并行多维 Delaunay 三角剖分实验报告

王元炜　卢政荣

1. 问题背景

二维平面上的 Delaunay 三角剖分是在一个给定的点集上进行的满足空圆性质的三角剖分，空圆性质即此剖分中的每个三角形的外接圆均不包含此点集中的其他点。在计算机图形学、数据可视化、模式识别等领域中，Delaunay 三角剖分均得到了广泛的应用。除此之外，我们还可以将其思想推广到 d 维空间中点集的 Delaunay 三角剖分，即将三角形推广为 $d+1$ 个两两之间有边相连的点构成的单纯形，需要满足每个单纯形的外接超球中没有此点集中的点。如果点集满足不存在 $d+2$ 个点共超球面的情况，则此剖分是唯一存在的。由于在二维平面上，Delaunay 三角剖分是 Voronoi 图的对偶图，一般的 Voronoi 图生成算法也可被用来生成 Delaunay 三角剖分，除此之外，还有如随机增量法等 Delaunay 三角剖分的生成算法在实际中应用效果非常好。但随着算法的大规模应用，其需要处理的数据量也越来越大，而算法本身已经达到了理论上的最优复杂度 $O(n \log n)$，无法从复杂度方面再进行优化了。在这种情况下，为了在数据规模更大的情况下仍然保证实际应用中的效率，需要对算法进行并行化，而传统的算法如随机增量法其每一个点的插入都依赖于之前的所有点构成的结果，天然是串行的结果，难以进行并行化，这就需要设计出可以高效并行的 Delaunay 三角剖分算法。

2. 相关工作

事实上已经有了非常多的基于分治和增量算法的并行 Delaunay 三角剖分算法。GE Blelloch 在 1999 年的工作中 [2]便已经提出了基于分治策略的 Delaunay 三角剖分的并行，即对于 Delaunay 三角剖分的分治算法，对于每个分治分支进行并行的处理从而达到并行的特征，但是对于分支的合并算法并没有给出很好的并行算法，也没有对分布式内存系统给出很好的解决方案。

Kohout 等人 [5]基于三角剖分增量式算法和分治算法给出了并行的二维和三维的 Delaunay 三角剖分工作，能够在小型共享内存集群上获得近似线性的加速比。但是对于分布式集群和多 cpu 并不能获得非常好的结局方案。

CGAL[1]是一个并行的计算几何数学库，它实现了很多常用的基础算法。并且能够得到一个不错的效率，但是它只实现了 2 维和 3 维的三角剖分并且没有实现分布式的算法。

Cignoni 等人 [3]提出了基于分布式递归算法的三角剖分算法。能够在小型集群上获得不错的结果。但是由于分治的合并算法需要用到大量 alltoall 通信，并不能获得一个非常好的扩展性。实际上该试验在 16 机上仅取得了 7 倍的加速比。

Funke 和 Sanders[4]提出了一个基于分治算法的可以适用于分布式和共享内存的并行算法，他们和核心贡献是提出了一个高度可并行的合并算法，并且基于这个合并算法能够在共享内存上对于 d 维 Delaunay 三角剖分获得近似线性的加速比。并且此工作也给出了一个和 [3]类似的并行实现。是第一份在比较大规模的集群上测试了次算法的扩展性。但是实际结果并不理想。工作在 2048 个计算核心的情况下只取得了 260 倍的加速比。

3. 解决方法

本项工作计划在实现共享内存算法的基础上实现分布式算法，并尝试通过改进 alltoall 的通信模式

来降低分布式的通信开销。从而达到一个不错的可扩展性。能够解决一个较大规模的多维三角剖分问题，并能获得客观的弱可扩展性。

4. 共享内存算法

Algorithm 1 Delaunay($\mathbf{P}, r$): shared memory parallel D&C algorithm

Input: points $\mathbf{P} = \{p_1, \ldots, p_n\}$ with $p_i \in \mathbb{R}^D$, recursion level r
Output: Delaunay triangulation T

1: **if** $n < N \vee r = \log P$ **then** ▷ for P processors
2: **return** *sequentialDelaunay*($\mathbf{P}$) ▷ base case
3: $k \leftarrow$ splittingDimension($\mathbf{P}$)
4: $(\mathbf{P}_1 \;\; \mathbf{P}_2) = (p_1 \;\cdots\; p_s \;\mid\; p_{s+1} \;\cdots\; p_n) \leftarrow$ divide($\mathbf{P}, k$) ▷ partition points in dim. k
5: $\mathbf{T} = (T_1 \;\; T_2) \leftarrow$ (Delaunay($\mathbf{P}_1, r+1$) Delaunay($\mathbf{P}_2, r+1$)) ▷ in parallel

Border triangulation:

6: $\mathbf{B} \leftarrow \varnothing$; $\mathbf{Q} \leftarrow$ convexHull(T_1) $\cup$ convexHull(T_2) ▷ initialize set of border simplices and work queue
7: **parfor** $s_{i,x} \in \mathbf{Q}$ **do** ▷ simplex originating from triangulation T_i
8: mark($s_{i,x}$) ▷ only process each simplex once
9: **if** circumsphere($s_{i,x}$) $\cap$ boundingBox(T_j) $\neq \varnothing$, with $i \neq j$ **then**
10: $\mathbf{B} \mathrel{+}= \{s_{i,x}\}$ ▷ circumsphere intersects other partition $\Rightarrow s_{i,x}$ is a borders simplex
11: **for** $s_{i,y} \in$ neighbors($s_{i,x}$) $\wedge \neg$ marked($s_{i,y}$) **do** ▷ process all neighbors
12: $\mathbf{Q} \mathrel{+}= s_{i,y}$; mark($s_{i,y}$)
13: $T_B \leftarrow$ Delaunay(vertices($\mathbf{B}$), $r+1$) ▷ triangulate points of border simplices

Merging:

14: $T \leftarrow (T_1 \cup T_2) \setminus \mathbf{B}$; $\mathbf{Q} \leftarrow \varnothing$ ▷ merge partial triangulations stripped from border
15: **parfor** $s_b \in T_B$ **do** ▷ merge simplices from border triangulation
16: **if** vertices(s_b) $\not\subset \mathbf{P}_1 \wedge$ vertices(s_b) $\not\subset \mathbf{P}_2$ **then**
17: $T \mathrel{+}= \{s_b\}$; $Q \mathrel{+}= \{s_b\}$ ▷ s_b spans both partitions
18: **else**
19: **if** $\exists s \in \mathbf{B}$: vertices(s) = vertices(s_b) **then**
20: $T \mathrel{+}= \{s_b\}$; $Q \mathrel{+}= \{s_b\}$ ▷ s_b replaces border simplex

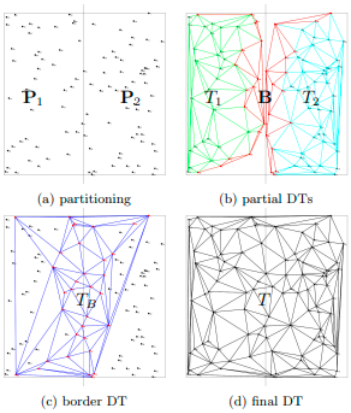
Neighborhood update:

21: **parfor** $s_x \in Q$ **do** ▷ update neighbors of inserted simplices
22: **for** $d \in \{1, \ldots, D+1\}$ **do**
23: **if** neighbors$_d(s_x) \notin T$ **then** ▷ neighbor not in triangulation anymore
24: $C \leftarrow \{s_c : f_d(s_x) = f_d(s_c)\}$ ▷ candidates with same facet hash
25: **for** $s_c \in C$ **do**
26: **if** $|$vertices(s_x) $\cap$ vertices(s_c)$| = D$ **then**
27: neighbors$_d(s_x) \leftarrow s_c$; $Q \mathrel{+}= s_c$ ▷ s_c is neighbor of s_x
28: **return** T

图 1. 共享内存算法流程

我们按照 [4]中描述的算法进行了实现，图1为其算法的伪代码流程。其算法基于递归分治的思想：首先将要计算 Delaunay 三角剖分的点集 P 按坐标划分为两个大小相等的子部分 P_1,P_2 并分别计算其三角剖分 T_1,T_2，然后考虑如何将这两部分的剖分面进行合并。首先我们可以发现，对于其中一个剖分中的三角形面，如果其外接圆与另一部分的点集没有交集，那么这个三角形面一定是安全的，我们可直接将其放入最终解 T 中，否则这个三角形面是危险的，可能被对面点集所破坏，我们暂时不能将这些面加入最终结果，我们将其放入 B 中进行存储。之后将所有 B 中面的点合并起来再做一次三角剖分得到 T_B；如果这个三角剖分中的某个面与原有的 B 中的某个面相同，则说明这个原有的面也不会被对面破坏，可以添加到最终结果 T 中；其次，如果最后剖分中有一个三角形跨越了两边的点集 T_1,T_2，说明这个三角形面是连接两个子集的，也要添加到最后结果 T 中。图2中的例子直观地表现了危险边集 B，对其点集的剖分 T_B 以及合并后的结果。

纵观这个算法，其实现的主要耗时在于枚举三角形面并进行处理，这使得其非常适合并行化：只要



(a) partitioning (b) partial DTs (c) border DT (d) final DT

图 2. 子剖分与合并结果

使用不同的线程来并行处理这些三角形面即可。其分治的实现也有利于并行化，我们可以在递归的过程中，将线程也递归划分到两边，直到只有一个线程时使用传统的串行算法。

在原论文的算法实现中，有一种特殊情况没有考虑：假如两个子点集恰好距离非常远以至于 B 为空，在这种情况下不会对中间进行处理，T_B 也为空，从而无法发现可能的一些中间连接面。我们在测试中发现了这一情况，并对其进行了修改：考虑两个子集的三角剖分 T_1,T_2 中的边，如果某条边只在一个三角形面上出现，说明这条边处于此三角剖分的边界上，可能需要与另一边连接，我们将这条边所在的面也视为是危险的，将其加入 B 中，以为其寻找与另一个点集相互连接的面。

5. 分布式内存设计框架

分布式内存设计的基本思路继承于共享内存设计，但是分布式需要进行一定的消息通信。从而能够充分利用各个处理器的计算能力。为了验证算法的多维能力，这里实现了三维分布式的三角剖分算法。算法提出了基本单机算法如下：

这里是将所有数据发送到某一个进程进行计算然后得到最终的结果。因为并行实现的内容比较复杂，并且在解耦中出现了一些问题导致工作量爆炸，这里采用了 CGAL 数学库所提供的单机三维三角剖分算法来代替。最终加速比比较也是与单机 CGAL 所提供的三角剖分算法进行比较。

Algorithm 3 DelaunayBase($\mathbf{P}_i, \mathbf{C}$): base case for distributed DT algorithm

Input: point subset $\mathbf{P}_i = \{p_{i,1}, \ldots, p_{i,n_i}\}$, PEs of partition $\mathbf{C}$
Output: Delaunay triangulation T_i of $\mathbf{P}_i$

1: **if** $|\mathbf{C}| = 1$ **then** ▷ base case
2: **return** Delaunay($\mathbf{P}_i$) ▷ shared memory DT algorithm
3: **if** $i = \min \mathbf{C}$ **then**
4: $\mathbf{P}' \leftarrow$ gather($\mathbf{P}_i, \mathbf{C}$) ▷ gather points from neighbors
5: $T' \leftarrow$ Delaunay($\mathbf{P}'$) broadcast(T') ▷ shared memory DT algorithm
6: **else** ▷ all other PEs
7: send($\mathbf{P}_i$) receive(T')
8: $T \leftarrow \{s \in T' : |\text{vertices}(s) \cap \mathbf{P}_i| \geq 1\}$ ▷ filter simplices
9: **return** T

图 3. 基础分布式三角剖分

Algorithm 2 Delaunay($\mathbf{P}_i, \mathbf{C}$): distributed memory parallel D&C algorithm

Input: point subset $\mathbf{P}_i = \{p_{i,1}, \ldots, p_{i,n_i}\}$, PEs of partition $\mathbf{C}$
Output: local view T of Delaunay triangulation $DT(\bigcup_{j \in \mathbf{C}} \mathbf{P}_j)$

1: **if** $\Sigma_{j \in \mathbf{C}} n_j < N \vee |\mathbf{C}| = 1$ **then** ▷ base case
2: **return** DelaunayBase($\mathbf{P}_i, \mathbf{C}$)
3: $S_i \leftarrow$ localVertexStatistics($\mathbf{P}_i$) ▷ local min, max and median
4: $S_{\text{all}} \leftarrow$ allReduce($S_i, \mathbf{C}$) ▷ global min, max and median
5: $k \leftarrow$ splittingDimension(S_{all})
6: $p_i = \text{median}(S_{i,k}) \geq \text{median}(S_{\text{all},k})$ ▷ PE's side of splitting plane in dim. k
7: $\mathbf{C}' \leftarrow \{j : p_j = p_i \quad \forall j \in \mathbf{C}\}$ ▷ set of all PEs on same side of splitting plane
8: $T \leftarrow$ Delaunay($\mathbf{P}_i, \mathbf{C}'$) ▷ triangulate own partition
9: $\mathbf{B}_i \leftarrow$ borderSimplices($T, \mathbf{C}, \mathbf{C}'$) ▷ simplices across splitting plane
10: $\mathbf{B}_i \leftarrow$ sparseAllToAll($\mathbf{B}_i, \mathbf{C}'$) ▷ receive border simplices from neighboring PEs
11: $\mathbf{C}_B \leftarrow \{j : \mathbf{B}_j \neq \varnothing \quad \forall j \in \mathbf{C}\}$ ▷ PEs with non-empty border
12: **if** $i \in \mathbf{C}_B$ **then**
13: $T_B \leftarrow$ Delaunay(vertices($\mathbf{B}_i$), $\mathbf{C}_B$) ▷ triangulate border vertices
14: $T \leftarrow$ merge($T, T_B, \mathbf{B}_i, \mathbf{P}_i, \mathbf{C}'$) ▷ slightly modified from Algorithm 1
15: $U_i \leftarrow$ updateNeighbors(T, Q) ▷ U_i set of performed updates
16: $U_{\text{all}} \leftarrow$ sparseAllToAll($U_i, \mathbf{C}_B$) ▷ exchange neighbor updates
17: **parfor** $(s_x \quad k \quad n) \in U_{\text{all}}$ **do** ▷ simplex s_x, neighbor no. k, neighbor simplex n
18: neighbors$_k(s_x) \leftarrow n$
19: **return** T

图 4. 分布式算法

本算法相比较于单机地算法，核心思想是如何均匀合理地将点分配到各个处理器上去。算法的中心思想是第 7-10 行。将通信子中的进程分为两部分。并使其进行尽可能小的通信的前提下，将全局的点分成两个子部分进行递归处理。由于本实验并没有以于超大规模为目标，故本算法的实现中并没有参考论文中提出的均衡算法而是在最开始对点进行了全局排序。从而减少了这种复杂的 alltoall 通信。然后依然是对与中间部分和可能与边线相交的面片（注意三维三角剖分中，面片对应的是三维超平面，即四棱锥）进行枚举。并通过邻居消息通信递归的完成面片通信。

6. 实验结果

由于采用了非常鲁棒的单线程算法，本算法可以顺利通过所有构造的小数据测试并能够在大数据上得到一个比较好的结果。

在 28 个核心 56 线程的实验机上，共享内存算

法对比单线程实现的算法。在一千万个点的数据上最多可以获得约 9.2x 的加速比（这是因为由于时间关系实现上依然存在一些不细致的地方）

在 36 进程下，三维三角剖分在 100 万点上可以获得对比单进程约 4x 的加速比，并且可以正确运行 1000 万点的数据。（加速比过低这是因为试验机并未安装 ib 网络，通信消耗了大量不必要的时间。在环境条件限制下并未能够在有 IB 网络的集群上配置实验环境所需的第三方库）

7. 实验分工

1. 卢政荣负责实现可并行算法，数据的制作。
2. 王元炜负责优化并行与分布式三维三角剖分算法设计以及数据可视化。

参考文献

[1] V. H. Batista, D. L. Millman, S. Pion, and J. Singler. Parallel geometric algorithms for multi-core computers. Computational Geometry, 43(8):663–677, 2010. 1

[2] G. E. Blelloch, G. L. Miller, J. C. Hardwick, and D. Talmor. Design and implementation of a practical parallel delaunay algorithm. Algorithmica, 24(3):243–269, 1999. 1

[3] P. Cignoni, C. Montani, R. Perego, and R. Scopigno. Parallel 3d delaunay triangulation. In Computer Graphics Forum, volume 12, pages 129–142. Wiley Online Library, 1993. 1

[4] D. Funke and P. Sanders. Parallel d-d delaunay triangulations in shared and distributed memory. In 2017 Proceedings of the Ninteenth Workshop on Algorithm Engineering and Experiments (ALENEX), pages 207–217. SIAM, 2017. 1, 2

[5] J. Kohout, I. Kolingerová, and J. Žára. Parallel delaunay triangulation in e2 and e3 for computers with shared memory. Parallel Computing, 31(5):491–522, 2005. 1