INTERNATIONAL JOURNAL FOR NUMERICAL METHODS IN ENGINEERING
Int. J. Numer. Meth. Engng 2003; **58**:161–176 (DOI: 10.1002/nme.765)

Parallel Delaunay mesh generation kernel

Nikos Chrisochoides[1,*,†] and Démian Nave[2]

[1]*Department of Computer Science, College of William and Mary, Williamsburg, VA 23187, U.S.A.*
[2]*Pittsburgh Supercomputing Center, Carnegie Mellon University, Pittsburgh, PA 15213, U.S.A.*

SUMMARY

We present the results of an evaluation study on the re-structuring of a latency-bound mesh generation algorithm into a latency-tolerant parallel kernel. We use concurrency at a fine-grain level to tolerate long, variable, and unpredictable latencies of remote data gather operations required for parallel guaranteed quality Delaunay triangulations. Our performance data from a 16 node SP2 and 32 node Cluster of Sparc Workstations suggest that more than 90% of the latency from remote data gather operations can be masked effectively at the cost of increasing communication overhead between 2 and 20% of the total run time. Despite the increase in the communication overhead the latency-tolerant mesh generation kernel we present in this paper can generate tetrahedral meshes for parallel field solvers eight to nine times faster than the traditional approach. Copyright © 2003 John Wiley & Sons, Ltd.

KEY WORDS: mesh generation; parallel computing; Delaunay triangulation; guaranteed quality; latency tolerant algorithms

1. INTRODUCTION

In this paper we focus on the re-structuring and the evaluation of a basic parallel Delaunay triangulation algorithm which is known as the Bowyer–Watson (BW) kernel [1, 2]. The BW kernel has been used successfully during the last twenty years for unstructured (tetrahedral) mesh generation on sequential machines [3–6]. The BW kernel is latency-bound because its computations are tightly coupled and memory intensive. The communication and synchronization latencies associated with the parallel execution of the BW kernel are long, variable, and unpredictable, because they involve data gather operations which use searching methods based on complicated floating-point operations; the number of floating-point operations required for searching the mesh is input dependent.

Parallel mesh generation programs, like any other parallel programs, should minimize communication overhead (cycles spent by a processor to push or pull messages from the

*Correspondence to: Nikos Chrisochoides, Department of Computer Science, College of William and Mary, Williamsburg, VA 23187, U.S.A.
†E-mail: nikos@cs.wm.edu

Contract/grant sponsor: NSF; contract/grant number: CCR-0049086, ACI-0085969, EIA-9972853, EIA-9726388

Received 30 January 2002
Accepted 4 November 2002

network interface) and tolerate the latency of remote service requests (cycles spent by a processor waiting for messages or remote requests to be serviced by other processors); throughout this paper we call this communication latency. Parallel mesh generation programs in addition should maintain *stability* i.e. parallel meshes should retain the quality of elements and partition properties of the sequentially generated and partitioned meshes. Mesh stability is an important requirement for the accuracy and cost effectiveness of parallel partial differential equation (PDE) solvers.

Therefore, in this paper, we focus on the design, implementation, and evaluation of a *stable*, *latency-tolerant*, *and scalable* Delaunay mesh generation kernel. The communication overhead is minimized by maintaining low surface-to-volume ratio of the submeshes as they are generated. The communication latencies are masked by overlapping communication with computation. The performance data we present are from an end-to-end application—*parallel Delaunay mesh* (PDM) for polyhedral domains—that considers the complete problem of generating, partitioning, and placing on the nodes of a parallel platform a tetrahedral mesh which is ready for parallel finite element analysis modules [7]. Our performance evaluation data suggest that it is possible to restructure the latency-bound BW algorithm and tolerate more than 90% of communication latencies at the cost of increasing the communication overhead between 2% (in the best case) and 20% (in the worst case). This is the first successful effort (to the best of our knowledge) in developing parallel and stable mesh generation methods by re-structuring scalar mesh generation kernels for tolerating communication latencies using a speculative execution model.

The rest of the paper is organized as follows: after providing an overview of the sequential BW kernel and parallel Delaunay mesh generation methods in Section 2, in Section 3 we describe our approach for parallelizing the BW kernel and we present a detailed latency tolerant element creation procedure which guarantees the stability and efficiency of the parallel BW kernel. In Section 4 we present a detailed performance evaluation and we conclude with Section 5 which summarizes our findings and our future work.

2. BACKGROUND AND RELATED WORK

The Delaunay criterion has been used successfully for sequential mesh generation of complex geometries since the late 1980s. There are many different Delaunay triangulation methods based on divide-and-conquer and gift-wrapping methods [6]. However, the most popular Delaunay meshing techniques are incremental methods. Incremental methods start with an initial mesh (usually a boundary conforming mesh) which is refined incrementally by inserting new points (one at a time) using a spatial distribution technique. Each new point is re-connected with the existing points of the mesh in order to form a new triangulation or a new mesh. The difference between the various Delaunay incremental algorithms in the literature is due to: (1) different spatial point distribution methods for creating the points and (2) different local reconnection techniques for creating the triangles or tetrahedra.

The two most popular local re-connection methods are the flip edge/face methods [8] which are difficult and expensive to parallelize and the Bowyer–Watson (BW) kernel [1, 2]. The BW kernel is an iterative procedure: at each iteration, an existing Delaunay mesh, M_i is refined and a new mesh M_{i+1} is generated by inserting a new point p_i into M_i after recovering the Delaunay property of the mesh through the local transformation: $M_{i+1} = M_i - C_i + B_i$. This

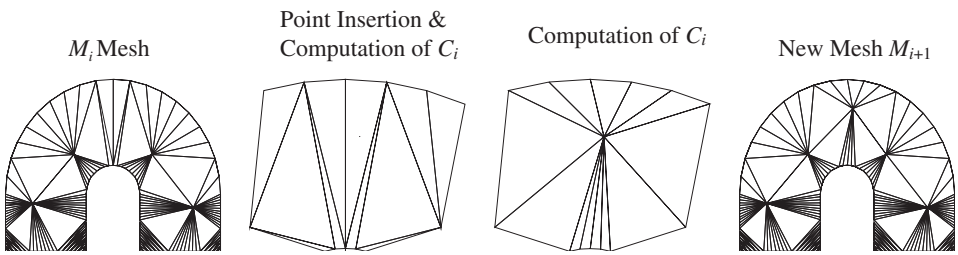

Figure 1. Point insertion and element creation steps of the BW algorithm; initial and final meshes at each iteration shown at the leftmost and the rightmost columns, respectively.

process is known as the BW kernel, it is described below.

Algorithm 1 ($BW(M_o, L_B, F_d)$)

1. **begin**
2. Input: M_o Delaunay mesh, L_B bad tets, F_d spatial distrib. function
3. $i \leftarrow 0$
4. **while** $L_B \neq \emptyset$ **do**
5. $t \leftarrow Get(L_B)$
6. Insert $p_i \notin M_i$ using F_d
7. $C_i \leftarrow \{t,\ t \in M_i$ that violate the Delaunay property$\}$
8. $B_i \leftarrow \{t,\ t$ a new tet $\ni$: $t \notin M_i$ and includes $p_i\}$
9. Set $M_{i+1} = M_i - C_i + B_i$
10. Insert in L_B the new tetrahedra $t \in B_i$ which are 'bad'
11. $i \leftarrow i + 1$
12. **endwhile**

Figure 1 depicts the ith iteration of the BW kernel. The efficiency of the Delaunay meshing methods depends on: (i) the efficiency of the searching algorithm that is used to identify the first triangle in conflict for each new point insertion, and (ii) the position in which the new points are inserted. Certain point insertion orders avoid the formation of short edges and thus the creation of bad[‡] triangles that would need to be improved again. Latency-aware algorithms for addressing such efficiency issues are out of the scope of this paper. In this paper we focus on a *latency tolerant implementation of the element creation step* (*lines* 7, 8 *and* 9) *whose computation is tightly coupled with variable and unpredictable calculation and communication behaviour*.

Parallel mesh generation is a relatively new area and only very few papers are published so far. A very good overview of parallel mesh generation papers can be found in Reference [9]. Practical parallel Delaunay triangulation methods relevant to the parallel BW kernel we present here are: (1) the Data-Parallel Delaunay Triangulation technique [10], (2) the Divide-and-Conquer Delaunay Triangulation method [11], (3) the Parallel Projective

[‡]There are many criteria to evaluate the quality of mesh elements. In this paper we use the circumradius to shortest-edge ratio. An element is 'bad' if this ratio is large.

Delaunay method [12], and (4) the Parallel Constrained Delaunay Mesh method [13]. None of the existing parallel Delaunay methods attempt to tolerate communication and synchronization latencies. Most of the parallel methods are designed to cope with the load imbalance problem.

3. PARALLEL BOWYER–WATSON KERNEL

In this section we present a stable and latency-tolerant parallelization of the BW kernel. While the sequential implementation of the BW kernel is relatively simple, its latency tolerant implementation is much more challenging. In order to efficiently tolerate communication latencies, one needs to explore concurrency at a fine-grain level which affects the *stability* of the Delaunay triangulation method. For example, if two points p and q are inserted concurrently and their cavities intersect (i.e. $C_p \cap C_q \neq \emptyset$), then the resulting cavities share faces and/or tetrahedra. Retriangulating intersecting cavities can lead to an inconsistent[§] and/or a non-Delaunay triangulation.

In the rest of this section we present the latency tolerant BW (LTBW) kernel and focus on its correctness, for three-dimensional Delaunay triangulations. Without loss of generality, we assume that the input to the LTBW kernel is a set of submeshes $M_i,\ i = 1, \ldots, N_s$ that partition the Delaunay triangulation T_o in the interior of a domain Ω. Also, we assume that $\forall p_i \in \bigcup_{i=1}^{N_s} M_i,\ C_i \cap \vartheta\Omega = \emptyset$, where $\vartheta\Omega$ is the external boundary of Ω. In Reference [14], we present an extension of the LTBW which allows meshing of the regions near the boundary of polyhedral domains.

3.1. Latency tolerant cavity creation

Each processor reads and 'owns' the data structure of one or more submeshes which will be refined further until stopping criteria like volume and quality criteria such as circumradius to shortest-edge ratio of tetrahedra are satisfied. The submeshes are treated as mobile objects [15] and they can move anytime in any processor for load balancing purposes. The MOL library [15] is responsible for maintaining a distributed directory and for efficiently forwarding messages to them. The partition of M_o into submeshes induces a separator, $I_{l,k}$, between submeshes, S_l and S_k if there exists at least one common face between them. $I_{l,k}$ consists of triangular *interface faces*, *edges*, *and vertices* which are replicated in all submeshes sharing them. Two submeshes S_l and S_k are called *adjacent* if $I_{l,k} \neq \emptyset$.

Each processor concurrently refines the mesh by inserting new points and re-triangulating their cavities. When tetrahedra in one of the submeshes S_l are non-Delaunay with respect to a point insertion in an adjacent submesh S_k, the cavity expands across the interface $I_{l,k}$ and it is called an *interface* cavity. Otherwise, the cavity is called *local or interior* and it is created and re-triangulated 'atomically'. Interface cavities remain active much longer than local cavities, because their expansion is interrupted and remote cavity expansion (remote data gather) is requested from other processors. Processors tolerate the long remote data gather operations by trying in the meantime to expand new cavities which are mostly local. Figure 2 depicts the partition of a triangulation and an interface cavity expanding along three subdomains.

[§]A mesh M is called consistent if the intersection of any two elements is either an empty set or a vertex in M, or an edge in M or a face (for 3-dimensional meshes).

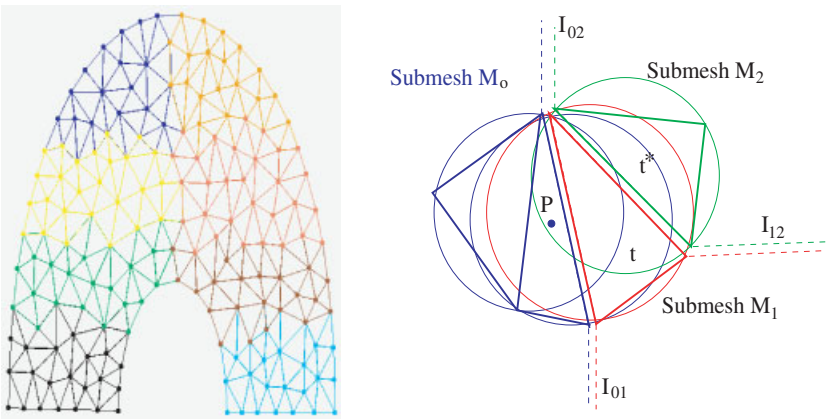

Figure 2. (Left) Submeshes of a triangulation and their inter-submesh interfaces. (Right) Cavity expansion over more than one submesh. The tetrahedra $t \in M_1$ and $t^* \in M_2$ are non-Delaunay w.r.t. point $P \in M_0$; thus the cavity C_P is an interface cavity. Dashed lines show the inter-submesh interfaces.

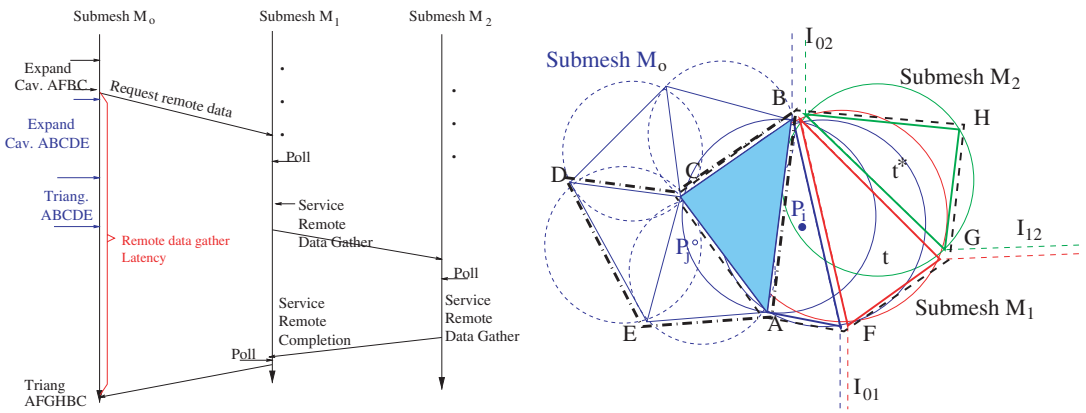

Figure 3. Left: Time diagram that depicts two overlapping cavities expanding concurrently while a remote request is serviced from the submesh M_1 for the expansion of the cavity AFBCA. Right: A new cavity ABCDEA is expanding concurrently in the submesh M_o. The two cavities share an element (shaded triangle ABC) and thus one of the cavities will free its elements (role-over or setback) and will expand at some point later.

After the completion of cavity expansion and re-triangulation, the data structure of the current triangulation is updated using the local transformation: $M_{i+1} = M_i - C_i + B_i$ (line 9 of BW kernel). Local data structure updates are atomic; some message passing is required to update connectivity information in the case of interface cavities. The distributed data structure and the new triangulation are consistent as long as, for any pair of points p and q, either $C_p \cap C_q = \emptyset$ or $C_p \cap C_q \neq \emptyset$ and the points are inserted into the mesh one after the other—the

specific order does not matter [5]. The BW algorithm terminates when all processors finish with the refinement of the mesh.

The BW kernel uses a simple list, L_B, to store 'bad' tetrahedra and two queues, Q_L and Q_R to store the interrupted local and remote active interface cavities, respectively. Adj(t) is the list of adjacent tetrahedra to t. The data structure for the faces are augmented so that the interface faces can store the mobile pointer [15] of the submesh that owns and maintains a second copy. The submeshes (or the processors) communicate to each other with remote service requests. Remote service requests are one sided messages that upon arrival of the message, a user defined handler is executed [16].

Algorithm 2 ($Latency\_Tolerant\_BW(M_{\text{current}}, L_B, F_d)$)

```
1.  begin
2.    repeat
3.      Poll network
4.      Service non-local remote interface cavity expansions stored in Q_R
5.      Service local interface cavity expansions stored in Q_L
6.      while L_B ≠ ∅ do
7.        t ← Get(L_B)
8.        Insert p_i ∉ M_current using F_d
9.        if ∃t_o ∈ M_current, where t_o contains p then set C_p ← {t_o}
10.       else request remote expansion for C_p endif
11.       if expand_cavity(C_p, p) = True then
12.           B_p ← {t, t ∉ M_current and includes p}
13.           M_current = M_current − C_p + B_p
14.           ∀t ∈ B_i that is 'bad' insert t ∈ L_B
15.       else Q_L ← C_p endif
16.       Poll network
17.       Service non-local remote interface cavity expansions stored in Q_R
18.       Service local interface cavity expansions stored in Q_L
19.     endwhile
20.     if (Q_R = ∅ & Q_L = ∅ & L_B = ∅) then set for termination state endif
21.   endrepeat when termination signal is received
22. end
```

The routine $expand\_cavity(C, p)$ (line 11, Algorithm 2) for cavity expansion is a source of communication overhead and communication latencies due to remote data gather operations. Figure 3 shows pictorially the latency of remote cavity expansion. However, $expand\_cavity(C, p)$, which is described in Algorithm 3, is designed to tolerate these communication latencies. Contrary to existing parallel mesh generation methods we have listed in Section 2, the *expand_cavity* permits the BW kernel to cross submesh interfaces. In order to tolerate the communication costs the LTBW kernel interrupts the computation of interface cavities. The interface cavities remain in a blocking and active state (i.e. hold their local elements) until all of their remote expansion requests are serviced and all elements that violate the Delaunay criterion are accumulated or until at least one of remote expansion requests is terminated—because it tries to expand on locked elements that already participate in another active cavity.

Algorithm 3 (*expand_cavity*(C, p))

```
1.  begin
2.    while there are unmarked tets t ∈ C do
3.      for t' ∈ Adj(t) do
4.        mark(t')
5.        if M_pid(t') = Pid then
6.          if t' is not locked and fails the empty-sphere criterion then
7.            lock(t')
8.            C ← C ∪ {t'}
9.          else
10.           send a rsr_expand(M_pid, t', p)
11.           return False
12.         endif
13.     end while
14.     return True
15. end
```

The concurrent triangulation of active cavities is a source of mesh inconsistencies and instabilities (i.e. non-Delaunay triangulations). Because it is possible for processors to handle more than one active cavities at a time. The LTBW kernel begins the expansion of new cavities due to new point insertions and it services other remote active interface cavity expansions while is waiting for active cavities to complete and triangulate. A pair of active cavities are related to each other in two possible ways:

Case I:

$$C_p \cap C_q = \emptyset, \quad p \neq q \tag{1}$$

The cavities do not intersect and thus they can be re-triangulated concurrently. Without loss of generality consider an initial Delaunay mesh, M_o, and two new points p, q within the convex hull of M_o such that $C_p \cap C_q = \emptyset$. Then by the BW kernel and the fundamental Delaunay Lemma [6] we have that the new meshes $M_{p,q}$ and $M_{q,p}$ that result from either:

$$M_{p,q} = M_p - C_q + B_q = (M_o - C_p + B_p) - C_q + B_q \tag{2}$$

or

$$M_{q,p} = M_q - C_p + B_p = (M_o - C_q + B_q) - C_p + B_p \tag{3}$$

are Delaunay triangulations. From (1), (2) and (3) and the uniqueness [6] of Delaunay triangulation—as long as the points of M are in general position¶—we have that $M_{p,q} = M_{q,p}$ and thus the points p and q can be inserted concurrently.

Case II:

$$C_p \cap C_q \neq \emptyset, \quad p \neq q \tag{4}$$

¶In practice it is not necessary to guarantee the general position of the points of M. $M_{p,q}$ and $M_{q,p}$ will be still consistent Delaunay meshes but maybe slightly different. In practice all we need is to maintain the Delaunay property.

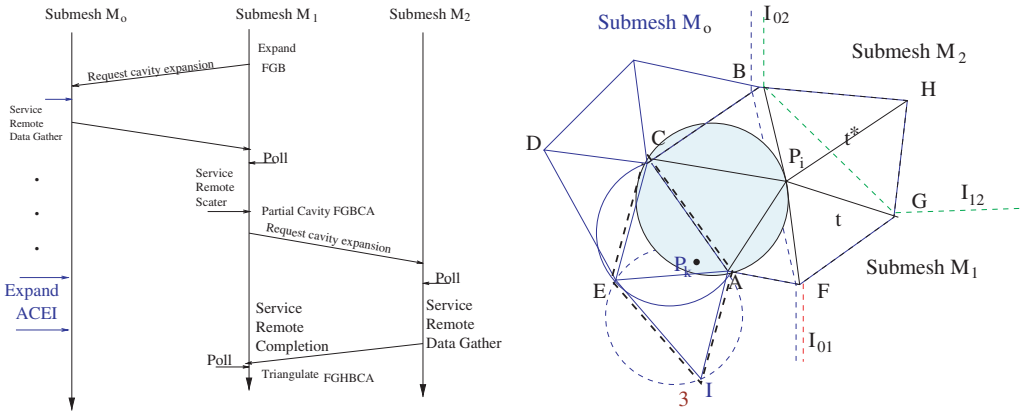

Figure 4. Time diagram of two adjacent cavities that expand concurrently. While the cavity AFGHBCA of the point P_i is expanding on one processor, the cavity ACEIA completes its expansion (on another processor) unaware of the new point P_i which violates the Delaunay criterion (shaded circle on the right).

The cavities intersect and have to be re-triangulated in a way that, for every pair of adjacent tetrahedra in the new triangulation, the empty sphere criterion holds and thus the fundamental Delaunay Lemma [6] can be applied. Otherwise the following inconsistency and correctness issues rise:

Case 1: $\exists t \in C_p \cup C_q$ such that $t \in C_p \cap C_q$

In this case the cavities share at least one tetrahedron and are called *overlapping* cavities. The concurrent re-triangulation of overlapping cavities results in an inconsistent mesh. Figure 3 depicts two cavities AFGHBCA and ABCDEA which overlap (share the triangle ABC). Their concurrent triangulation will lead to an inconsistent mesh, where the edges P_jB and CP_i will intersect in a point not in the mesh.

Case 2: $\nexists t \in C_p \cup C_q$ such that $t \in C_p \cap C_q$

In this case the two cavities share a face, an edge or vertex on their boundary surfaces. In the case that they share face (in three-dimensions) or an edge (in two-dimensions) they are called *adjacent*. The concurrent re-triangulation of adjacent cavities may result in a non-Delaunay mesh. Figure 4 depicts two cavities AFGHBCA and ACEIA which are adjacent (share the edge AC). Their concurrent triangulation will lead to a non-Delaunay triangulation, because the circumscribed circle of the new triangle AP_iC contains the point P_j. The invariant of the empty-sphere criterion does not hold after the transformation of line 13.

Both cases can be treated by *locking* all tetrahedra in the closure $\bar{C}_p$ of a cavity C_p:

$$\bar{C}_p = C_p \cup \{t, t \cap \vartheta C_p \neq \emptyset\}, \quad \vartheta C_p = \textit{is the boundary surface of } C_p \tag{5}$$

If any other cavity C_q tries to acquire a locked tetrahedron in $\bar{C}_p$, then the cavity C_q terminates and its elements are freed. The point q is queued and it is inserted in the mesh later.

Finally, upon the re-triangulation of cavities, each processor updates the data structure (triangulation) using the local transformation: $M_{\text{current}} = M_{\text{current}} - C_p + B_p$. These updates are

Table I. Breakdown of the total execution time (in seconds) using the traditional approach on a 16 processor SP.

	Sequential			Parallel		
Mesh size	Mesh gen.	CSR tran.	*I/O*	ParMetis	Data move.	Total time
1M tets	255.9	24.28	217.8	5.46	85.0	588.5
2M tets	461.8	50.53	471.0	8.58	173.3	1165.2

completed 'atomically' and their data structures and the new triangulation are both consistent. The new elements are 'owned' by the processor that is responsible for their creation. Subsequently, some of the submeshes may end up with many more elements depending on the initial distribution of the geometry (i.e. 'bad' elements). This approach might lead to workload imbalance that can affect the performance of the mesh generation.

Balancing the processors' workload in the LTBW kernel as well as in the case of the traditional parallel mesh generation methods is a source of synchronization and expensive memory access operations. The traditional explicit load balancing schemes use global synchronization in order to get all processors to the point that they can re-distribute the elements. Element re-distribution can take place either at the end of the mesh generation phase or in between the parallel mesh generation phases [17–19]. Table I breaks down the cost involved in element re-distribution. Although the solution of the element partition problem is inexpensive [20] the cost of moving the elements and updating the data structures is 94% of the load balancing step. This cost is expected to be much higher on large CoWs and the Grid [21] due to higher network latencies and lower bandwidth. In Reference [22] we have presented a partitioning method, SMGP, which distributes the new elements generated by the LTBW kernel as they are generated. This method eliminates the global synchronization, minimizes the *I/O* and data-movement overhead by eliminating redundant memory operations (loads/stores) from and to cache, local & remote memory, and disk.

4. PERFORMANCE EVALUATION

Experimental set-up: We have used as a Model Problem, Ω, a unit cube, within which is suspended a regular octahedral hole centred on the centroid of the cube. The vertices of the octahedron are positioned 0.25 units from the cube's centroid along the perpendicular bisectors of the cube's faces, such that vertices along the same bisector are 0.5 units apart. The domain Ω is discretized with two different meshes of 1 000 000 (1M) to 16 000 000 (16M) tetrahedral (tests) each. We have generated the same meshes sequentially on a Wide SP node. A Wide SP node is a 135 MHz Power2 SuperChip (P2SC) with 2 GB memory, 128 K cache, and 256 bit memory bus. For the parallel mesh generation we have used 16 Thin SP nodes and a cluster of 32 Sparc workstations (CoW). A Thin SP node is a 120 MHz P2SC with 1 GB memory, 128 K cache, 256 bit memory bus. The Thin SP nodes are connected via the SP switch; the SP switch is a TB3 switching fabric with 150 MB/s peak hardware bandwidth. The SP machine was located at Cornell Theory Center. The Solaris performance data were collected on a network of Sun Ultra 5 machines with 333 MHz processors connected by a 100 Mb/s

Table II. Total execution time (in seconds) for the sequential and traditional approach as well as the parallel mesh generation using the LTBW, its SMGP implementation, and the SMGP implementation post-processed by ParMetis for further improving the quality of the partitions on a 16 processor SP.

Mesh size	Seq. 1 Proc.	Traditional 16 Proc.	LTBW 16 Proc.	SMGP 16 Proc.	SMGP + ParMetis 16 Proc.
1M tets	255.9	588.5	38.63	71.10	71.54
2M tets	461.8	1165.2	85.15	87.46	126.57

fast-ethernet network and with 256 MB memory. The meshes were generated both sequentially and in parallel using the following two quality criteria: (1) minimize the maximum element volume, and (2) minimize the largest element circumradius-to-shortest-edge ratio (AR). The maximum element volume criterion controls the size of the mesh, while the AR weakly bounds the quality of the worst element in the mesh.

Overall evaluation: The parallel Delaunay mesh (PDM) generator [14] which uses the LTBW kernel and its SMGP implementation is eight to nine times faster than the traditional approach (see Tables I and II) without compromising the quality of partitions and elements of the mesh (see Figure 6 and Table III).

Table I shows a breakdown of the total execution time for generating, partitioning and placing the data structures of one million and two million tetrahedral meshes on the nodes of an SP machine for parallel field solvers [7]. The total execution time of the traditional approach in Table I is given by the sum of execution times: (1) for sequentially generating the mesh, (2) for transforming the mesh data structures into the CSR format—a widely used format for efficiently storing sparse graphs—that generic graph partitioning libraries like Parallel Metis [20] (ParMetis) require as input, (3) for loading the mesh data structure and the CSR data on the processors, (4) for partitioning in parallel the mesh-graph using ParMetis, (5) for migrating elements as dictated by ParMetis and updating the data structures so that re-partitioned meshes can be used by FE formulators. Despite the recent progress in parallel partitioning libraries and I/O hardware and software technology, it is still very expensive to sequentially generate unstructured meshes for parallel field solvers. The I/O overhead many times prevents engineers from using more efficient solvers which can adapt the mesh based on the needs of the analysis (or solution) phase [23].

Figure 5(left) shows a breakdown of computation and overheads like communication and load balancing for generating a half, one, and two million element meshes on a 16 node SP machine. The communication overhead is 33% for a half million and decreases to 31% and 26% for one and a two million element mesh, respectively. This is the time spent by the processor to push, pull, set-up remote service requests and cannot be hidden. The overhead due to decision making and data movement for load balancing is between 5 and 6%; it increases as the problem size increase. However, the communication and load balancing overhead of the parallel Delaunay mesh (PDM) code is between 8 and 20 times lower than the I/O and load balancing overheads of the traditional approach for the same size meshes. Figure 5(right) shows the execution times for different size meshes for LTBW and its SMGP implementation on 2, 4, 8, and 16 nodes of an SP machine. This figure suggests that the fixed speedup of the PDM code is $O(\log P)$, where P is the number of processors.

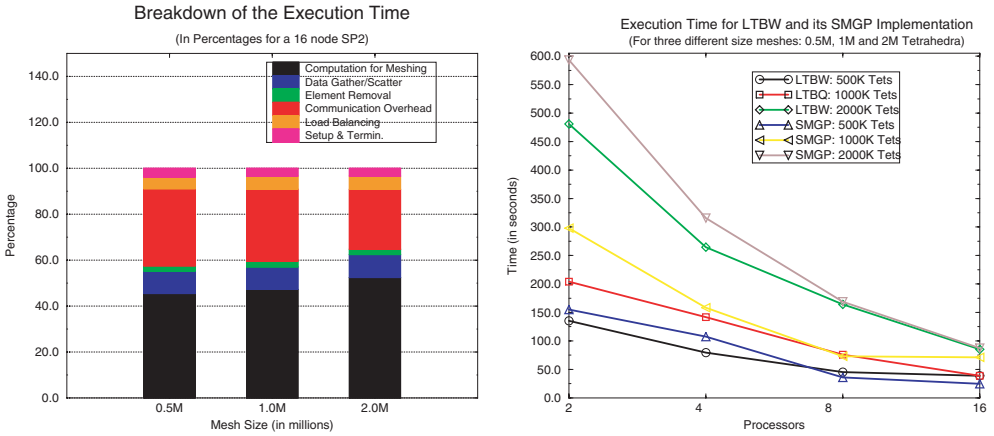

Figure 5. Breakdown in percentages of overheads occurred by a parallel mesh generator based on the SMGP implementation of the LTBW kernel, for 16 nodes on an SP machine and three different mesh sizes.

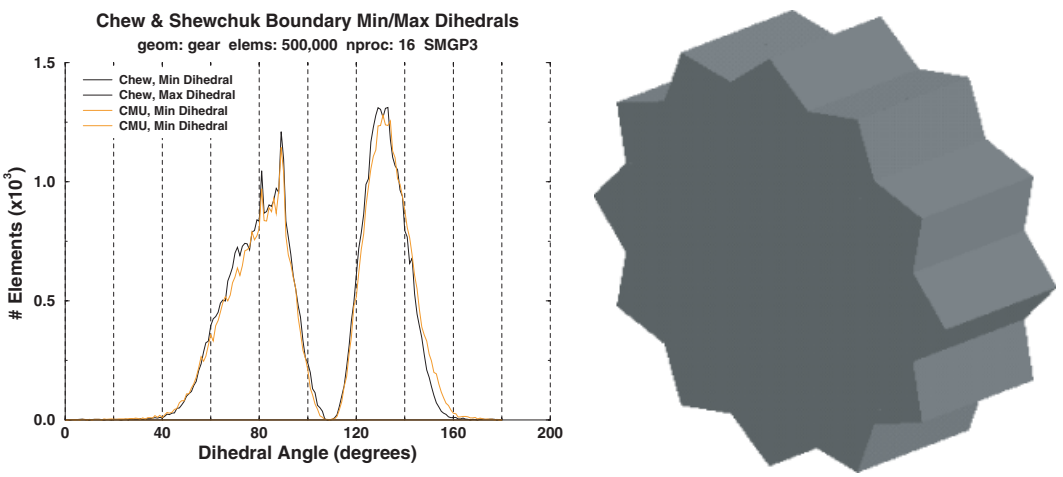

Figure 6. Quality of the mesh measured in terms of angles.

Finally, the PDM code for polyhedral domains retains the same quality of elements with the sequential mesh generator. Figure 6(left) shows the distribution of the elements in terms of their dihedral angles for a half million mesh generated on a 16 node SP; the discretized geometry is the gear-like geometry on the right. This figure depicts the quality of the elements generated by PDM using two different algorithms [5, 24] for meshing the geometry near by boundary [16], for the gear-like geometry. Moreover the LTBW kernel and its SMGP implementation generate partitions with very good quality i.e. the generated submeshes have very good equi-distribution of tetrahedra and surface-to-volume (S/V) ratio

Table III. Quality of submeshes measured in terms of the maximum surface-to-volume ratio (S/V) and imbalance on a 16 node SP.

	Traditional		LTBW		SMGP		SMGP+ParMetis	
Mesh size	S/V	Imbal.	S/V	Imbal.	S/V	Imbal.	S/V	Imbal.
1M tets	0.042	1298	0.101	11194	0.064	3058	0.044	883
2M tets	0.038	2009	0.090	23059	0.061	6578	0.037	1529

Table IV. Breakdown of the number of successfully completed cavities for the processor which is the least effective (first row) and most effective (last row) in tolerating latency for generating 16M element mesh for the Cube-in-Cube geometry on a 32 node (Sparc) CoW.

	Total number		No Act. Cav.		Act. Cav.	
Mesh size	Local	Interf	Local	Interf	Local	Interf
16M	77640	20577	32840	2893	44800	17684
16M	73270	20495	29826	2907	43444	17588

(see Table III) compared to the traditional state-of-the-art partitioners like Parallel Metis (ParMetis). Table III compares four different partitions of one and two million element meshes in terms of partitioning criteria like the S/V ratio, which is measured in terms of faces, and the imbalance, which is measured in terms of the difference between the submesh with the maximum number of tetrahedra and the submesh with the minimum number of tetrahedra.

4.1. Cost for tolerating latencies

Hiding latencies of remote service requests with throughput is not free! The LTBW in its effort to tolerate long latencies of remote interface cavity expansions, it inserts new points and it expands their cavities while it is waiting for the completion of other remote cavity expansions (see Figures 3 and 4). However, for stability reasons, some cavities terminate before they re-triangulate and free all of their elements. We call this form of a roll-back in the cavity computation a **setback** in the progress of the LTBW kernel. It is out of the scope of this paper to identify and analyze the relationship between setbacks and parameters like degree of concurrency, S/V ratio, and network latency. This study is complicated and will appear elsewhere. In this section we focus on two important questions: (1) *What's the impact of setbacks in the effectiveness of the LTBW kernel to tolerate long latencies?* (2) *Are there any other costs for tolerating latencies other than setbacks?* Next, we present statistics from the 'Cube-in-Cube' geometry with which we calculate the effectiveness and overheads of the LTBW to perform useful computations that overlap with the communication latency. We perform the rest of the analysis on the CoWs.

Table IV shows a breakdown of the number of successfully completed and triangulated cavities during the execution of LTBW. The percentage of cavities that are completed successfully in the presence of active interface cavities varies between 63 and 65% for the 16 M

Table V. Total run time for the processor with the worst latency tolerance. Also total time spend in expanding and triangulating cavities (meshing time), total time spent in expanding and triangulating cavities in the presence of active interface cavities i.e. meshing time in presence of active cavities (Act. Cavs), the number in parenthesis depict the time spend in meshing interface cavities, total setback time and percentage of latency the LTBW kernel utilizes effectively on a 32 node CoW for meshing the Cube-in-Cube geometry with 8 and 16 million elements. All times are in seconds.

Mesh size	Run time	Total compl. meshing time	Meshing time presence of Act. Cavs	Setback time	Latency tolerance
8M	201.30	107.37	54.78 (41.43)	4.18	0.96
16M	388.36	235.45	107.40 (42.39)	9.98	0.94

element mesh. This implies that the LTBW kernel during the time that waits for the completion of remote service requests, it completes more than 60% of the mesh refinement. The rest of the mesh refinement takes place while there are no active interface cavities present (see third and fourth columns of Table IV).

It is not easy to accurately measure the latency of all remote service requests in the LTBW kernel. However we can easily calculate the latency from the work that was accomplished during the presence of active cavities (cavities that are waiting for completion of remote service requests) and the work that has been lost due to the speculative nature of the LTBW kernel (i.e. setbacks). Our calculations from Table V show that the LTBW kernel effectively utilizes more than 90% of the total communication latency. Clearly the LTBW kernel is very effective in tolerating the communication latency. However, this is possible at a cost which we analyse next.

There are two options to increase the effectiveness of the LTBW. (1) Uncouple the submeshes and schedule point insertions and cavity creation in way that local cavities and active interface cavities do not intersect. The description of this option is out of the scope of this paper and is addressed elsewhere. (2) Reduce the number of local setbacks by reducing the 'life-time' and number of active interface cavities. The LTBW reduces the 'life-time' and number of active interface cavities by assigning the highest priority to all remote tasks (remote service requests) that are related to the completion of either local or remote interface cavity expansions. Since remote cavity expansions are unpredictable, the only way we can guarantee that remote expansion requests are serviced with the highest priority is to poll the network frequently for remote service requests. However, frequent polling of the network can result in *overpolling*. Figure 7 suggests that the communication overhead (cost of receiving remote service requests as soon as possible) for effectively tolerating the latency varies from processor to processor. Table VI suggests that between 2% (in the best case) and 14% (in the worst case) of the total run time is spent in unsuccessfully polling the network (i.e. although we poll the network there are no incoming messages). The percentage of unsuccessful polling increases as we decrease the size of the problem—for mesh sizes between 1 and 2 M elements the unsuccessful polling varies between 8 and 20% of the total run time—and imbalance increases (see Table VI). A more careful design and scheduling (load balancing) of the application will reduce this overhead. However because the computation and communication patterns in parallel Delaunay meshing are variable and unpredictable, we believe it will be very difficult to substantially reduce this overhead.

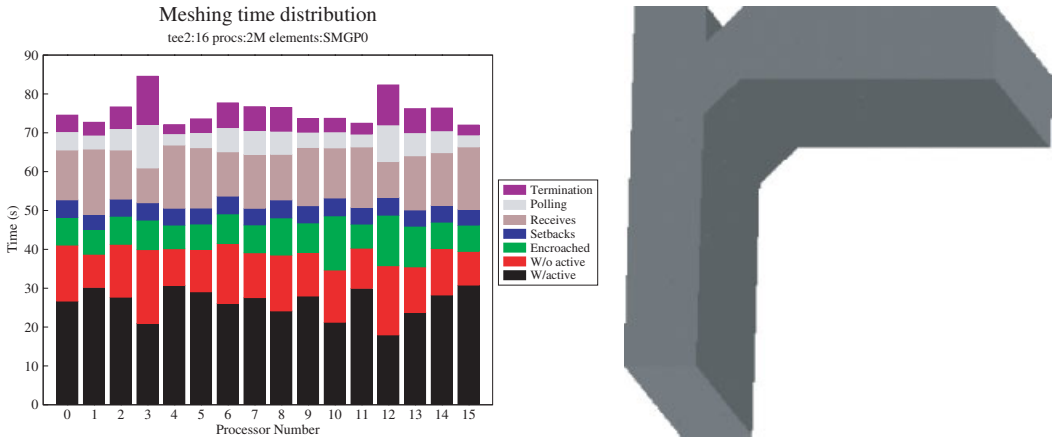

Figure 7. Breakdown in percentages of cavity computation occurred by a parallel mesh generator based on the LTBW kernel, for one and two million element meshes of the Tee geometry on 16 nodes of a CoW.

Table VI. Total run time, imbalance time due to variable and unpredictable nature of the BW kernel, and communication overhead (receive side) for the processor with least useful polling (first four rows) and maximum useful polling (last four rows). All times are in seconds, and were generated on a 32 node (Sparc) CoW.

			Polling time		Poll count		
Mesh size	Run time	Imbal. time	Total	Useful	Total	Useful	Message count
8M	201.30	33.95	58.63	30.11	861388	43292	200684
16M	388.26	49.38	93.15	51.19	1330290	71205	343375
8M	201.30	2.51	57.75	53.95	157744	50546	380254
16M	388.35	4.93	102.30	94.10	289530	91451	643776

5. CONCLUSIONS

In this paper we presented the results of an evaluation on the re-structuring of the latency-bound BW kernel into a latency-tolerant one. The task of tolerating the latencies of remote data gather operations with throughput using speculative execution and without introducing additional overheads is difficult because the latencies are *long*, *variable*, *and unpredictable*. Also, we analysed the overall performance of the LTBW kernel. Our data suggest that the LTBW kernel effectively tolerates more than 90% of the latency, which is equivalent to more than 60% of the total run time. The cost of tolerating latency varies between 2 and 20% of the total execution time and it is due to additional communication overhead (poll the network frequently in order to give high priority to remote service request handlers). However, despite these overheads the PDM code which is based on the LTBW kernel and its SMGP

implementation is eight to nine times faster than the traditional approach of generating, partitioning, and placing the data structures of unstructured meshes on the nodes of parallel platforms for parallel field solvers.

The main problem of the approach we presented here is the code complexity for retaining stability in the context of concurrency nearby the boundary of three-dimensional domains. A detailed description of a guaranteed quality and latency tolerant extension of the BW kernel for polyhedral domains is described elsewhere [14]. The kernel presented here is one of the three options for efficiently generating stable meshes on parallel platforms, today! The other two viable options which do not mathematically guarantee the quality of the elements but generate good quality meshes are a parallel octree based method [17] and a master-worker approach [19].

Our future work is focusing on investigating scheduling techniques that minimize communication overhead by using a combination of Delaunay admissible domain decomposition methods [12] and the SMGP implementation of the LTBW kernel.

ACKNOWLEDGEMENTS

We want thank the anonymous referees and our colleagues Keshav Pingali, Paul Chew, Paul Stodghil and Gerd Heber from Cornell for their continuous support and help to integrate this work in the crack propagation test-bed. Also we thank IBM's Research Program and Professor Marc Snir for helping us to acquire a useful, for this project, SP machine. Finally, this work was performed (in part) using computational facilities at the College of William and Mary which were enabled by grants from the National Science Foundation (EIA-9977030) and Sun Microsystems (SAR # EDU00-03-793, EDU-02-Q1-197).

REFERENCES

1. Bowyer A. Computing Dirichlet tessellations. *The Computer Journal* 1981; **24**(2):162–166.
2. Watson D. Computing the *n*-dimensional Delaunay tessellation with applications to Voronoi polytopes. *The Computer Journal* 1981; **24**(2):167–172.
3. Baker T. Delaunay triangulation for three dimensional mesh generation. *MAE Report 1733*, Princeton University, 1985.
4. Weatherill N. Delaunay triangulation in computational fluid dynamics. *Computers and Mathematics with Application* 1992; **24**(5/6):129–150.
5. Chew P. Guaranteed-quality triangular meshes. *Technical Report TR 89-983*, Department of Computer Science, Cornell University, 1989.
6. George PL, Borouchaki H. *Delaunay Triangulation and Meshing*: *Applications to Finite Element*. Hermis: Paris, 1998; 413.
7. Carter B, Chuin-Shan Chen, Chew LP, Chrisochoides N, Gao GR, Heber G, Ingraffea AR, Krause R, Myers C, Nave D, Pingali K, Stodghill P, Vavasis S, Wawrzynek PA. *Parallel FEM Simulation of Crack Propagation—Challenges*, *Status*, Lecture Notes in Computer Science, vol. 1800. Springer: Berlin, 2000; 443–449.
8. Lawson C. Transforming triangulations. *Discrete Mathematics* 1972; **3**:365–372.
9. de Cougny HL, Shephard MS. Parallel unstructured grid generation. *CRC Handbook of Grid Generation*, Thompson JF, Soni BK, Weatherill NP (eds). CRC Press: Boca Raton, 1999; 24.1–24.18.
10. Ansel Teng Y, Francis Sullivan, Isabel Beichl, Enrico Puppo. A data-parallel algorithm for three-dimensional Delaunay triangulation and its implementation. In *Supercomputing 93*. ACM: Providence, RI, 1993; 112–121.
11. Blelloch G, Hardwick J, Miller G, Talmor D. Design and implementation of a practical parallel Delaunay algorithm. *Algorithmica* 1999; **24**:243–269.
12. Galtier J, George PL. Prepartitioning as a way to mesh subdomains in parallel. *Proceedings of the 1997 ASME/ASCE/SES Summer Meeting*, *Special Symposium on Trends in Unstructured Mesh Generation*, Northwestern University, Evanston, IL, 29 June–2 July 1997.
13. Chew P, Chrisochoides N, Sukup F. Parallel constrained Delaunay meshing. In *Proceedings of the 1997 ASME/ASCE/SES Summer Meeting*, *Special Symposium on Trends in Unstructured Mesh Generation*, Northwestern University, Evanston, IL, 29 June–2 July 1997.

14. Nave D, Chrisochoides N, Chew P. Parallel Delaunay mesh generation for polyhedral domains. *Proceedings of the 18th Symposium on Computational Geometry*, Barcelona, Spain, 2002, pp. 135–144.
15. Chrisochoides N, Barker K, Nave D, Hawblitzel C. Mobile object layer: a runtime substrate for parallel adaptive and irregular computations. *Advances in Engineering Software* 2000; **31**(8–9):621–637.
16. Barker K, Chrisochoides N, Dobbelaere J, Nave D, Pingali K. Data movement and control supstrate for parallel adaptive applications. *Concurrency and Computation Practice and Experience* 2002; **14**:1–15.
17. Shephard M, Georges M. Automatic three-dimensional mesh generation by the finite octree technique. *International Journal for Numerical Methods in Engineering* 1991; **32**:709–749.
18. Lohner R, Cebral J. Parallel advancing front grid generation. *Proceedings*, *8th International Meshing Roundtable*, South Lake Tahoe, CA, U.S.A., 1999; 67–74.
19. Said R, Weatherill N, Morgan K, Verhoeven N. Distributed parallel Delaunay mesh generation. *Computer Methods in Applied Mechanics and Engineering* 1999; **177**:109–125.
20. Schloegel K, Karypis G, Kuma V. Parallel multilevel diffusion schemes for repartitioning of adaptive meshes. *Technical Report 97-014*, University of Minnesota, 1997.
21. Foster I, Kesselman C (eds). *The Grid*: *Blueprint for a New Computing Infrastructure*. Morgan Kaufmann: Los Altos, CA, 1999.
22. Chrisochoides N. New approach to parallel mesh generation and partitioning problem. In *Computational Science*, *Mathematics and Software* 2001; 319–344.
23. Bao H, Bielak J, Ghattas O, Kallivokas LF, O'Hallaron DR, Shewchuk JR, Xu J. Earthquake ground motion modeling on parallel computers. *Supercomputing*' *96*, Pittsburgh, PA, November 1996.
24. Shewchuk J. Delaunay refinement mesh generation. *Ph.D. Thesis*, School of Computer Science, Carnegie Mellon University, May 1997, Available as CMU Tech Report CMU-CS-97-137.

