Parallelizing ACO algorithm

楊尊宇 0316328, 王威斌 0316081, 駱昭旭 0312203

***Abstract*—螞蟻演算法可用於尋找優化路徑，又因其連續運行並適應即時變化，特別適合用於最佳路徑規劃。然而，隨著道路的增加與城市交通複雜化，螞蟻演算法應用於路徑規劃的速度越來越慢。因此，本專題利用pthread, OpenMP, CUDA, SSE 等平行加速技巧，希望能讓螞蟻演算法更快速地找到優化路徑。**

Keywords—ACO algorithm; Optimal path; pthread; OpenMP; CUDA; SSE

I. INTRODUCTION

螞蟻演算法最早在1996年由Dorigo, Maniezzo, 和 Colorni 提出[5]，透過模仿自然界螞蟻在尋找食物時的移動行為，用以解決複雜的優化問題。最初，是用來解決旅行業務員的問題(Traveling Salesman Problem, TSP)，而後期也漸漸應用於許多組合優化問題，如車間作用調度問題、多站車輛路徑問題、二次分配問題、最佳路徑規劃、運輸繞徑問題與網格工作流調度問題等。

對宅配業者來說，最佳路徑規劃尤其重要，隨著電子商務的興起與高油價的時代來臨，宅配業者面臨了極大的衝擊與挑戰，為了有效降低經營成本與提升顧客滿意程度，許多業者會以螞蟻演算法來最佳化宅配路線。然而，現今交通系統錯綜複雜，相同道路在不同時段的路況不盡相同，也使得最佳路徑有所不同。因此，使用螞蟻演算法找出最佳路徑所花費的時間將大幅提升。本專題藉由平行化以加速螞蟻演算法，希望能應對不同交通狀況即時算出最佳宅配路徑，以達到最低運輸成本。

為了證明上述文字所表示的最佳路徑，我們利用所寫的程式，模擬出三種情況，分別為未塞車情況、塞車情況、路程遠近情況，而 Fig1、Fig2、Fig3 為此三種情況的示意圖。需要另外提及的是，Fig0為程式剛執行的情況，由此可見，螞蟻尚未收斂成最短路徑，只要給予足夠時間，螞蟻便會如Fig1、Fig2、Fig3中的荷爾蒙的分布狀況，收斂成一條最短路徑。

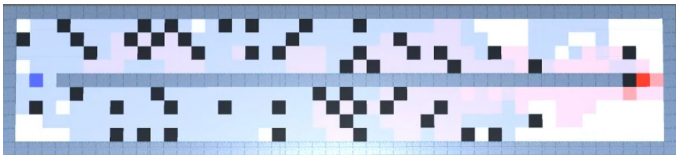

Fig0, 未收斂

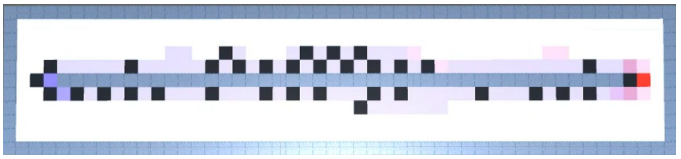

Fig1, 未塞車情況，已收斂

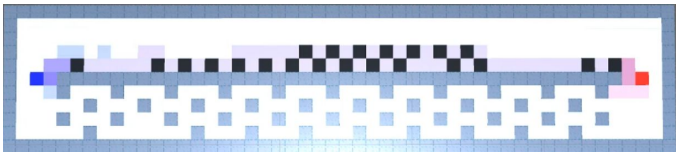

Fig2, 塞車情況，已收斂

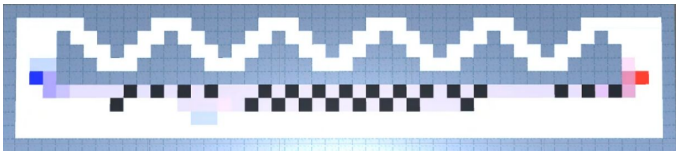

Fig3, 路程不同遠近情況，已收斂

視覺化的部分是利用 Unity 寫成。Fig1 是模擬未塞車情況，當上條道路與下條道路到終點的距離相等，且路上並無塞車時，螞蟻會以同樣機率的前提下在兩條道路上各自收斂成一條線，代表兩條道路皆是最佳路徑。然而，今假設下條道路塞車且上條道路沒塞車，如 Fig2 所示，則起始點走下條道路到終點的距離，會比從起始點走上條道路到終點的距離還要長，因此，螞蟻便會傾向走上條未塞車的道路，所以此情況上條道路則為最佳路徑。最後，Fig3顯示了上條道路比下條套道路還要遠的情況，可以很明顯地看出，螞

蟻傾向走下條道路，距離終點較近，在此情況，下條道路為起點到終點的最佳路徑。

螞蟻演算法的核心在於模仿螞蟻尋找食物的特性，並找出起點到終點的優化路徑。起初，每隻螞蟻都會從起點(巢穴)開始爬行以尋找食物，尋找食物的螞蟻會在已走的路徑上留下荷爾蒙，稱為藍色荷爾蒙，此種藍色荷爾蒙會給予已經找到食物的螞蟻一條回家的路。相對的，已經找到食物的螞蟻會沿路散發另一種荷爾蒙，稱為紅色荷爾蒙，而此種紅色荷爾蒙會給予給正在找食物的螞蟻一條找到食物的路徑。因此，對於來回走一趟取得食物的螞蟻而言，長路徑需要較多的時間完成，代表該路徑上的荷爾蒙濃度隨時間遞減地越多，長時間下來，長路徑上的荷爾蒙濃度將相較於短路徑上的荷爾蒙濃度低許多。根據此趨向，螞蟻會偏向走短路徑，在地圖上便會漸漸形成一條優化路徑。

為了加速上述的螞蟻演算法，本專題將演算法流程分成三個部分(Fig.4)，由左至右分別為：*螞蟻在地圖上留下荷爾蒙*、*螞蟻決定移動方向*、*地圖上荷爾蒙揮發*。最後會有一個依據螞蟻所走的步數，來決定程式是否要終結。其中加速的技巧共嘗試了pthread、OpenMP、CUDA、SSE 等平行處理技巧。

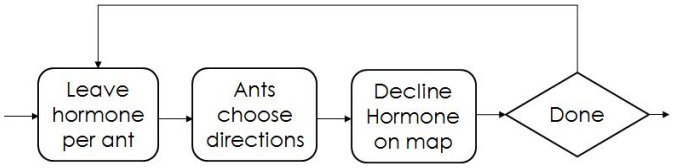

Fig4, 螞蟻每輪的移動行為

此外，因 SSE 較為特別，故在此特別作介紹。SSE(Streaming SIMD Extensions) 是 Intel 一個 SIMD 的指令集，是繼 MMX 的擴充指令集。SSE 有 8 個 128 位元的暫存器，如下圖 Fig5 所示。因此，如果今天暫存器中要存入浮點數，為 32 位元，則每個暫存器共可存入 4 個不同的浮點數值。

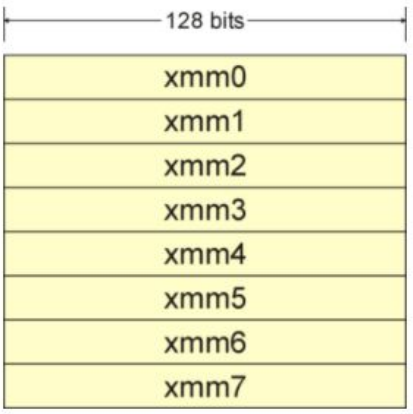

Fig5, SSE的暫存器

而 SSE 的運算行為如 Fig6 所示。先把要運算的數值分別存入暫存器中，再分別對兩個暫存器做運算，如上述例子，如果對兩個存有浮點數的暫存器進行加法運算，則只需要一個加法的時間，便可得到 4 個浮點數加法的結果。

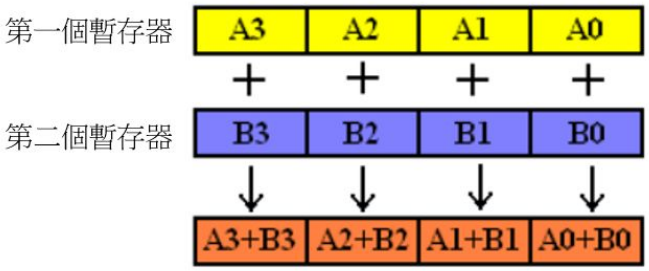

Fig6, SSE運算行為

II. PROPOSED SOLUTION

本程式的行為共分為三個部分，如 Fig.4 所示，由左至右分別為: *每隻螞蟻在地圖上留下荷爾蒙*、*每隻螞蟻決定下一步行進方向*、*地圖上所有的荷爾蒙發散*，以下將針對各個部分的運算行為進行更深入的介紹與說明平行加速的方法。

A. 每隻螞蟻在地圖上留下荷爾蒙

在本程式中，為了紀錄螞蟻沿路所發散的荷爾蒙數值，會有兩個浮點數的三維陣列分別儲存空間地圖上的藍色荷爾蒙與紅色荷爾蒙，在每個時刻，每隻螞蟻皆會根據自己所在的位置，疊加地圖上對應的荷爾蒙濃度。

然而，在演算法執行的過程中，有可能會有兩隻以上的螞蟻走到同一位置上，如果用多執行緒的方式進行平行化加速，會有競爭資源的問題

，必須使用互斥鎖(mutex)保護，但互斥鎖必須保護整張地圖上每一個點，使用的記憶體過高，此外螞蟻的數量並不多，若用單執行緒的方式依序更新螞蟻所在位置的賀爾蒙濃度所花費的時間並不多。因此，經過評估，此部分便只用單執行緒對每隻螞蟻在地圖上的位置進行荷爾蒙累加。

B. 每隻螞蟻決定下一步行進方向

每隻螞蟻會根據地圖上的荷爾蒙來決定下一步的方向，以2D空間為例，因為地圖是方格狀，所以每隻螞蟻共有前、後、左、右四種方向可作選擇。假設某隻螞蟻的位置在(i, j)，且h(a, b)為位置(a, b)所殘留的荷爾蒙值，P(i, j+1)代表螞蟻到位置(i, j+1)的機率，則該隻螞蟻針對每個方向所行進的機率為

$$P(i+1,\, j) = \frac{h(i+1,\, j)}{h(i+1,\, j)+h(i,\, j+1)+h(i-1,\, j)+h(i,\, j-1)}$$
$$P(i,\, j+1) = \frac{h(i,\, j+1)}{h(i+1,\, j)+h(i,\, j+1)+h(i-1,\, j)+h(i,\, j-1)}$$
$$P(i-1,\, j) = \frac{h(i-1,\, j)}{h(i+1,\, j)+h(i,\, j+1)+h(i-1,\, j)+h(i,\, j-1)}$$
$$P(i,\, j-1) = \frac{h(i,\, j-1)}{h(i+1,\, j)+h(i,\, j+1)+h(i-1,\, j)+h(i,\, j-1)}$$

此部分的行為是每隻螞蟻從荷爾蒙地圖讀出該位置四周的荷爾蒙值，因此，本部分不會有寫值，就不會有資源競爭的問題，可用多執行緒的方式平行化，讓每條執行緒均勻分攤所有螞蟻選擇方向的工作。至於為何此部分不使用顯示卡加速，是因為程式在計算完每個方向的機率後，會隨機產生一個值以決定前往哪個方向，而決定方向是使用條件判斷式決定，再加上確定方向後，程式會確認是否已經到了起點或是終點，這又需要兩個條件判斷式。因此，我們認為這邊需要的條件判斷式過多，加上螞蟻數目也不多，用顯示卡加速可能主要花費時間都在傳輸資料上，故沒有使用顯示卡資源做加速。

C. 地圖上所有的荷爾蒙發散

如上所提過，程式中有兩個三維空間的陣列，分別記錄著地圖上每個地方的藍色與紅色荷爾蒙值。而隨著時間的推移，兩個地圖上每個荷爾蒙值皆會乘上一個小於1的參數，如下條算式所示。

For all i, j bigger than 0, *smaller than the size of map*

$h(i,\, j) = h(i,\, j) * c$, *where c is a positive constant* < 1

因為其簡單的運算行為加上大量的資料，此部分可有很多種平行加速的方法，而這邊我們使用了pthread、OpenMP、CUDA、SSE 嘗試進行加速，但使用 CUDA 或SSE 時，皆只有一核心在控制，其他核心處於空閒狀態，為了不要有浪費運算資源的情況，我們也結合了 pthread 與 CUDA，和 pthread 與 SSE，分別進行加速。

III. Experimental Methodology

本次的實驗我們使用了課程所提供的平台，sslab，處理器為 Intel(R) Core(TM) i7 CPU @ 2.93GH，為四核心八執行緒，顯示卡為NVidia 1060，作業系統是 Ubuntu 16.04，記憶體大小為8GB，另外，我們也有使用 Intel SIMD 指令集(SSE)。

關於整個程式的參數，有很多可以設定，如荷爾蒙值的遞減率、每隻螞蟻所散發的荷爾蒙量、螞蟻喜好荷爾蒙程度等等，但上述的參數皆不會影響到執行時間，真正會影響到執行時間的參數只有三項。第一個是整張地圖的大小，這邊使用1000*1000*10。第二個是螞蟻的數量，設定為10000隻。第三項是設定每隻螞蟻要走幾步才終結程式，原先測試時，大約需要接近100000步才會收斂，但後來發現執行時間過長，等待結果的時間會變很久。而從 Fig4 的演算法流程圖可以看到，步數會跟整體程式的執行時間呈現線性關係，因此，程式最後的設定步數為1000步，執行時間比原先會收斂的100000步快了100倍，提升了不少測試時間的效率。

最後，我們測試時間的方式，藍色和紅色地圖皆是 1000*1000*10 的浮點數陣列、螞蟻數為10000隻的前提下，讓每隻螞蟻跑了1000步後所測試的時間。因為每次的測試時間都有浮動，所以在本文所呈現的數據皆是測試五次後的平均。

IV. Experimental Results

在開始展示平行化後的數據前，表1為原程式沒有加速的數據。

	執行時間 (sec)	所佔比例
每隻螞蟻在地圖上留下荷爾蒙	0.54	0.8%

每隻螞蟻決定下一步行進方向	9.52	11.9%
地圖上所有的荷爾蒙發散	69.53	87.3%,

表1, 原程式的花費時間

第一部分，*每隻螞蟻在地圖上留下荷爾蒙*，並沒有進行平行化，而第二部分，*每隻螞蟻決定下一步行進方向*，我們分別使用了pthread 和 OpenMP。結果如下圖Fig5呈現。

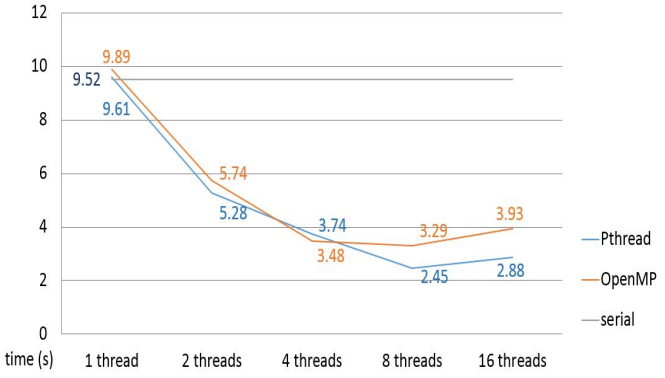

Fig5, *每隻螞蟻決定下一步行進方向* 的加速結果

由 Fig5 可見，當執行緒的數量小於等於 4 的時候，pthread 和 OpenMP 加速的結果差不多，數值相差不大。但是當執行緒的數量大於 8 的時候，可看見 pthread 和 OpenMP 的結果便有了較大的差異，會有這樣的結果是因為程式中 OpenMP 的設定是 dynamic scheduling，再加上螞蟻的數量不多，只要執行緒數量一多，還要去動態去分配每個執行緒工作，這樣花費的成本太高，而 pthread 的寫法是一開始就分配完了所有的工作量，並非動態分配，所以才會造成 OpenMP 的執行時間比 pthread 還要高。為了驗證想法，我們另外去測試了8個執行緒與16個執行緒 OpenMP的 static scheduling，發現結果如表2所示，static scheduling OpenMP 的執行時間介於 pthread 與 dynamic scheduling OpenMP 之間，從 static 和 dynamic 的觀點來看，dynamic 確實會讓執行時間變長，但表2 中也可以明顯的發現，pthread 與 static OpenMP 的執行時間有一段落差，原因可能是因為本部分的計算行為過於複雜，所以 OpenMP 的優化比較沒那麼好，使得 pthread 的效率比 OpenMP 高。另外，Fig5 中顯示，當有8條執行緒時的執行速度比有4條執行緒時還要快，此情況是合理的，因為本專題使用的處理器支援 hyper-threading 技術，故在一個核心上可跑兩個執行緒，所以 8 條執行緒時的執行速度確實會比 4 條執行緒時還要快，但是當執行緒增加到 16 條時，因為硬體上處理器只有 4 核心，故能同時執行的執行緒數最多只有 8 條，而16條執行緒已經超過能同時執行的最大值，故效率會降低。

	pthread	OpenMP (dynamic)	OpenMP (static)
8 threads execution time(sec)	2.45	3.29	2.98
16 threads execution time(sec)	2.88	3.93	3.59

表2, 比較OpenMP static scheduling與其他數據

接下來顯示的是第三部分的平行加速成果，*地圖上所有的荷爾蒙發散*，首先比較的是pthread、OpenMP、SSE的執行速度比較圖，如下圖Fig6所示。

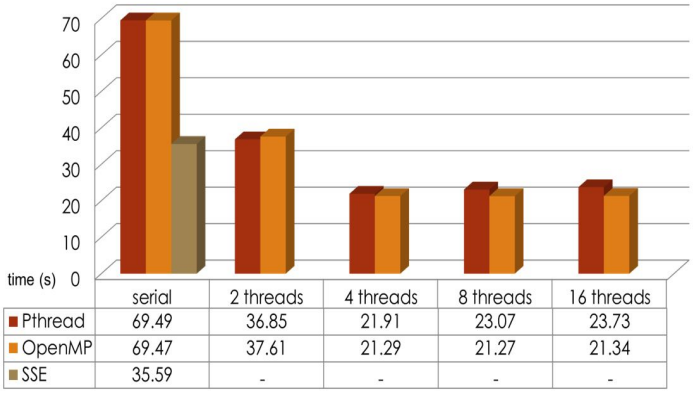

Fig6, *地圖上所有的荷爾蒙發散*, pthread、OpenMP、SSE的執行速度比較圖

首先看到當執行緒只有 1 條時，也就是連續版本的程式，執行時間為 69.48 秒左右，而使用 SSE的話，執行時間可降到 35.59 秒。但理論上使用 SSE 應可提升 4 倍的速度，但這邊指提升了約 2 倍的速度，原因是為了使用 SSE 指令集，必須先將資料搬到 SSE 的暫存器中，算完後

還要將暫存器的值搬出來，造成了overhead，所以速度並不能如理想值而增加 4 倍。為了證明上述講法，我們實測了使用 SSE 時將浮點數搬運到 SSE 暫存器和從 SSE 暫存器搬回記憶體的時間，如表3所顯示，發現光 SSE 進行傳輸就需要 17.48 秒。因此使用 SSE 才無法有 4 倍的加速效果。

	傳輸時間 (sec)	總共時間 (sec)
SSE	17.48	35.59

表3, 測試 SSE 傳輸時間

與先前部分類似，pthread 跟 OpenMP 在執行緒數量分別為 1、2、4 時，執行時間皆差不多，而當執行緒數量大於等於 8 時，pthread 和OpenMP 的數據開始有比較大的變化，但與之前不同的是，這邊的 pthread 執行時間比 OpenMP 還要長。而我們認為會有這樣的結果，原因是讓兩張地圖上所有的荷爾蒙發散是很大計算量，在這樣的前提下，OpenMP 的 dynamic scheduling 就有它的價值，可讓原本的工作量做動態規劃，讓某些執行緒不會因先做完而等待，讓每個執行緒皆可被利用，進而提升執行速度。為了驗證想法，跟上述一樣，我們分別測試了8條執行緒和16條執行緒的 OpenMP 的 static scheduling，結果如表4所顯示，static scheduling 的 OpenMP 與 pthread 的執行時間差不多，故驗證了我們的想法。另外，從 Fig6 可得知，從 4 條執行緒提升到8條執行緒時，執行速度不升反降，與 Fig5 從 4 條執行緒提升到 8 條執行緒的結果不同。這邊速度降低的原因是因為 hyper-threading 技術的關係。從 hyper-threading 技術可知，為了達到超執行緒的效果，處理器上會多增加一個邏輯處理單元，而*每隻螞蟻決定下一步行進方向*，裡面有不少的邏輯判斷式，所以對超執行緒來說是可以同時運算的，然而，在本部分中，*地圖上所有的荷爾蒙發散*，運算行為皆只是單純的將每一個浮點數去乘上一個浮點數常數，但是在 hyper-threading 技術中，浮點運算單元並未增加，造成就算有 hyper-threading 的技術，但是在一核心上兩個執行緒還是只能輪流等待浮點運算單元，才會造成本部分從 4 條執行緒提升到 8 條執行緒時，執行時間變多的原因。

	pthread	OpenMP (dynamic)	OpenMP (static)
8 threads execution time(sec)	23.07	21.27	22.56
16 threads execution time(sec)	23.73	21.34	23.24

表4, 比較 OpenMP static scheduling 與其他數劇

另外，第三部分的平行加速，*地圖上所有的荷爾蒙發散*，除了pthread、OpenMP、SSE外，本專題也使用了CUDA進行加速。如 Fig7 所示。

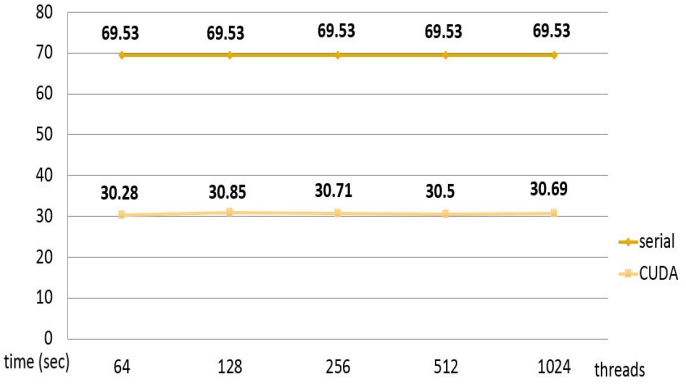

Fig7, *地圖上所有的荷爾蒙發散* 使用CUDA加速

可發現使用 CUDA 加速，不管調動的執行緒數量為多少，速度皆差不多。因此，我們去察看了資料傳輸時間與資料處理時間，發現如表5所示。可發現大多數的時間都花在傳輸資料，真正的運算所花的時間只占了少數，因此就算調整到適合數量的執行緒，能加速的空間也不大。

	傳輸資料 (sec)	運算資料 (sec)
CUDA	28.37	2.13

表5, 比較CUDA傳輸資料與運算資料的時間差異

而 Fig8 是比較*地圖上所有的荷爾蒙發散*全部個別最快的平行方法。

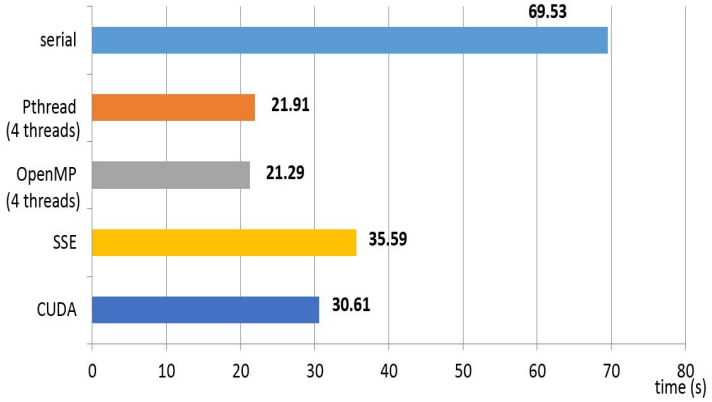

Fig8, *地圖上所有的荷爾蒙發散* 各個最快的平行方法

得到以上數據後，我們認為，單純用 CUDA 或 SSE 算太浪費效能了，畢竟除了讓一核心去控制 CUDA 或 SSE 外，另外三核心處於空閒狀態，因此我們最後提出的方法是結合 pthread 與 CUDA，和結合 pthread 與 SSE 進行加速。在 pthread 方面，我們統一使用4條執行緒，其中一條控制 CUDA 或 SSE，另外3條一起加入運算，而每條執行緒的工作量，會依據方才測試的時間，依照比率的方式分配，所以工作量分配依照下條算式:

(CUDA 或 SSE) : thread : thread : thread
= 2 : 1 : 1 : 1

而執行的結果如下表6 所示，可發現達到了最快速度。

	CUDA+ thread(3條) (sec)	SSE+ thread(3條) (sec)
地圖上所有的荷爾蒙發散	16.04	16.48

表6, CUDA+pthread 與 SSE+pthread 比較表

最後，我們結合了各個部分最快的方法，並與原本未加速的程式進行比較，結果如表7 所示。可發現，原本全部的執行時間為 79.59 秒，加速後最快的執行時間為 20.46 秒，因此加速後的速度快了 79.59 / 20.46 = 3.89 倍。我們嘗試更激進的加速方法，例如使用 CUDA+SSE+pthread 進行結合，但因為要編譯 CUDA 程式的編譯器，nvcc 並不支援編譯 SSE，故無法使用。

	每隻螞蟻在地圖上留下荷爾蒙*(sec)*	每隻螞蟻決定下一步行進方向(sec) (thread= 4)	地圖上所有的荷爾蒙發散 (sec)	全部時間 (sec)
serial	0.54	9.52	69.53	79.59
CUDA+ pthread	0.62	3.8	16.04	20.46
SSE+ pthread	0.6	3.81	16.48	20.89

表7, 最終比較圖

V. Related work

1991年Dorigo et al. [6]在研究中認為螞蟻演算法擁有分散計算、正向回饋的優勢

A. 分散計算

每隻螞蟻皆為獨立個體，雖然在面對相同的荷爾蒙濃度時，所選擇方向會有相似，但可透過分散計算是讓螞蟻由不同起點出發，增加其探索性，避免螞蟻陷於局部最佳化。

B. 正向回饋

正向回饋是自動優化的過程，較多螞蟻行進的路徑，因荷爾蒙濃度較高，將吸引更多螞蟻前往，使得演算法能快速收斂，並找出最佳的路徑。

從最早於1997，Dorigo等人試圖解決旅行業務員的問題[3]，到2008徐嘉吟、黃士滔在蟻群演算法於宅配業路線最佳化之研究[8]，都是利用螞蟻演算法找出最佳路徑。而2016於ACM一篇論文中[7]，討論最佳旅行路徑規劃，給定起點和終點，由系統推薦最大效益的觀光路徑，其推薦系統主要也是由螞蟻演算法構成。

螞蟻演算法的缺點是執行時間較長，尤其是隨著地圖擴大或螞蟻數量增加，運算的時間會巨幅增加，現今的問題結構，對螞蟻演算法來說是極為沉重的負擔，所以平行化螞蟻演算法已成為當前很重要的問題。

當然，除了利用平行化加速執行時間外，設定好對的參數也可以加快收斂的時間，但設定好

的參數需要數學上的推演和證明，不在此專題中進行研究。

VI. Conclusions

經過第III、IV節的實驗後，得到最好的平行化模型是Fig4第二部分用pthread，第三部分同時用 CUDA 和 pthread，在4條執行緒時，比起原始未加速程式快3.89倍。經過歸納與分析，我們整理出以下幾點結論。

A. pthread適合用在行為複雜的情況

OpenMP為高階平行化函式庫，許多功能都是被包裝好的，在複雜的程式中，很難針對個案去做特殊的設定，如 SSE 的使用，必須指定記憶體位置，pthread可以簡單算出陣列位置，但 OpenMP 卻需要較多的程式碼才能實現。

B. 當資料量小，固定配置比動態配置快

配置資料給執行緒的方法中，在總體資料小的時候，固定配置比動態配置來得快，因為動態配置需在執行時使用類似互斥鎖的機制分配資料，這會拖慢總體執行效率；反之，隨著資料量增長，動態配置的效益會超過配置時所耗的時間。

C. 在缺乏GPU資源時，可透過Intel SSE加速

近期GPU的使用越來越廣泛，然而現今大部份的企業仍受困於工業慣性缺乏GPU資源，若要進一步加速程式，就必須善用既有CPU的資源，Intel SSE或Arm NEON皆能提供在硬體上平行化的運算，在IV節的實驗中，SSE 透過較短的資料搬移時間與CUDA較快的運算速度相抵消後，SSE只比 CUDA 慢一些，這代表善用 CPU 資源也能大量增加程式的效能。

D. 善用多種平行化工具

pthread、OpenMP、CUDA、SSE都有各自的優點與缺點，在本次實驗中，最佳的模型是在特定的處理流程導入最合適的函式庫，意即唯有結合各函式庫的優點，方能讓程式有最大的效率。

VII. References

[1] Marco Dorigo Univ. Libre de Bruxelle, Brussels, Mauro Birattari Univ. Libre de Bruxelle, Brussels, Thomas Stutzle Univ. Libre de Bruxelle, Brussels, Ant colony optimization, Nov. 2006

[2] Sezgin Kilic Industrial Engineering Department, Turkish Air Force Academy, Istanbul, Turkey, Omer Ozkan Industrial Engineering Department, Turkish Air Force Academy, Istanbul, Turkey, Ant colony optimization approach for satellite broadcast scheduling problem, 19-22 June 2017

[3] M. Dorigo IRIDIA, Vrije Univ., Brussels, Belgium, L.M. Gambardella, Ant colony system: a cooperative learning approach to the traveling salesman problem, Apr. 1997

[4] D. Merkle Inst. of Appl. Informatics & Formal Description Methods, Karlsruhe Univ., Germany, M. Middendorf, H. Schmeck, Ant colony optimization for resource-constrained project scheduling, 07 Nov. 2002

[5] M. Dorigo Univ. Libre de Bruxelles, Belgium, V. Maniezzo, A. Colorni, Ant system: optimization by a colony of cooperating agents, Feb. 1996

[6] Dorigo, M., V. Maniezzo, and A. Colorni, “Positive feedback as a searchstrategy,” Technical Report no.1-016 Revised, Dip. Elettronica, Politecnico di Milano,1991a

[7] Wang, Xiaoting, et al. "Improving personalized trip recommendation by avoiding crowds." Proceedings of the 25th ACM International on Conference on Information and Knowledge Management. ACM, 2016.