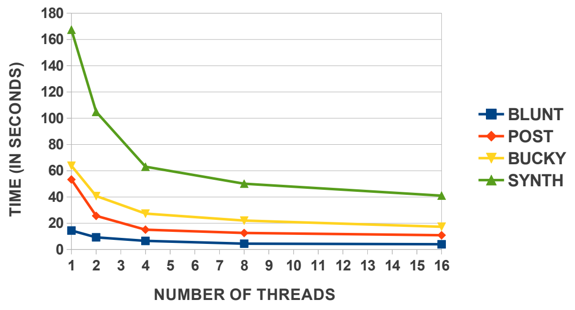


Fig. 9. Total times taken (including pre-processing step) with increasing number of cores

to lock their coboundaries and boundaries and hence will not participate in reductions in this iteration. Finally, at the end of this iteration, we have a reduced complex as shown in Figure 7(c). The brown cell in the centre is formed due to the reduction of the interior cell.

VI. EXPERIMENTS AND RESULTS

The inputs to our experiments were mostly tetrahedral meshes obtained from the aim@shape repository[22]. BLUNT represents the blunt fin dataset which consists of ~1 million simplices, POST is the liquid post dataset with ~3 million simplices and BUCKY is the buckyball dataset which ~6 million simplices), all from the aim@shape repository. SYNTH is a synthetically generated dataset with ~10 million simplices.

All the timings presented for both the multicore and the GPU algorithms include data pre-processing times as well. For the GPU algorithm, data transfers to and from the device are included in the reported timings.

A. Multicore

For computation of homology on multicore systems, we use METIS library for generating load-balanced graph partitions. REDHOM is written in C++ and the parallelization is implemented in OpenMP. The experiments were performed on an x86_64 Linux machine with 16GB of RAM and a 2GHz Intel Xeon Processor E5-2650. We enable hyperthreading to get 16 processing threads over the 8 physical cores.

The time taken by the various functions during sequential algebraic reductions in REDHOM is shown in Figure 4. In all cases, construction of reducible complex is the most time-consuming operation followed by algebraic reductions. The complex read and construction of the simplicial complex is done sequentially. Codes are assigned to the simplices and reducible complexes are constructed on the different partitions in parallel. Algebraic reductions are performed on all non-boundary simplices in parallel following which the unreduced simplices are merged.

The results of this parallelization for different number of threads on SYNTH dataset are presented in Figure 8.

For all datasets, it is observed that the algebraic reductions step of the sequential algorithm scales linearly with increasing number of cores. We obtain up to 10.7× speedup for this step with 16 cores. We notice that the execution times for construction of the reducible complex decrease with increasing number of cores. With 16 cores, the maximum speedup attained is 8.76× over the sequential execution of this step. Also, the initial read and construction of the simplicial complex is performed sequentially and lack of parallelism in this step eventually makes this function a major contributor to the execution time. The total execution times for the various datasets with increasing number of cores is in Figure 9. Performance gains of up to 4.9× are obtained over sequential algebraic reductions.

B. GPU

The GPU algorithm for homology computations was implemented in CUDA for devices with compute capability 2.0 and higher. The $-arch = compute\_20$ and $code = sm\_20$ flags are used for compiling the code with the Nvidia compiler nvcc 5.0. We evaluate the performance of our algorithm on a 1.15GHz Nvidia Tesla C2070 Card. It belongs to the Fermi GPU series and has 6GB of global memory and 448 CUDA cores.

The timings vary over different runs for the same dataset due to the randomized nature of the algorithm. For comparisons, we use the average times over 20 runs of homology computations for each dataset. We list the maximum, minimum, and average times (in seconds) for each dataset in the following table.

-	Max. time	Min. time	Avg. time
BLUNT	0.57s	0.48s	0.51s
POST	1.67s	0.96s	1.32s
BUCKY	2.93s	2.26s	2.71s
SYNTH	4.73s	3.19s	4.18s

The speedups of the average GPU timings over the sequential algebraic reductions are shown in Figure 10. For SYNTH and POST datasets, we obtain about 40× speedup. In Figure 11, we compare the performance of multicore algebraic reductions and the GPU algorithm for reductions. The GPU algorithm performs upto 9.8× better than multicore algebraic reductions.

We also perform experiments to compare the running times of the GPU algorithm against the optimized version of REDHOM. The optimized version includes an entire range of

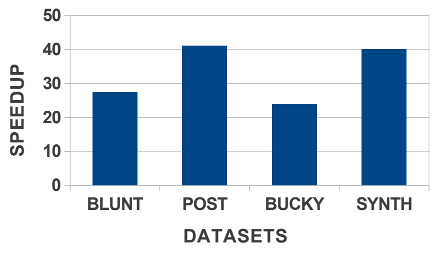


Fig. 10. Speedup of average GPU timings with respect to sequential algebraic reductions

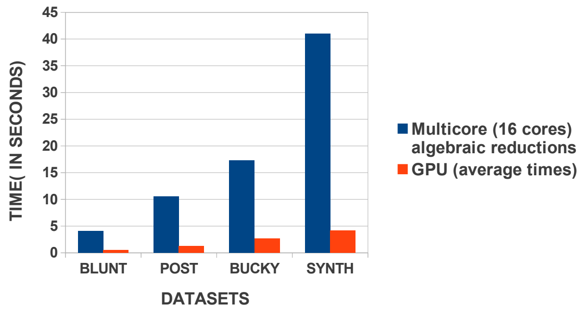


Fig. 11. Comparison of GPU and multicore timings

algorithms for efficient homology computation in addition to the algebraic reductions. Even when all homology computation engines of REDHOM are switched on, speedups of upto 4.5× are observed with the GPU algorithm, as seen in Figure 12.

We also observe better speedups for POST and SYNTH datasets compared to the other datasets for both the algorithms. In Figure 4, we notice that for both these datasets the time spent in algebraic reductions forms a high percentage of the total execution time. This relationship between the contribution of the algebraic reduction step to the the total execution time and the speedups obtained for the dataset was observed in general for all the datasets on which we tested our algorithm.

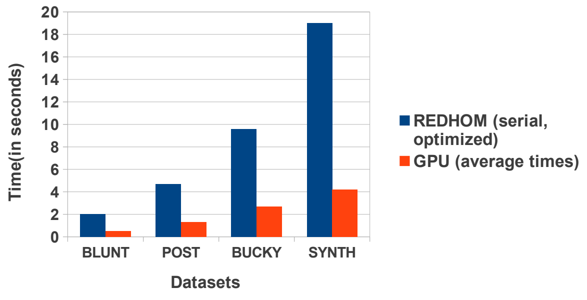


Fig. 12. Comparison of average GPU timings with optimized REDHOM, which includes the sequential algebraic reductions together with other advanced homology engines supported in the software.

VII. Conclusions and Future Work

In this work, we have developed algorithms for homology computations on multicore and manycore GPU systems. We observe up to 4.9× speedup with 16 cores over sequential algebraic reductions on multicore systems. A speedup of up to 40× over the sequential algebraic reductions is observed using our GPU algorithm. The GPU algorithm compares favourably with the REDHOM library which has a series of algorithms for homology computations, giving up to 4.5× performance gains.

We have explored the possibility of parallelization exclusively based on algebraic reductions. There are many other types of reduction algorithms implemented in REDHOM. We plan to extend our work further by identifying algorithms that work at a local level to reduce the size of the simplicial complex and then using a similar approach to parallelize it. Another possible extension could be parallel algorithms for homology computations in a distributed memory environment.

Acknowledgements

This work was partially supported by the Department of Science and Technology, India, under Grant SR/S3/EECE/0086/2012.

References

[1] T. Kaczynski, K. Mischaikow, and M. Mrozek, *Computational Homology*. New York: Springer, 2004, vol. 157.

[2] G. Singh, F. Memoli, T. Ishkhanov, G. Sapiro, G. Carlsson, and D. L. Ringach, "Topological analysis of population activity in visual cortex," *Journal of vision*, vol. 8, no. 8, 2008.

[3] S. Maadasamy, H. Doraiswamy, and V. Natarajan, "A hybrid parallel algorithm for computing and tracking level set topology." HiPC, 2012.

[4] A. Gyulassy, V. Natarajan, V. Pascucci, P.-T. Bremer, and B. Hamann, "A topological approach to simplification of three-dimensional scalar functions," *Visualization and Computer Graphics, IEEE Transactions on*, vol. 12, no. 4, pp. 474–484, 2006.

[5] H. Edelsbrunner and J. L. Harer, *Computational Topology: An Introduction*. American Mathematical Soc., 2010.

[6] R. van de Weygaert, G. Vegter, H. Edelsbrunner, B. J. Jones, P. Pranav, C. Park, W. A. Hellwing, B. Eldering, N. Kruithof, E. P. Bos *et al.*, "Alpha, betti and the megaparsec universe: on the topology of the cosmic web," in *Transactions on Computational Science XIV*. Springer, 2011, pp. 60–101.

[7] R. Ghrist and A. Muhammad, "Coverage and hole-detection in sensor networks via homology," in *Proc. Intl. symp. Information processing in sensor networks*. IEEE Press, 2005, p. 34.

[8] T. Kaczyński, M. Mrozek, and M. Ślusarek, "Homology computation by reduction of chain complexes," *Computers & Mathematics with Applications*, vol. 35, no. 4, pp. 59–70, 1998.

[9] L. Dagum and R. Menon, "Openmp: an industry standard api for shared-memory programming," *Computational Science & Engineering, IEEE*, vol. 5, no. 1, pp. 46–55, 1998.

[10] A. J. Zomorodian, *Topology for Computing*. Cambridge University Press, 2005.

[11] J. R. Munkres, *Elements of Algebraic Topology*. Addison-Wesley Reading, 1984.

[12] R. Kannan and A. Bachem, "Polynomial algorithms for computing the smith and hermite normal forms of an integer matrix," *SIAM Journal on Computing*, vol. 8, no. 4, pp. 499–507, 1979.

[13] B. R. Donald and D. R. Chang, "On the complexity of computing the homology type of a triangulation," in *Proc. Annual symp. Foundations of Computer Science*, 1991, pp. 650–661.

[14] C. J. A. Delfinado and H. Edelsbrunner, "An incremental algorithm for betti numbers of simplicial complexes on the 3-sphere," *Computer Aided Geometric Design*, vol. 12, no. 7, pp. 771–784, 1995.

[15] "REDHOM," http://redhom.ii.uj.edu.pl/.

[16] M. Mrozek and B. Batko, "Coreduction homology algorithm," *Discrete & Computational Geometry*, vol. 41, no. 1, pp. 96–118, 2009.

[17] M. Mrozek, P. Pilarczyk, and N. Żelazna, "Homology algorithm based on acyclic subspace," *Computers & Mathematics with Applications*, vol. 55, no. 11, pp. 2395–2412, 2008.

[18] S. Harker, K. Mischaikow, M. Mrozek, V. Nanda, H. Wagner, M. Juda, and P. Dłotko, "The efficiency of a homology algorithm based on discrete morse theory and coreductions," in *Proc. Intl. Workshop Computational Topology in Image Context (CTIC 2010). Image A*, vol. 1, 2010, pp. 41–47.

[19] R. H. Lewis and A. Zomorodian, "Multicore homology," http://comptop.stanford.edu/preprints/, 2012.

[20] "METIS," http://glaros.dtc.umn.edu/gkhome/views/metis.

[21] R. Nasre, M. Burtscher, and K. Pingali, "Morph algorithms on gpus," in *Proc. ACM SIGPLAN symp. Principles and practice of parallel programming*, 2013, pp. 147–156.

[22] "Aim@Shape," http://www.aimatshape.net/.