옥트리로 색인한 3차원 포인트 클라우드의 다중코어 기반 병렬 탐색

Multi-core-based Parallel Query of 3D Point Cloud Indexed in Octree

한수희¹⁾

Han, Soohee

Abstract

The aim of the present study is to enhance query speed of large 3D point cloud indexed in octree by parallel query using multi-cores. Especially, it is focused on developing methods of accessing multiple leaf nodes in octree concurrently to query points residing within a radius from a given coordinates. To the end, two parallel query methods are suggested using different strategies to distribute query overheads to each core: one using automatic division of 'for routines' in codes controlled by OpenMP and the other considering spatial division. Approximately 18 million 3D points gathered by a terrestrial laser scanner are indexed in octree and tested in a system with a 8-core CPU to evaluate the performances of a non-parallel and the two parallel methods. In results, the performances of the two parallel methods exceeded non-parallel one by several times and the two parallel rivals showed competing aspects confronting various query radii. Parallel query is expected to be accelerated by anticipated improvements of distribution strategies of query overhead to each core.

Keywords : LiDAR, 3D Point Cloud, Query, Octree, Parallel Processing

초 록

본 연구는 옥트리로 색인한 대용량 3차원 포인트 클라우드를 다중코어를 이용하여 병렬로 탐색함으로써 탐색 속도를 향상시키는 것을 목표로 한다. 특히 주어진 좌표로부터 일정 반경 내에 존재하는 포인트를 병렬로 탐색하기 위하여 다수의 리프 노드에 동시에 접근하는 방식을 개발하는 것에 초점을 두었다. 이를 위하여, 탐색 부하를 각 코어에 분배하는 과정에서 코드 중 for 루틴을 OpenMP에 의하여 자동으로 나누는 방식과 공간적 분할을 고려하는 방식 등 두 가지 병렬 탐색 방식을 제안하였다. 병렬 및 비병렬 탐색 방식을 평가하기 위하여 지상 레이저 스캐너로 취득한 약 1800만개의 3차원 포인트로부터 옥트리를 생성하고 8개 코어가 집적된 CPU가 1개 장착된 시스템에 적용하였다. 결과적으로 두 가지 병렬 탐색 방식 모두 비병렬 탐색 방식보다 수배의 성능 향상 효과를 나타내었으며, 두 병렬 방식은 탐색 반경에 따라 서로 경합하는 양상을 나타내었다. 향후 코어별 탐색 부하 분배 방식을 개선하여 병렬 탐색 속도를 향상시킬 수 있을 것으로 기대된다.

핵심어 : 라이다, 3차원 포인트 클라우드, 탐색, 옥트리, 병렬 처리

1. 서 론

3차원 지상 레이저 스캐너로부터 취득한 3차원 포인트 클

라우드(point cloud)는 역설계(reverse engineering) 분야에서 매우 중요한 자료로 사용되고 있다. 스캐닝 장비의 발달과 저장 장치의 가격 하락에 힘입어 포인트 클라우드의 크기가 급

1) Corresponding author, A regular member, Department of Geoinformatics Engineering (E-mail:scivile@kiu.ac.kr)

격히 증가하고 있어 대용량 자료의 효율적인 처리에 대한 관심이 높아지고 있다. 특히 관련 연구 또는 작업에서 필요성이 높은 포인트간 인접성 분석(proximity analysis)을 효율적으로 구현하기 위해서는 적절한 색인(indexing) 방식의 사용이 필수적이다(Han *et al.*, 2011). 옥트리는 이미 2.5차원 및 3차원 포인트 클라우드의 처리와 관련된 다양한 연구에서 사용되고 있으며(Saxena *et al.*, 1995; Woo *et al.*, 2002; Wang *et al.*, 2004; Schnabel *et al.*, 2007; Cho *et al.*, 2008; Marechal, 2009), 메모리 사용량 및 탐색 속도 면에서 합리적인 방식임이 증명되었다(Han *et al.*, 2011; Han *et al.*, 2012). 이에 Han(2013a)은 옥트리 구현 시 메모리를 더욱 효율적으로 사용하기 위하여 메모리 사용량에 영향을 미칠 수 있는 노드 클래스의 구성 변수와 트리 생성 방식을 분석하고 개선 방안을 제시하였다.

본 연구는 옥트리로 색인한 3차원 포인트 클라우드로부터 포인트의 탐색 속도를 향상시키는 것을 목표로 한다. 컴퓨터 그래픽스 분야에서는 옥트리의 생성과 공간 탐색 및 가시화를 고속으로 수행하기 위해 병렬처리를 수행하는 연구가 활발하게 진행되고 있다. 그러나 이를 3차원 포인트 클라우드의 색인 및 탐색에 바로 적용하기는 어려우며 옥트리의 구현 방법 및 병렬처리 하드웨어에 따라 다양한 탐색 방법이 개발될 수 있다. Point Cloud Library(PCL, 2013)라는 대규모 프로젝트에서는 포인트 클라우드의 처리와 관련된 다양한 API를 제공하고 있으며 본 연구의 목적과 유사한 기능을 하는 옥트리 생성 및 쿼리 API를 제공하고 있다. 아울러 Point Cloud Library는 병렬처리를 위하여 GPU(Graphic Processing Unit)를 사용하는 특징이 있다.

본 연구에서는 Han(2013a)이 제시한 메모리 효율적인 옥트리를 기반으로 최근 대부분의 CPU에 집적되는 다중코어(multi-core)를 이용하여 주어진 좌표 주변에 위치한 포인트들을 병렬로 탐색하기 위한 방법론을 제안하였다. 특히 OpenMP를 이용한 병렬 탐색 코드 제작 시 발생할 수 있는 문제점과 해결 방안, 서로 다른 탐색 방법에 의한 병렬 탐색의 성능 차이를 분석하였다.

본 논문의 2.1절에서는, C++ 언어로 구현한 일반적인 옥트리 코드의 구조와 문제점, 메모리 효율이 향상된 옥트리의 구조와 성능에 대해 Han(2013a)의 연구를 요약적으로 설명하였다. 2.2절에서는 옥트리로 색인한 3차원 포인트 클라우드의 OpenMP를 이용한 병렬 탐색 코드를 설명하였으며 2.3절에서는 일정 반경 내에 존재하는 포인트의 병렬 탐색을 위한 두 가지 방법론을 제시하였다. 3절과 4절에서는 비병렬 및 두 가지 병렬 탐색을 실제 3차원 포인트에 적용하여 실험한 결과와 결론을 기술하였다.

2. 본 론

2.1 메모리 효율적인 옥트리의 구조와 구현

옥트리는 계층적 트리 구조의 일종으로 상위 계층인 부모 노드(parent node)가 여덟 개의 자식 노드(child node)와 연결된 구조이며 쿼드트리(quadtree)의 3차원 확장이다. 공간적으로는 상위 노드가 관할하는 3차원 공간을 가로, 세로, 높이 방향으로 각각 양분하여 총 8개의 동일한 크기의 공간을 자식 노드가 관할한다(Fig. 1). 일반적으로 3차원 포인트 클라우드를 색인화하기 위하여 C++ 언어로 옥트리를 구현할 경우 각 노드는 자식 노드에 대한 포인터, 현 노드의 최소 경계 입방체(Minimum Bounding Hexahedral, 이 후 MBH) 정보를 저장하기 위한 변수, 현 노드의 단계(level)를 저장하기 위한 변수, 현 노드가 최하위 노드(leaf node, 리프 노드)일 경우 실제 포인트를 저장하기 위한 아카이브(archive)에 대한 포인터를 갖는다(Fig. 2). 이 경우 MBH의 좌표 변수의 형에 따라 노드 한 개의 크기는 36 bytes(float일 경우) 또는 60 bytes(double일 경우)가 되며, 자식 노드의 단계가 증가할 때마다 한 개의 부모 노드에 대하여 자식 노드가 8개씩 생성되므로 메모리 사용량이 크게 증가하는 문제점이 있다.

Fig. 1. Octree (Wikipedia, 2010)

메모리 효율적인 옥트리를 구현하기 위하여 Fig. 3과 같이 노드에 정의된 최소 경계 입방체 좌표와 단계 저장 변수를 제거하여 노드의 크기를 8 bytes로 감소시킬 수 있다. 제거된 변수들의 정보를 보전하기 위하여 변형된 포인트 입력 함수(AddPoint)를 통해 각 노드는 MBH(mbh)와 단계 값(level)을 부모 노드로부터 전달 받아 갱신 후 자식 노드에게 전달한다.

아울러, Fig. 2에서 자식 노드를 생성하고 리프 노드에서 vector 구조체를 생성하는 과정에서 메모리 동적 할당을 위한 new 연산자를 사용할 경우 작은 크기의 메모리를 반복적으로 동적 할당하는 과정에서 메모리를 추가적으로 차지함

```

struct MBH{ // 최소 경계 입방체 좌표 저장을 위한 구조체
    double minX, minY, minZ, maxX, maxY, maxZ // 좌상, 우하 3차원 좌표
}

Class CNode{
    MBH mbh // 최소 경계 입방체 좌표 구조체형 변수
    int level // 현 노드의 단계 저장 변수
    vector * ptList // 실제 포인트를 저장하기 위한 아카이브에 대한 포인터
    CNode * pChild // 자식 노드들에 대한 CNode형 포인터

    AddPoint(Point pt){ // 포인트 입력을 통한 트리 생성 함수
        int n = SelectChildUsing(mbh, pt) // 포인트를 포함하는 자식 노드 ID 결정
        if (level == DestLevel-1) // 현 노드가 리프 노드의 부모 노드인지 판별
            if (pChild[n].ptList == NULL)
                // 리프 노드에 vector 구조체가 선언되어 있는지 판별
                pChild[n].ptList = new vector
                // 리프 노드에 vector 구조체 동적 할당
            pChild[n].ptList.Add(pt) // 리프 노드에 포인트 저장
        else
            if (pChild == NULL) // 현 노드에 자식 노드가 없을 경우 판별
                pChild = new CNode[8] // 8개의 자식 노드 생성
                for i=0 to 7
                    pChild[i].mbh = CalcMBHUsing(mbh, i)
                    // 자식 노드에 새 MBH 부여
                    pChild[i].level = level + 1 // 자식 노드에 새 단계 값 부여
                pChild[n].AddPoint(pt) // 선택된 자식 노드에 재귀적으로 포인트 입력
    }
}

```

Fig. 2. Pseudo codes for basic octree node(Han, 2013)

```

Class CNode{
    vector * ptList
    CNode * pChild

    AddPoint(Point pt, MBH *mbh, int level){
        int n = SelectChildUsing(mbh, pt)
        if (level == DestLevel-1)
            ...
        else
            if (pChild == NULL)
                pChild = new CNode[8]
            mbh = CalcMBHUsing(mbh, i)
            pChild[n].AddPoint(pt, mbh, level+1)
            // 선택된 자식 노드에 포인트, 갱신된 mbh와 단계 값을
            // 재귀적으로 입력
    }
}

```

Fig. 3. Pseudo codes for MBH-free octree node(Han, 2013)

을 실험을 통해 확인하였다. 따라서 실제 트리 생성에 앞서 수도 트리(pseudo tree)를 생성하고 모든 부모 또는 자식 노드의 수와 리프 노드의 수를 산정하여 배열을 선언한 후 정식 트리를 생성하는 방법을 구현함으로써 메모리 효율성을 제고시킬 수 있음을 확인하였다.

본 연구에서는 이상 Han(2013a)의 연구에서 최종적으로 제안한 메모리 효율적인 옥트리 코드를 기반으로 병렬 탐색 코드를 개발하였다. 메모리 효율적인 옥트리에 대한 보다 자세한 내용은 Han(2013a)을 참조하도록 한다.

2.2 다중코어를 이용한 병렬 탐색의 구현

일반적으로 3차원 포인트 클라우드에 대한 탐색은 주어진 직육면체 안에 존재하는 포인트들을 탐색하거나 어떤 좌표 주위에 존재하는 포인트들을 탐색하기 위하여 수행하며 특정 좌표를 가진 포인트의 존재 유무를 판별하기 위한 목적으로도 수행한다. 옥트리의 최상위 노드인 헤드 노드(head node)로부터 목표 리프 노드에 이르기까지의 경로는 유일하며 경로에 속해 있는 모든 단계의 자식 노드에 대한 순차적인 접근을 통해 이루어진다. 따라서 한 개의 리프 노드에 접근하는 과정 자체를 병렬화하기는 어렵다. 대신 본 연구에서는 다수의 리프 노드에 병렬로 접근하는 탐색 방식을 제안하였다. Fig. 4에서 1개의 격자는 1개의 리프 노드에 할당된 공간을 의미하며 실제로는 직육면체 형태이나 편의상 평면 격자로 도시하였다. Fig. 4(left)와 같이 단일 코어만을 사용해서 9×9 격자의 포인트를 탐색할 경우 총 81개의 리프 노드에 순차적으로 접근해야 한다. Fig. 4(right)와 같이 4개의 다중코어를 사용할 경우 4개의 리프 노드에 병렬로 접근할 수 있으며 각 코어는 27 또

는 18개의 리프 노드에 접근하면 되므로 탐색 시간을 단축시킬 수 있다.

Fig. 4(right)와 같은 형식의 병렬 탐색은 CPU의 각 코어가 메인 메모리상에 구성된 옥트리에 접근 가능해야 하며, 일반적으로 공유 메모리를 기반으로 CPU의 다중 코어를 제어하기 위한 API인 OpenMP(OpenMP ARB. 2013)를 이용하여 구현할 수 있다. Fig. 5는 Fig. 4(right)의 병렬 탐색 개념을 z축 방향으로 확장시킨 것으로서, 주어진 포인트(pt)를 중심으로 3축 양방향의 거리가 range 이내인 포인트를 병렬로 탐색하기 위한 수도코드이다. CPU는 '#pragma omp parallel for' 전처리 명령문 다음 행에 위치한 for 루틴의 연산을 자동으로 균등하게 나누어 각 코어로 하여금 동시에 처리하도록 한다. 즉, x축 방향으로 $2 \times \text{range}$ 크기의 영역을 코어의 개수(n)로 분할하여 각 코어로 하여금 $(2 \times \text{range}/n) \times (2 \times \text{range}) \times (2 \times \text{range})$ 크기의 3차원 공간에 포함된 포인트를 탐색하도록 한다. 'private()' 구문은 for 루틴 안에서 각 코어에 의해 갱신되는 변수들에 의해 발생하는 false sharing(Chandler, 2010; Tian 등, 2012)을 방지하기 위한 것이다. Fig. 5에서 pt, range, octree(메모리에 구성되어 있는 옥트리) 등의 변수는 메모리의 같은 영역을 참조하고 있는 반면 x, y, z(탐색 대상 좌표), ptList(검출된 포인트들의 아카이브 포인트), mbh(노드의 최소입방체 경계 좌표) 등의 변수는 변수명은 동일하지만 각 코어마다 독립적으로 선언되고 갱신되어야 한다. 만약 private() 구문을 통해 x, y, z, ptList, mbh 등의 변수를 독립적으로 선언하지 않으면 어떤 코어가 그 값을 갱신할 경우 나머지 코어도 같은 값을 참조하여 정확한 연산이 수행되지 않는 문제가 발생한다. 이와 같은 현상을 false sharing이라고 하며 false sharing 현상이 발생할

Fig. 4. Concepts of point query using a single core(left) and multiple cores(right) (modified from Han(2003b))

```

ResetMBH(mbh); // mbh의 좌표를 전체 포인트 클라우드의 경계 입방체 좌표로 설정
#pragma omp parallel for private(x, y, z, ptList, mbh)
    // OpenMP 기반 for 루틴의 병렬화를 위한 전처리 명령문
for (x = pt.x-range; x<= pt.x+range; x++) // x축 방향으로 연산을 분할하여 병렬
처리
    for (y = pt.y-range; y<= pt.y+range; y++)
        for (z = pt.z-range; z<= pt.z+range; z++)
        {
            ptList = octree->GetPtListNear(x, y, z, mbh);
            // (x, y, z) 좌표를 포함하는 리프 노드로부터 포인트 아카이브를
            반환 받음
            if (ptList != empty)
                Do something...
        }

```

Fig. 5. Pseudo codes for parallel octree query based on OpenMP

```

Vector * Octree::GetPtListNear(double x, double y, double z, MBH * mbh)
{
    if (pChild == NULL)
        return ptList
    else
    {
        Select a child node using x, y, z and current node's mbh
        Update mbh for the selected child node // 자식 노드의 mbh 갱신
        return pChild[nChild].GetPtListNear(x, y, z, pMBR);
        // 자식 노드들에 대한 재귀적 함수 (nChild = 선택된 자식 노드의 ID)
    }
}

```

Fig. 6. Pseudo codes for function GetPtListNear

경우 처리 시간도 비약적으로 증가하여 비병렬 처리보다 더욱 많은 시간이 소요되는 문제도 발생한다. 따라서 '#pragma omp parallel for' 전처리 명령문 이후의 for 루틴에 사용되는 변수 중 그 값이 갱신되는 변수는 반드시 'private()' 구문에 추가해야 한다. 한편, Fig. 6의 GetPtListNear 함수 중 자식 노드의 mbh 갱신은 함수 내부에서만 수행되며 GetPtListNear 함수를 호출(call)하는 Fig. 5의 for 루틴에서는 사용되지 않으나 이러한 경우에도 false sharing 현상이 발생함을 확인하였다. 따라서 GetPtListNear 함수의 내부 변수인 mbh를 파라미터로 노출하고 private 변수 목록에 추가함으로써 false sharing 현상을 방지하였다.

2.3 일정 반경 내 탐색의 병렬 구현

포인트 클라우드의 탐색은 Fig. 4와 같이 직사각형 영역(실제로는 직육면체 영역)을 대상으로 수행하는 경우도 있으나 Fig. 7과 같이 주어진 좌표 또는 포인트로부터 일정 반경 내의 영역에 존재하는 포인트를 대상으로 수행하는 경우가 보다 빈번하다. 이와 같이 일정 반경 내부를 탐색하는 방식에 Fig. 4(right)와 같은 병렬 탐색 방식을 적용하면 그 형태는 Fig. 7(left)와 같다. 이 경우 탐색 과정은, ①Fig. 5의 for 루틴에서 갱신되는 (x, y, z) 좌표를 내포하는 리프 노드의 전체 또는 일부 영역이 탐색 반경 내부에 존재하는지를 판별하는 과정, ②헤드 노드로부터 ①의 조건을 만족하는 리프 노드까지

접근하는 과정, ③리프 노드의 아카이브 내에 존재하는 개별 포인트에 접근하는 과정, ④포인트가 주어진 반경 안에 존재하는지 재판별하는 과정으로 구분할 수 있다. 여기에서 과정 ①, ④는 단순 거리 계산으로, 과정 ②, ③에 비해 소요 시간이 미미하며, 과정 ②, ③은 Han(2013a)에서 분석하였듯이 트리의 단계, 즉, 가지의 길이와 리프 노드의 크기를 조절함으로써 소요 시간을 최소화할 수 있다. 그러나 Fig. 7(left)와 같은 병렬 탐색 방식은 코어간 부하 불균형이 발생하여 효율이 저하될 수 있으며 구체적으로 다음과 같은 세 가지 부하 불균형 원인이 존재한다.

첫째, 과정 ①에 의하여 일부 코어에서 실제로 탐색해야 하는 포인트 수가 감소하여 코어간 부하 불균형이 발생한다. 예를 들어 Fig. 7(left)에서 코어 1, 4는 모서리 부분의 일부 노드에 포함된 포인트에 대한 탐색을 생략할 수 있으므로 코어 2, 3

에 비하여 적은 포인트를 탐색하게 된다. 이 경우 보다 많은 부하가 가해진 코어 2, 3이 탐색 작업(task)을 마칠 때까지 코어 1, 4가 유휴(idle) 상태에 있으므로 병렬 탐색의 효율은 저하된다.

둘째, 탐색 대상 리프 노드의 수가 항상 홀수, 즉 $(2n+1)^2$, 개이므로 짝수의 가용 코어를 지닌 대부분의 시스템에서 탐색 대상 리프 노드의 수를 정확하게 분배하기 어렵다. 여기서 n 은 탐색 반지름에 해당하는 리프 노드의 수이다. 예를 들어 Fig. 7(left)에서 코어 1은 $3 \times 9 = 27$ 개의 리프 노드를 탐색해야 하는 반면 코어 2, 3, 4는 $2 \times 9 = 18$ 개의 리프 노드만 탐색하면 되므로 코어간 부하 불균형이 발생한다.

셋째, 탐색 반경이 작은 경우 일부 코어는 탐색에 사용되지 않을 수 있다. 예를 들어 Fig. 7(right)에서 비교적 작은 탐색 반경에 대하여 위와 같은 병렬 탐색 방식을 적용할 경우 코어 1, 2, 3은 탐색에 사용되나 코어 4는 사용되지 않는다.

Fig. 7. Concepts of parallel query within varying radii

Fig. 8. Concepts of more stratified version of Fig. 7

Table 1. Number of queried leaf nodes in Fig. 7(left) and Fig. 8(left)

Fig. 7 left				Fig. 8 left			
core 1	core 2	core 3	core 4	core 1	core 2	core 3	core 4
$2 \times 7 - 6 = 8$	$2 \times 7 = 14$	$2 \times 7 - 2 = 12$	$1 \times 7 - 4 = 3$	$3 \times 3 - 3 = 6$	$4 \times 3 - 3 = 9$	$3 \times 4 - 3 = 9$	$4 \times 4 - 3 = 13$
sum = 37, max = 14(core 2)				sum = 37, max = 13(core 4)			

Table 2. Number of queried leaf nodes in Fig. 9

Fig. 9 left				Fig. 9 right			
core 1	core 2	core 3	core 4	core 1	core 2	core 3	core 4
$3 \times 9 - 6 = 21$	$2 \times 9 = 18$	$2 \times 9 = 18$	$2 \times 9 - 6 = 12$	$4 \times 4 - 3 = 13$	$5 \times 4 - 3 = 17$	$4 \times 5 - 3 = 17$	$5 \times 5 - 3 = 22$
sum = 61, max = 21(core 1)				sum = 61, max = 22(core 4)

Fig. 9. Comparison of the two models in a different radius

본 연구에서는 상기의 문제점들을 완화하기 위하여 Fig. 8(left)와 같이 탐색 중심으로부터 각 축 방향으로 영역을 분할하여 병렬로 탐색하는 방식을 제안하였다. 이 방식의 경우 모든 코어가 모서리 부분의 노드를 탐색하여 보다 향상된 부하 분배 효과를 기대할 수 있다. 아울러 Fig. 8(right)와 같이 비교적 작은 탐색 반경에 대해서도 모든 코어를 사용할 수 있는 장점이 있다. 다만, 이 경우에도 탐색 대상 리프 노드의 수를 정확하게 분배하지 못하는 한계는 여전히 존재한다.

Fig. 7(left) 방식과 Fig. 8(left) 방식에 의한 병렬 처리 성능을 이론적으로 비교하기 위하여, 2.3절의 과정 ①에 의한 비탐색 영역을 제외한 코어별 최종 탐색 대상 리프 노드의 수를 산정한 결과는 Table 1과 같다. Fig. 7(left)의 경우 core 2에 의하여 최대 14개, Fig. 8(left)의 경우 core 4에 의하여 최대 13개의 리프 노드를 탐색하게 된다. 병렬 처리에서는 최대 부하가 걸린 코어가 전체 처리 시간을 결정하므로 Fig. 8(right)이 상대

적으로 적은 시간을 소요할 것으로 예상할 수 있다.

그러나 Fig. 9와 같이 다른 탐색 반경을 적용할 경우의 코어별 최종 탐색 대상 리프 노드의 수는 Table 2와 같다. 이 경우엔 Fig. 9(left)가 상대적으로 적은 시간을 소요할 것으로 예상할 수 있다. 따라서 두 가지 방식의 탐색 시간은 탐색 반경의 크기에 따라 대소 관계가 바뀔 것으로 예상할 수 있다.

3. 실험 및 고찰

실제 터널 내부에서 취득한 약 1800만개의 포인트로 구성된 포인트 클라우드를 대상으로 옥트리를 생성하고 8개의 코어가 집적된 CPU가 1개 장착된 시스템을 사용하여 일정 반경 내부의 포인트를 탐색하는 실험을 수행하였다. 실험에 사용한 포인트 클라우드 및 시스템의 상세 사양은 Tables 3 and 4와 같다.

Table 3. Specifications of point cloud

Terrestrial Laser Scanner	Scan Station 2, Leica Geosystems
Scanned Object	A Tunnel in Sejong-Si, South Korea (Length: 55.82m, Curvature: 252.50m, Width:10.52m)
Point Cloud	Number of points: 18,376,726 points Point Density: Approx. 0.25 points/1cm ²

Table 4. Specifications of processing system

CPU	AMD FX 8150 (8 cores running at 3.6GHz)
RAM	8 GB
OS	Windows 7 Professional 64bit
Code development and compilation	C++ 64bit mode in Visual Studio 2010 with OpenMP support

사전 실험을 통하여 가장 작은 탐색 시간을 소요하는 9단계를 적용하여 옥트리를 생성하였다. 포인트 클라우드의 1/50에 해당하는 367,535개의 샘플 포인트를 중심으로 일정 반경 안에 존재하는 포인트들에 접근하는데 소요된 시간을 측정하였으며 탐색 반경으로는 5cm부터 50cm까지 총 10개의 값을 사용하였다. 비교 대상 방식은 세 가지로서, ①비병렬 탐색 방식(이 후 Sequential), ②Fig. 7과 같이 OpenMP에 의해 for 루틴을 자동으로 분할하는 병렬 탐색 방식(이 후 Parallel), ③Fig. 8

Table 5. Shared properties among methods

Tree generation time(sec)	6.66(average)
Memory usage(MB)	225
Cell size in x, y, z(cm)	10.95, 4.99, 2.34
Search radii(cm)	No. of accessed points
5	157288485
10	610062972
15	1331926997
20	2305601532
25	3515743646
30	4951641066
35	6603904942
40	8464207518
45	10524386272
50	12777732727

과 같이 균등 부하 분배를 위해 공간적 분할을 수행한 병렬 탐색 방식(이 후 Stratified)이다. 옥트리 생성 속도 및 메모리 사용량, 리프 노드의 크기, 탐색 반경에 따라 접근한 포인트 수는 Table 5와 같으며, 백그라운드 서비스 상태 등 시스템 상황에 따라 생성 속도에 미세한 차이가 난 것을 제외하고는 세 가지 방식이 모두 같은 수치를 기록하였다.

세 가지 방식에 의한 탐색 시간은 Fig. 10과 같으며, 병렬 탐색 방식인 Parallel과 Stratified 방식이 비병렬 탐색 방식인 Sequential 방식에 비해 현저히 작게 측정되었다. Parallel 방식은 Sequential 방식에 비하여 1/5.3(반경 30cm)에서 1/2.8(반경 5cm)의 탐색 시간을 소요하였으며, Stratified 방식은 Sequential 방식에 비해 1/5.0(반경 5cm)에서 1/4.1(반경 20cm)의 탐색 시간을 소요하였다. 이와 같이 두 가지 병렬 탐색 방식이 비병렬 탐색 방식에 비해 우수한 성능을 나타냄을 확인할 수 있었다.

Fig. 10 Comparison of query time of the three models

두 가지 병렬 탐색 방식에 의한 탐색 시간은 Fig. 11과 같으며 서로 경합하는 양상을 나타내었다. 다만, Stratified 방식은 탐색 반경이 증가함에 따라 탐색 시간이 단조 증가하는 반면 Parallel 방식은 상대적으로 불안정하게 증가하는 현상을 보였다. 이와 같은 형상은 2.3절의 병렬 탐색의 효율 저하 요인 중 첫 번째 요인으로 해석할 수 있다. 즉, Parallel 방식의 경우 탐색 반경에 따라 탐색 부하의 불균형이 심화 또는 완화되는 현상이 반복되는 반면 Stratified 방식은 탐색 부하의 분배 상태가 탐색 반경의 변화에 크게 영향을 받지 않고 거의 일정하게 이루어지기 때문인 것으로 판단된다. 아울러 Fig. 12와 같이, 비교적 작은 탐색 반경에서는 Stratified 방식이 성능 우위를 나타내었다. 이와 같은 현상은 2.3절의 병렬 탐색의 효율 저하 요인 중 세 번째 요인에 의한 것으로, Parallel 방식에서는 모든 코어를 사용하지 못하고 일부 코어만 사용한 것으로 판단할 수 있다.

Fig. 11. Comparison of query time of Parallel and Stratified models

Fig. 12. Comparison of query time in smaller radii

4. 결론

본 연구에서는 대용량 3차원 포인트 클라우드의 탐색을 위하여 기존 연구에서 제시한 메모리 효율적인 옥트리(Han, 2013)를 기반으로 다중코어를 이용한 병렬 탐색 방법론을 제시하고 평가하였다. 이를 위하여 다중코어를 이용한 병렬 처리를 용이하게 하기 위한 API인 OpenMP을 C++ 환경에서 이용하였으며, for 루틴에서 false sharing을 유발하는 옥트리 노드 내 변수를 노출시켜 병렬 처리의 성능 저하를 방지하였다. 주어진 포인트로부터 일정 반경 내에 존재하는 포인트를 탐색하는 과정을 병렬화하기 위하여 다수의 리프 노드에 병렬로 접근하였으며 OpenMP의 자동화된 for 루틴 분배 방식과 코어간 부하를 고려한 공간적 분배 방식을 비교, 평가하였다. 터널에서 지상 레이저 스캐너로 취득한 1800만개의 3차원 포인트로부터 옥트리를 생성하고 다양한 탐색 반경에 대한 탐색 시간을 측정함으로써 평가를 수행하였다. 8개의 코어가 집적된 CPU를 1개 장착한 시스템에 적용한 결과 두 가지 병렬 탐색 방식 모두 비병렬 탐색 방식보다 수배의 성능 향상 효과를 나타내었으며, 두 방식은 탐색 반경에 따라 서로 경합하는 양

상을 나타내었다. 그러나 공간적 분배 방식이 작은 탐색 반경에 대해서는 우위의 성능을 나타내었으며 전반적으로 안정적인 처리 시간을 기록하였다. 향후 공간적 분배 형상을 개선하여 코어간 부하의 균질도를 제고함으로써 병렬 탐색 속도를 향상시킬 수 있을 것으로 기대된다.

감사의 글

본 연구는 경일대학교 2011년 신입교원정착연구과제의 지원을 받아 수행한 연구임

References

- Chandler, D. (2010), Reduce False Sharing in .NET*, Intel Corporation, URL: <http://software.intel.com/en-us/articles/reduce-false-sharing-in-net> (last date accessed: 4 June 2013).
- Cho, H., Cho, W., Park, J., and Song, N. (2008), 3D building modeling using aerial LiDAR data, *Korean Journal of Remote Sensing*, Vol. 24, pp. 141–152. (in Korean with English abstract)
- Marechal, L. (2009), Advances in octree-based all-hexahedral mesh generation: handling sharp features, 18th International Meshing Roundtable, Salt Lake City, UT, USA, pp. 65–84.
- Han, S., Lee, S., Kim, S. P., Kim, C., Heo, J., and Lee, H. (2011), A Comparison of 3D R-tree and octree to index large point clouds from a 3D terrestrial laser scanner, *Korean Journal of Geomatics*, Vol. 29, No. 1, pp. 531–537. (in Korean with English abstract)
- Han, S., Kim, S., Jung, J. H., Kim, C., Yu, K., and Heo, J. (2012), Development of a hashing-based data structure for the fast retrieval of 3D terrestrial laser scanned data, *Computers & Geosciences*, Vol. 39, pp. 1–10.
- Han, S. (2013a), Design of Memory-Efficient Octree to Query Large 3D Point Cloud, *Korean Journal of Geomatics*, Vol. 31, No. 1, pp. 41–48. (in Korean with English abstract)
- Han, S. (2013b), Enhancing Query Speed of 3D Points Structured by Octree Using Multi-threads, 2013 Spring Conference, Korean Society of Surveying, Geodesy, Photogrammetry, and Cartography, Busan, Korea, pp. 381–

382. (in Korean with English abstract)
- OpenMP ARB (2013), The OpenMP® API specification for parallel programming, *OpenMP Architecture Review Board*, URL: <http://openmp.org/wp/> (last date accessed: 21 May 2013).
- PCL (2013), Point Cloud Library, *Open Perception*, URL: <http://www.pointclouds.org/> (last date accessed: 21 August 2013).
- Saxena, M., Finnigan, P. M., Graichen, C. M., Hathaway, A. F., and Parthasarathy, V. N. (1995), Octree-based automatic mesh generation for non-manifold domains, *Engineering with Computers*, Vol. 11, pp. 1–14.
- Schnabel, R., Wahl, R., and Klein, R. (2007), Efficient RANSAC for point-cloud shape detection, *Computer Graphics Forum*, Vol. 26, pp. 214–226.
- Tian, T. and Shih, C-P (2012), Software Techniques for Shared-Cache Multi-Core Systems, Intel Corporation, URL: <http://software.intel.com/en-us/articles/software-techniques-for-shared-cache-multi-core-systems> (last date accessed: 4 June 2013)
- Wang, M. and Tseng, Y.-H. (2004), Lidar data segmentation and classification based on octree structure, XXth ISPRS Congress, ISPRS, Istanbul, Turkey.
- Wikipedia (2010), Schematic drawing of an octree, a data structure of computer science, Wikimedia Foundation, Inc., URL: <http://en.wikipedia.org/wiki/Octree> (last date accessed: 20 January 2013).
- Wikipedia (2012), C dynamic memory allocation, Wikimedia Foundation, Inc., URL: http://en.wikipedia.org/wiki/C_dynamic_memory_allocation (last date accessed: 20 January 2013).
- Woo, H. ,Kang, E., Wang, S., and Lee, K. H. (2002), A new segmentation method for point cloud data. *International Journal of Machine Tools and Manufacture*, Vol. 42, pp. 167–178.

(Received 2013. 08. 02, Revised 2013. 08. 20, Accepted 2013. 08. 28)