

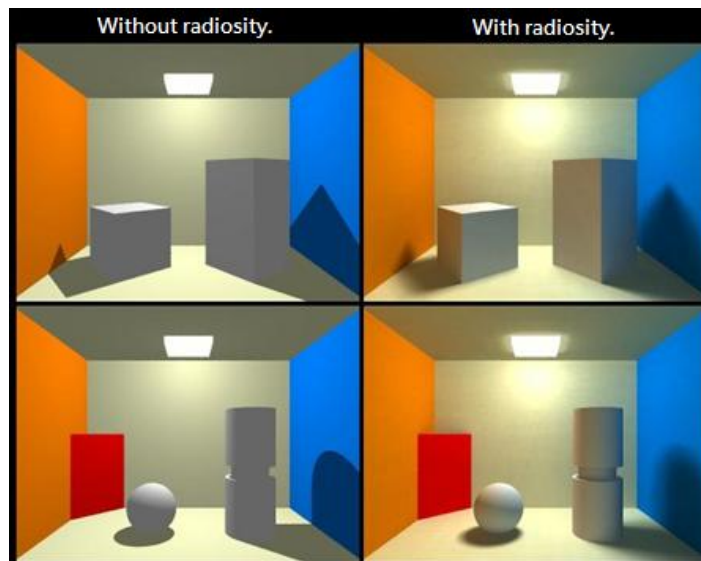
# Parallel Computing and Optimization Skills for Radiosity

R04922067 楊翔雲 資訊工程研究所

R04922133 古君葳 資訊工程研究所

## 介紹 Introduction

熱輻射法 (Radiosity) 是一種渲染的技術。相較於光線追蹤法 (Ray Tracing)，熱輻射法可以產生更接近於現實場景中光亮的變化。如左下圖，當場景使用光線追蹤法時，物體的陰影的邊緣相對銳利，但在現實情況下，物體陰影漸層呈現，因此使用熱輻射法可以更貼近我們想要的。



Radiosity 算法公式如下

$$B_i dA_i = E_i dA_i + R_i \int_j B_j F_{ji} dA_j$$

,  $A_i$  : Area of element  $i$  (computable)

$B_i$  : Radiosity of element  $i$  (unknown)

$E_i$  : Radiant emitted flux density of element  $i$  (given)

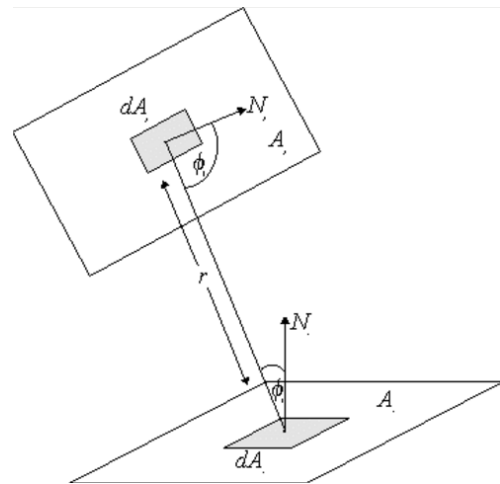
$R_i$  : Refletance of element  $i$  (given)

$F_{ji}$  : Form Factor from  $j$  to  $i$  (computable)

假設整個場景中有  $N$  個三角形，每一次迭代選擇一個最亮的三角形當作光源，由這個光源計算與場景中其他三角形的 Radiosity 之值。其中，判斷光源是否可以輻射到某個三角形之複雜度為  $O(\log N) \sim O(N)$  (視 Data structure 而定)，而計算 Form-Factor 的花費可以視為常數  $O(1)$ ，因此每次迭代的複雜度為  $O(N \log N) \sim O(N^2)$ 。

其中佔據效能的因素是 Form-Factor 估算，因此有像 Hemicube 之類的近似逼近，大幅度減少計算量，但投影回到原本物件上會失真。

$$F_{ij} = \frac{1}{A_i} \int_{A_i} \int_{A_j} \frac{\cos \phi_2 \cos \phi_1}{\pi r^2} dA_i dA_j$$



# 優化技術 Code Review & Optimization

首先，我們先對助教提供的程式碼加速，分成以下幾個部分討論

- 減少光線投射計算量 Strength Reduction for Ray Casting
- 減少 Form-Factor 計算量 Strength Reduction for Form-Factor
- 改善資料局部性 Improve Data Locality
- 其他優化 Other Optimization:
  - Improve I-cache Miss
  - Short Circuit Design
  - Clipping Algorithm
  - Strength Reduction for Float-Point
  - Shrink the Scope of Variables
  - Reduce the Number of Arguments
  - Remove Implications of Pointer Aliasing
  - Copy Optimization

## 減少光線投射計算量 Strength Reduction for Ray Casting

判斷射線 (Ray) 是否能打到三角形  $A$  上，先用 bounding box 包住  $A$ ，計算  $p$  到 bounding box 的時間  $t$ ，若  $t$  大於目前的最小  $t_{\min}$ ，則退出。相反地，再計算更精準的  $t$ 。加入利用已知結果  $v = p + t_{\min} \cdot d, t_{\min} > 0$

```
int TriangleHitted(Vector p, Vector d, TrianglePtr tp, float *t_min) {  
    float t = /* time t from p to bounding box of Triangle tp */;  
    if (t < eps)  
        return false;  
    if (t >= *t_min) /* important !! */  
        return false;  
    /* ... */  
    *t_min = t;  
    return true;  
}
```

## 減少 FF 計算量 Strength Reduction for Form-Factor

根據公式  $F_{ij} = \frac{1}{A_i} \int_{A_i} \int_{A_j} \frac{\cos \phi_2 \cos \phi_1}{\pi r^2} dA_i dA_j$ ，一般我們的判斷順序會得如下：

```
float computeFormFactor(TrianglePtr srcTri, int logSrc, TrianglePtr
desTri, int logDes, Vector p) {
    Vector dir = srcTri->c - p;
    float ff = 0;
    if (RayHitted(p, dir, logDes) == logDes) {
        float theta1 = CosTheta(dir, srcTri->normal), theta2 =
CosTheta(dir, desTri->normal);
        ff = theta1 * theta2 * srcTri->area / (norm2(dir) * PI);
    }
    return max(ff, 0.f);
}
```

效能考量的因素：

- RayHitted() 需要大量的計算
- Form-Factor 在 float 儲存格式下可能無法得到貢獻，  
改採優先計算 Form-Factor 的值，再運行 RayHitted 判斷。調整加速了 2 倍多。

```
float computeFormFactor(TrianglePtr srcTri, int logSrc, TrianglePtr
desTri, int logDes, Vector p)
{
    Vector dir = srcTri->c - p;
    float theta1 = CosTheta(dir, srcTri->normal),
        theta2 = CosTheta(dir, desTri->normal);
    float ff = theta1 * theta2;
    if (ff <= 0) return 0.f;
    ff *= srcTri->area / (norm2(dir) * PI);
    if (ff <= 0) return 0.f;
    if (RayHitted(p, dir, logDes) == logDes)
        return ff;
    return 0.f;
}
```

## 改善資料局部性 Improve Data Locality

程式碼中使用 3D-DDA Line Algorithm/Bresenham's Line Algorithm 搭配 Octree，在找尋某個射線與哪個最近三角形相交。

- 只需要儲存葉節點代表的立方體中，所有可能相交的三角形編號
- 移除掉中間產生的編號，讓每一次 access 的 cache-miss 下降

在 3D-DDA 中，我們需要反查找空間中一點  $p$  在哪一個葉節點中，藉由固定的長寬高切割長度，可以在  $O(1)$  時間內得知  $[i][j][k]$  各別的值。若限制大小，則建立陣列  $[i][j][k]$  查找。若自適應大小，則建立 hash 表查找，但根據實驗結果，效能並沒有改善，因為三角形個數過多導致命中機率過低。

對於 Static Tree 的 Memory Layout，大致上分成四種 DFS Layout、Inorder Layout、BFS Layout、和 van Emde Boas Layout，目前程式使用的是 DFS Layout，這方面會影響到存取的效能。若有更多的時間，我們也可以測試平行處理的細粒度與這些 Memory Layout 的影響。

## 其他優化 Other Optimization

### Improve I-cache Miss

壓縮程式碼長度以改善 I-cache miss，因為大部分的初始化只會運行一次，不應該交錯在時常被呼叫的函數之間，指令載入效能才會提高，同時也要做好 Code Layout，就能改善執行效能。

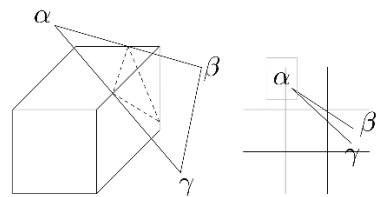
```
x, y = compute(0)
buildTree(x, y)
x, y = compute(1)
buildTree()
x, y = compute(2)
buildTree()
x, y = compute(3)
buildTree()
...
```

```
for i from 0 to n
    x, y = compute(i)
    buildTree()
```

## Short Circuit Design

判斷三角形與一個正交立方體是否相交，使用投影到二維空間中與一條線段的交點是否皆存在。投影方法有 3 種 x-y, y-z, z-x，線段投影共有 2 種，共計 6 種情況。原先程式沒有做好短路設計，只要其中一種不符就應退出。

```
int CrossOver(TrianglePtr tri,
               Vector g0, Vector g1) {
    for (xyz = 0; xyz < 3; xyz++) {
        // front face project
        if (!test())
            return false;
        // back face project
        if (!test())
            return false;
    }
    return true;
}
```



## Clipping Algorithm

我們實作課程中所描述的 Cohen–Sutherland Algorithm 降低 branch 次數，使用 bitwise 操作引出 SSE (Streaming SIMD Extensions)。儘管 compiler -O2 替我們優化，為減少 stack push/pop 的次數，實作時請不要使用的 procedure call，否則會慢上許多。

```

void computeRadiosity(TrianglePtr srcTri, TrianglePtr desTri,
float ff[3]) {
    char mask1 = (p0[x] < g0[x])<<0 |
        (p0[x] > g1[x])<<1 |
        (p0[y] < g0[y])<<2 |
        (p0[y] > g1[y])<<3 ;
    char mask2 = (p1[x] < g0[x])<<0 |
        (p1[x] > g1[x])<<1 |
        (p1[y] < g0[y])<<2 |
        (p1[y] > g1[y])<<3 ;
    if (mask1&mask2)
        return false;
    if (!(mask1|mask2))
        return true;
    // ... test

```

## Strength Reduction for Float-Point

兩個外積結果相乘小於零，減少 instruction cycle 量，盡量用整數作為運算型態。

<pre> float a = cross(/* */); float b = cross(/* */); if (a * b &lt; 0)     return false; b = cross(/* */); if (a * b &lt; 0)     return false; ... </pre>	<pre> int a = cross(/* */) &lt; 0; int b = cross(/* */) &lt; 0; if (a != b)     return false; b = cross(/* */) &lt; 0; if (a != b)     return false; ... </pre>
--	---

## Shrink the Scope of Variables

減少變數生命週期的長度以增加放入暫存器的機會，而非 stack 上。

<pre>float rgb[3];  for (int i = 0; i &lt; 3; i++)     rgb[f(i)] = g(i); /* ... */ if (maybe) {     for (int i = 0; i &lt; 3; i++) {         rgb[h(i)] = g(i);     }     /* ... */ }</pre>	<pre>{     float rgb[3];     for (int i = 0; i &lt; 3; i++)         rgb[f(i)] = g(i);     /* ... */ } if (maybe) {     float rgb[3];     for (int i = 0; i &lt; 3; i++) {         rgb[h(i)] = g(i);     }     /* ... */ }</pre>
--	---

## Reduce the Number of Arguments

減少 stack push/pop 次數

<pre>struct Arg {     int a0, a1; };    // p1.a1 = p2.a0 int f(Arg p1, Arg p2) {     /* ... */ }</pre>	<pre>struct Arg {     int a0, a1, a2; }; int f(Arg p1p2) {     /* ... */ }</pre>
--	--

## Remove Implications of Pointer Aliasing

移除指標 Aliasing，意指可能會指向相同記憶體位址，導致每次計算都要重新載入，不能放進暫存器中。如下述的寫法，編譯器無法判定 srcTri 是否與 desTri 相同，在累加時則重新載入 srcTri->deltaB[] 的數值，計算上可能會產生數次的 cache miss，隨著迴圈次數不斷突顯效能差異。



```

void computeRadiosity(TrianglePtr srcTri, TrianglePtr desTri,
                     float ff[3]) {
    for (int v = 0; v < 3; v++) {    // vertex
        for (int c = 0; c < 3; c++) {    // color RGB
            float deltaB = desTri->Frgb[c]/255.0*RefRatio*srcTri->Frgb[c]*ff[v]/3;
            desTri->deltaB[c] += deltaB;
            desTri->accB[v][c] += deltaB;
            desTri->deltaAccB[v][c] += deltaB;
        }
    }
}

```

- **方法 1:** 加入 `if (srcTri != desTri)` 判斷，讓編譯器在 Function Pass 階段著手的 Dependency Analysis 更好
- **方法 2:** 使用 Copy Optimization，同時把重複計算搬到 stack 上，或者使用 Polyhedral 表示法進行 Reordering Accesses for Loop Nest。這裡我們選擇前者，更容易引出 SSE。

```

void computeRadiosity(TrianglePtr srcTri, TrianglePtr desTri,
                     float ff[3]) {
    const float k = RefRatio / 255.0;
    float lo[3] = { desTri->Frgb[0]*k*(srcTri->deltaB[0]),
                    desTri->Frgb[1]*k*(srcTri->deltaB[1]),
                    desTri->Frgb[2]*k*(srcTri->deltaB[2]) };
    for (int v = 0; v < 3; v++) {    // vertex
        for (int c = 0; c < 3; c++) {    // color RGB
            /* calculate the reflectiveness */
            float deltaB = lo[c] * ff[v] / 3;
            desTri->deltaB[c] += deltaB;
            desTri->accB[v][c] += deltaB;
            desTri->deltaaccB[v][c] = deltaB;
        }
    }
}

```

## Copy Optimization

我們可以考慮使用 Copy 的方式放入較近的 stack 上，而非傳遞指標。將資料放得近可提升 cache 使用率，隨著使用次數增加而明顯。當運行次數多，才能讓 data reused 的比例上升，意即 n 要夠大。

```
int f(const Triangle *from) {  
    for (int i = 0; i < n; i++)  
        compute(from, i);  
}
```

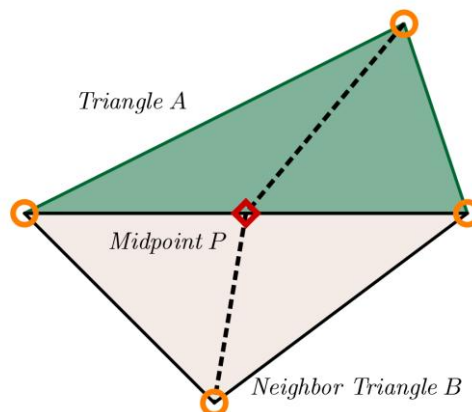
```
int f(const Triangle *from) {  
    Triangle tmp = *from;  
    for (int i = 0; i < n; i++)  
        compute(tmp, i);  
}
```

## 平行設計 Parallel Design

回顧原始算法，為了加快收斂速度，每一次會挑選單位面積能量最多的三角形  $f$ ，之後便拿所有三角形  $t$  進行傳導，直到傳導能量小於閾值，迭代將停止。算法如下所述：

```
while (not converge) {  
    f = PickMaxRadiosityTriangle()  
    foreach triangle t in model  
        shader(f, t);  
    clearRadiosity(f)  
}
```

在嘗試傳遞的過程中，若三角形  $t$  的三頂點的能量差異大，則選擇自適應切割三角形，直到三頂點能量差異小，這麼計算 Form-Factor 才會正確。在自適應部份，切割方法如下圖所示：



當偵測到綠色三角形 A 頂點之間的 Form-Factor 差異過大時，使用最長邊的中點切割，這麼做是盡可能產生銳角三角形，為了圖形完整，必然也要對鄰居切割。為減少計算量，只算新增中心點 P 的 Form-Factor。對於下方的三角形 BB 而言，分成兩種情況，已在這一輪完成計算，則重新計算 Form-Factor；相反地，不做任何事。

## Single-Source Parallel Algorithm

我們發現計算 Form-Factor 相當獨立的，但自適應處理需要遞迴切割，因此選用多核心平台，而非通用圖形處理器平台，因為目前的 GPU 實作遞迴所需的 stack 使用 global memory 作為存取位址，所以一般多核心平台效果會更

好。在這一次報告中，我們選用 OpenMP 這套跨平台多執行緒 API 進行實驗。

著手將多個三角形平行處理，意即每一個執行緒負責多個三角形的 Form-Factor 計算。

```
// Single-Source Parallel Algorithm
while (not converage) {
    f = PickMaxRadiosityTriangle()
    parallel foreach triangle t in model
        shader(s, t);
    clearRadiosity(f)
```

## Multi-Source Parallel Algorithm

從原始算法中，我們發現到每一次迭代將只有一個熱源輻射到場景中，當場景有多個高能量熱源時，場景必須經過好幾次迭代才能近似最終結果。藉由平行處理的效能，我們可以一次迭代多熱源，便可將低執行緒之間分配工作不均的情況，不僅僅前幾次迭代就能近似最終結果，同時也能加速運算。

```
// Multi-Source Parallel Algorithm
while (not converge) {
    set<Triangle> f = RadiosityTriangleCandidateCandidate();
    parallel foreach triangle t in model
        if (f.find(t))
            continue;
        foreach s in f
            shader(s, t);
    clearRadiosity(f);
}
```

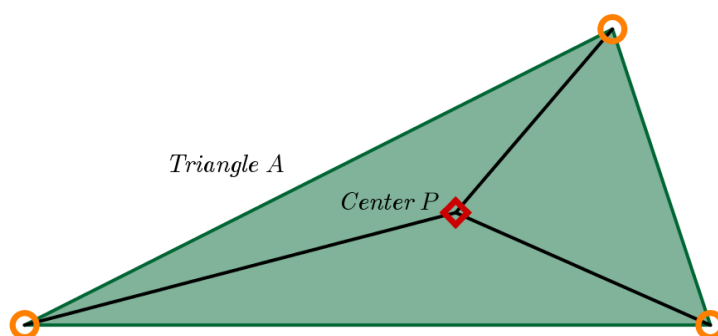
我們所用的平行方無法搭配上上述自適應的切割方案，其原因在於分裂過程中，同時也要對鄰居三角形分裂，整個圖形產生的節點與邊的關係才會正確，無法保證鄰居在同一執行緒內處理，若沒有做好空間切割，我們便無法處理這

部份。若模型格式會是數個獨立的物體，而非單純的三角形資訊，可分配每一個執行緒處理多個獨立物體，我們預期可以達到更好的效果。

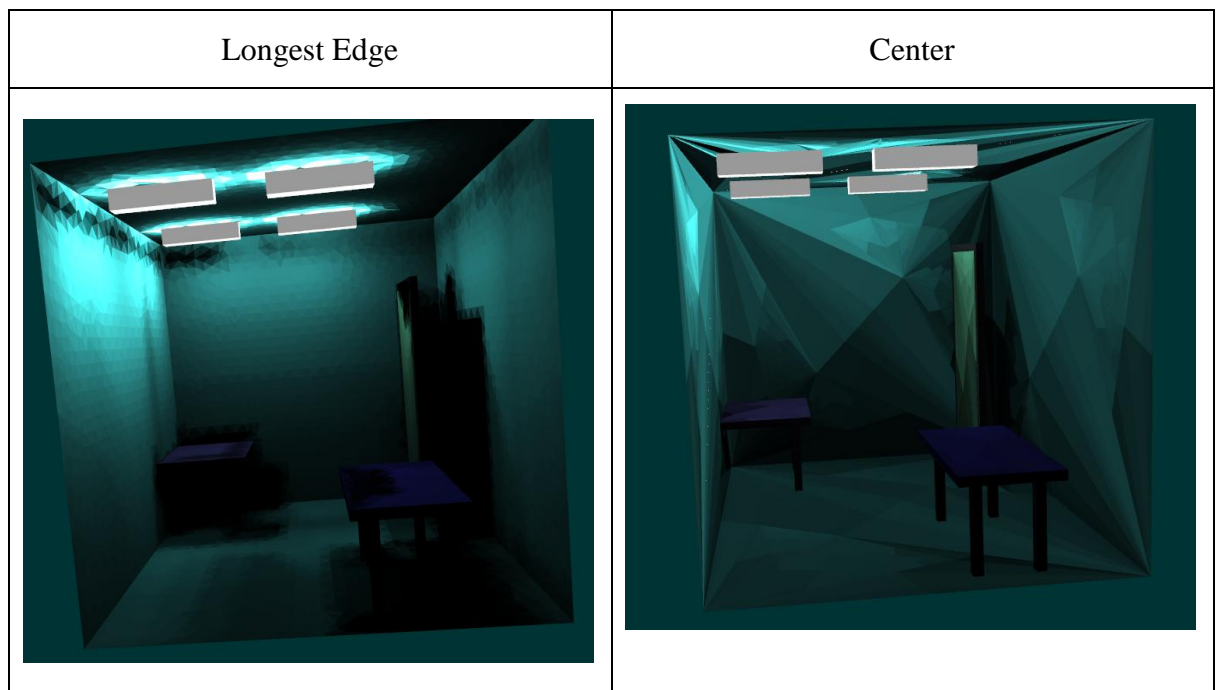
平行過程中每一個執行緒共享和衝突的區段越少越好，這意味著我們必須在運行輻射前就必須將模型切得相當細緻。特別注意到，切得細緻與否對於光線投射 (Ray Casting) 複雜度不變，因為邏輯上他們處理同一平面。

一旦切得細緻，傳遞的效果就不是這麼好，在邊界的陰影更加顯著。根據理論和實作層面推測，其一原因是能傳遞的總能量隨著迭代減少，那麼從分裂過程中傳遞能量採用較多的加法完成，相較於多個 32-bit floating point 誤差就少了許多。

我們也試著使用獨立的切割方案—重心切割，切割的結果不依賴鄰居，只需要在加入三角形清單部份使用 critical section 即可，效能影響並不大。



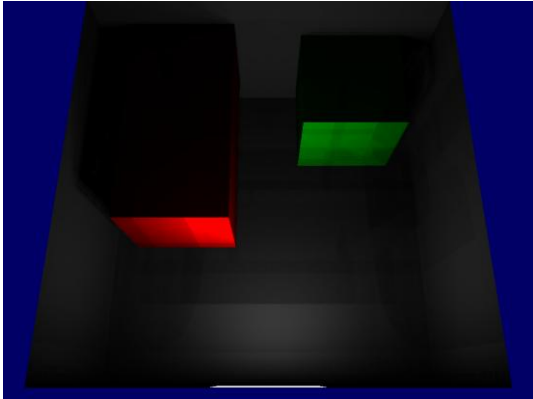
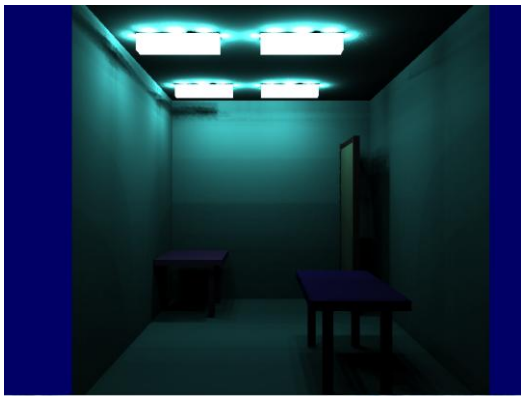
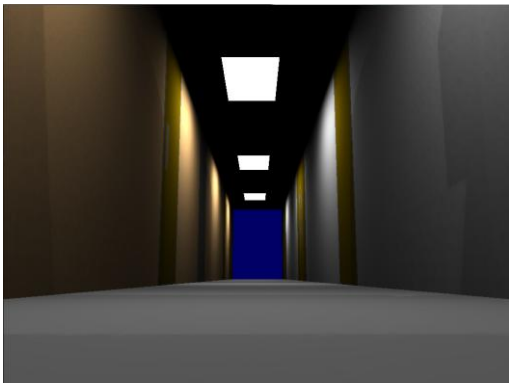

根據重心切割，下述實驗中，從 156 個三角形，自動分裂到 30000 個三角形後進行輻射的結果如下圖所示：



明顯地，根據重心的切割方法容易產生鈍角三角形，看起來就會像很多紡錘體。在眾多數學性質中，只使用重心也許不是好的解決方案，這是值得探討的一部份，由於製作上的時間限制，我們並沒有去探討各個不同切割方案，所對應的自適應的效果如何。

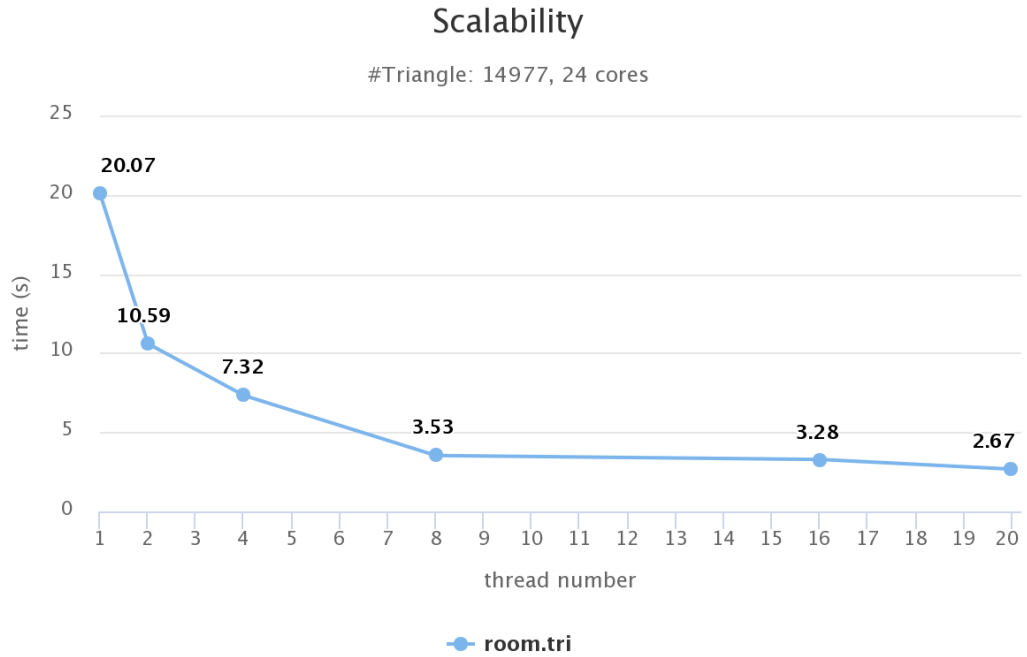
## 展示結果 Demo

只使用優化技術渲染結果

blocks	room
	
hall	church
	

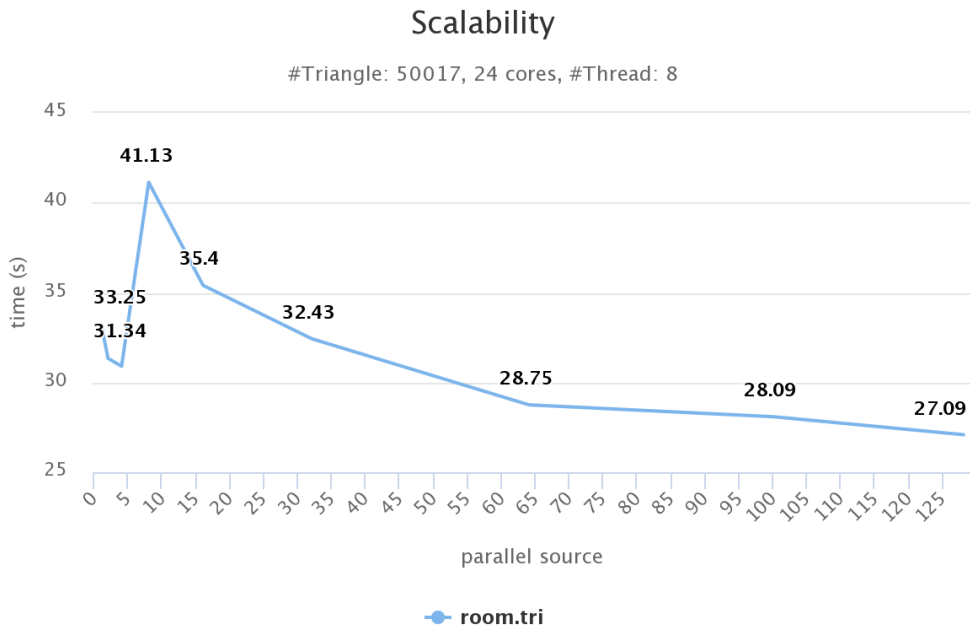
## 平行效能比較

在 Intel Xeon E5-2620 v3 上，我們測試不同平行度帶來的影響，由於只有兩個實體 CPU，每一個 CPU 有 6 個核心，每個核心皆有 Hyper-threading 技術，故可產生 24 個執行緒。



Highcharts.com

我們對模型 room.tri 以預先切割 14977 個三角形後，根據先前提到的平行算法 Single-Source Parallel Algorithm，即是迭代一次只取一個熱源，平行計算所有三角形到此熱源的 Form-Factor 值，針對不同的執行緒個數和運行時間記錄，結果如上圖。在由於過多的執行緒可能會帶來更多的 false sharing，造成資料在不同的 CPU 之間運行 data transmission，所以效果就逐漸不明顯。



Highcharts.com

接著，我們測試 Multi-Source Parallel Algorithm，以 room.tri 預先切割



50017 個三角形後，每次迭代皆取數個熱源，平行計算所有三角形到所有被選取熱源的 Form-Factor 的總和。同樣地，因為查找的資料重複存取的模式造成不好的影響，類似上述所提到的 false sharing 影響，故會呈現一種陡坡。

## Reference

- Fast conversion of floating-point values to string:

<http://0x80.pl/notesen/2015-12-29-float-to-string.html>

我們使用這一篇討論中，在輸出 Triangle 格式部份加速浮點數轉文字加快 4 倍速。

## 原始碼 Source Code

<https://github.com/morris821028/hw-radiosity>