系統軟體與平台主題

多核系統效能分析與優化

多核系統效能分析與偵錯工具模組

實驗模組名稱:多核心應用程式開發

**開發教師: 洪士灝**([hungsh@csie.ntu.edu.tw](mailto:hungsh@csie.ntu.edu.tw))

**開發學生: 葉志威**([freedomyeh@hotmail.com](mailto:freedomyeh@hotmail.com))

**學校系所: 台灣大學資訊工程學系**

**聯絡電話: 02-3366-4888 x 504**

**聯絡地址: 臺北市羅斯福路四段一號**

**台灣大學德田館504室**

**繳交日期: 2014年1月22日**

**實驗平台: Windows or Linux**

**實驗主軸: 效能檢測軟體及應用**

**實驗內容關鍵字:** Intel Vtune, pThread, performance analysis

**建檔編號: (??)**

**實驗目的**

Given a program, students are expected to use Pthread for parallelizing a program and profiling it by Intel Vtune Performance Analyzer.

**實驗器材**

* PC x 1
  + Requirement: Any PC with dual core at least.
  + Purpose: To run a multi-threading program.

**實驗所需軟體**

**PC**

* + Windows or Linux OS
  + Vtune–Parallel Studio XE 2013
  + Extern Pthreads Library (For Windows Only)
  + GCC (MinGW, GCC, Dev-Cpp, Microsoft Visual Studio)

Table of Contents

[1. Theorem Introduction 3](#_Toc378113531)

[i. Why is Multi-tasking? 3](#_Toc378113532)

[ii. What is parallel look like? 3](#_Toc378113533)

[iii. Do We Really Benefit from Multitasking? 5](#_Toc378113534)

[iv. When Process meets Thread? 6](#_Toc378113535)

[v. What Does a Thread Look Like? (On the other hand…) 7](#_Toc378113536)

[2. How to parallelize? 10](#_Toc378113537)

[i. Task Parallelism 10](#_Toc378113538)

[ii. Data Parallelism 10](#_Toc378113539)

[iii. Granularity 11](#_Toc378113540)

[iv. Pipeline Parallelism 11](#_Toc378113541)

[3. How to Use Pthread? 12](#_Toc378113542)

[i. Naming Convention 13](#_Toc378113543)

[ii. Example : Create a Thread 14](#_Toc378113544)

[iii. Example : Passing Data 15](#_Toc378113545)

[4. Parallelism Essence 18](#_Toc378113546)

[i. Master-Slave Model 18](#_Toc378113547)

[ii. Degree of Parallelism 19](#_Toc378113548)

[iii. Data Division 19](#_Toc378113549)

[iv. The Devil is in The Details 22](#_Toc378113550)

[5. Installing required software 24](#_Toc378113551)

[6. An example of profiling a parallel program 24](#_Toc378113552)

[7. Homework 28](#_Toc378113553)

[Homework Report Example 29](#_Toc378113554)

[**APPENDIX** 30](#_Toc378113555)

[**i.** **GCC with Pthread in WINDOWS:** 30](#_Toc378113556)

[**ii.** **GCC with Pthread in Linux:** 30](#_Toc378113557)

[**iii.** **Dev Cpp with Pthread:** 30](#_Toc378113558)

[**iv.** **Visual Studio with Pthread:** 30](#_Toc378113559)

[**v.** **Coarse-Grained (in data size) Example:** 31](#_Toc378113560)

[**vi.** **Fine-Grained (in data size) Example:** 31](#_Toc378113561)

[**vii.** **Registration Steps:** 31](#_Toc378113562)

1. Theorem Introduction
   * 1. Why is Multi-tasking?

在現在的年代，多工是必備的，而一個程式的平行化也更是重要的，所謂團結就是力量！以前一開始只要透過增強CPU的效能就可以提高程式的速度，然而在2004年開始我們已經不得不走向多核心處理器來提高CPU的效能，由於Power Wall、Memory Wall、等等的牆壁擋住我們的去路，只好拐個彎換個角度來思考，加上Moore’s Law說明電晶體數量每過18個月就會增加一倍，所以多核心是勢在必行。

為了提高程式效能，有很多不同的方式，CPU、GPU、FPGA個別施展個別的優點來做專門的平行化工作，像是CPU可以處理複雜的條件判斷和計算和IO，所以很多IO的控制都會獨立出來減少blocking的時間，計算繁重資料量小的程式也比較適合在CPU上面執行；另外像是把原本專門拿用作3D圖形渲染的GPU拿來使用加速平行計算的功力，例如：N-Body、KNN、等等，這類資料相依性很低的程式都是在GPU上面表現極佳的程式；而FPGA則是透過他的Pipeline和組合邏輯的特性，使得他可以在某些特定情況下近乎一個clock就輸出一筆輸出資料，clock rate只需要150M Hz就可以跑得比CPU還要快，因為只要放得進去FPGA，他也可以在一個clock處理非常多運算，這使得他非常適合串流運算，像是網路、亂數產生器、影像處理等等。

這次的主題是著重在CPU的平行化部分，接下來會帶你了解CPU平行化的觀念和如何平行化一個程式。

* + 1. What is parallel look like?

所謂平行化，就是使用multi-tasks或multi-threads來完成一個工作，透過把工作分散開來，我們可以利用到現在CPU多核心的效能，如果沒有平行化其實程式也可以跑，只是一顆CPU現在都起碼是雙核心，如果只用到其中一顆，另外一顆還是要消耗掉基本的電量，這個耗電量是非常可觀的！具體的數字不一定，但是大約最少會有10%的滿載耗電量，也就是說，一顆100W的CPU光是待機就會消耗10W的電量；對於**雙核心**手機的話，假設1個core待機耗電量0.05W。假設跑完一個**沒有**平行化的程式需要60秒，這隻程式對整個CPU會增加0.1W的功耗，則功耗為；假設跑平行化過的程式需要40秒，這隻程式對整個CPU會增加0.15W的功耗，則功耗為，節能約1.2倍！！假設這隻程式是作業系統裡面的一個時常被執行到的程式，他平行化之後會使得系統效能提升，此外也會降低耗電量，使得使用時間增加1.2倍！

平行化就像是把工作負擔分給多個核心一起做，增加計算效率，減少等待時間，理論上的說明非常複雜，想要了解到很透徹的需要讀過計算機結構才行，這邊我們就透過觀念式的模型來告訴你什麼是平行化！

所謂的平行化就像是去大賣場購物，當我們推著手推車就像是載著一籃的程式碼，然後每台車都是一個task，要去結帳時我們會做什麼？

Load Balance，先看看哪個櫃檯人最少才會過去排隊，這個就是所謂的附載平衡(Load Balance)，此外我們還會看哪個櫃台結帳比較快，因為有些人載了滿滿的程式碼，有些櫃台效率比較快有些比較慢，為了自己能夠快速結帳出去，我們除了觀察哪邊人最少之外還會觀察結帳速度，這也是附載平衡(Load Balance)的一部分。

Context Switch，我們結帳的時候會發現有時候自己一開始觀察錯誤了！想不到某一排後來結帳結的特別快，偏偏就在隔壁(In the same cluster)，這時候就會想要換道排隊，這就是所謂的Context Switch，一旦發現另外一邊空出來，馬上就可以結帳了我們就會換車道，儘管這會消耗一些**時間**和**力氣**來把自己的**狀態**轉移到另外一個車道，我們還是會換過去，因為這樣比較快！

Simultaneous Multithreading，最後會發現，這樣結帳還是不夠快，有時候要換車道偏偏後面又有人，不方便，所以我們要怎麼辦呢？這時候就需要更多的小奴隸，來做更動態的轉換，剛剛我們發現車子太笨重了，所以現在就透過小奴隸來搬運程式碼，把程式碼裝成一箱一箱的可以分開的(Private Data)，如果分不開的舊保留在車中(Shared Data)，每箱東西都不會太重(Light Weight)，透過人工搬運的方式來切換車道，這樣子可以很快速的直接跨過去到任一個附近的櫃台，非常動態，隨時這邊結一個物品之後就換下個櫃台結另外一個物品都可以，直到把箱子裡面的程式碼都結完，這樣動態切換的方式稱作SMT(Simultaneous Multithreading)，當然這樣切換還是有些overhead，像是結完一個物品就必須算一次發票，因為收銀機的資料沒辦法共用(registers)，但是由於現在的CPU架構非常好，所以這方面的overhead已經幾乎沒有了。

Congestion，由於每次結帳完自己手上的箱子就必須重新去拿下一段資料，所以手推車這邊會造成擁塞(Congestion)，Threads太多會造成手推車擁塞(Congestion)之外也會造成櫃台擁塞(Congestion)，所以太多也是不好的！

Dead Lock，如果是被命令是要結帳某類型物品，有些物品是在兩種類別都算的，這時候就會可能造成Dead Lock，因為要等到自己東西都拿完才能去結帳，如果這時候別人持有自己要的東西，自己持有別人要的東西，彼此又不肯放手就會造成Dead Lock。這部分雖然比喻有點牽強，不過概念上就是這樣。

於是乎透過以上技巧我們就能夠快速地把自己買的東西都結帳完，這就是multithreading的觀念還有運作方式，下次買東西的時候可以試試看是不是真的比較快喔！

* + 1. Do We Really Benefit from Multitasking?

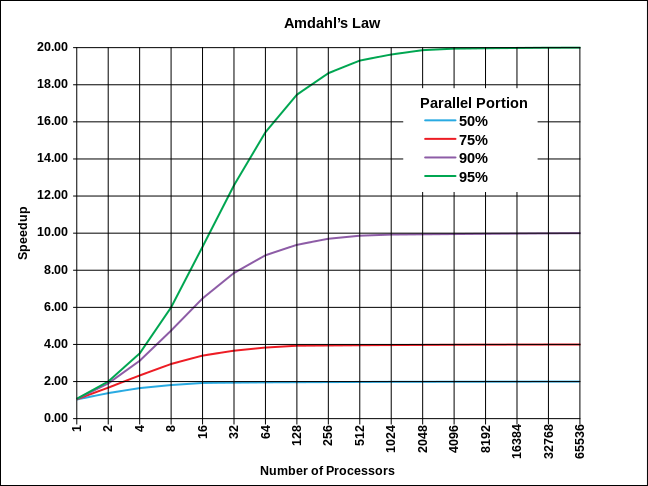
在多工的時候有個很重要的問題，就是是不是真的有增加效能？分別可以從軟體和硬體角度來看這件事情。

首先是軟體方面，不是每個程式都適合多工，如果資料相依性非常高或著其他情況都會影響到多工效能，依據Amdahl's Law：

S(n) = \frac{ T\left(1\right)}{T\left(n\right)} = \frac{T\left(1\right)}{T\left(1\right)\left(B + \frac{1}{n}\left(1 - B\right)\right) } =  \frac{1}{B + \frac{1}{n}\left(1-B\right)}

Source:<http://en.wikipedia.org/wiki/Amdahl's_law>

我們可以畫出



Source:<http://en.wikipedia.org/wiki/Amdahl's_law>

所以程式平行化的效能其實是有極限的，最開始可以很快獲得提升，但是接下來就會慢慢變慢，最後幾乎持平，在這段曲線中，功率消耗也是跟著成長的，再效能提升慢慢變慢的階段，什麼地方是功率消耗和效能提升比(Performance per walts)最低的時候，通常是最好的點。

從硬體角度來看就又是另外一回事了，CPU硬體架構會影響平行化的效能這在上一個主題就學過了，這邊要來探討另外一個有趣的問題，在有些手機還是單核心的情況下我們應不應該要平行化呢？

這個答案是肯定的，因為就算是在單核心，程式還是會時常發生資料相依、等待IO回應、等等事件，所以透過平行化能夠提高CPU的使用效率，減少等待延遲，還是能夠獲得效能提升的！

* + 1. When Process meets Thread?

除了multithreading之外還有所謂的multitasking，這兩種都可以達到多工，一種是多個小奴隸來結帳，另外一種是多台推車來結帳，這兩種方式有很大的差別，當然個別也有個別的好處。

Big V.S. Small，multitasking需要很多台推車，推車很大，想要切換櫃台(Context Switch)時不方便，反之multithreading的小奴隸們可以快速翻越櫃台。不過太多的小奴隸反而會造成櫃台擁塞(Congestion)；反之由於推車已經把所有被分配要結帳的物品都放在推車裡了不會像一直翻越回去拿資料和程式，所以比較能夠專一的結完帳，也比較不會造成擁塞(Congestion)使得大家結帳都慢下來。

Shared Memory，小奴隸(multithreading)的作法是找到很多小奴隸來結帳，所以只需要複製狀態就好，有需要拿程式碼或資料再回去推車拿，所以耗費的空間比較小，並且資料是共用的，所以拿東西的時候一旦有變更也會馬上知道；相對的，推車(multitasking)的作法是先把所有程式和狀態都複製一份，找一台推車來裝，然後去結帳，所以要生成task比較慢，資料都是private的，一旦更新資料就必須要透過Message Passing**通知**另外一台推車資料要變動。

* + 1. What Does a Thread Look Like? (On the other hand…)

Thread就像是task裡面的task，他是一段非常輕量級的程式碼，根據維基百科定義A thread is a smallest sequence of instructions that can be managed by operating systems.右圖就是常看到的multithreading的架構圖。上面的框框是CPU，也就是剛剛我們比喻的很多個結帳櫃檯，下面的三個框框每個都代表一個task則是手推車，裡面的線則代表thread就是小奴隸們。

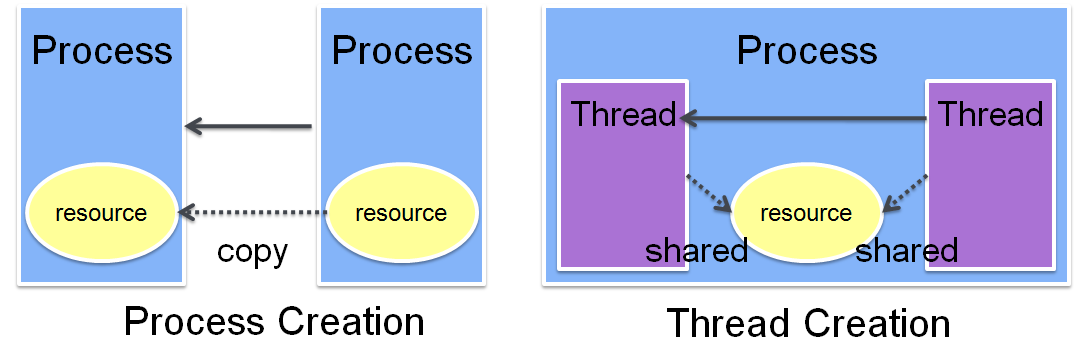
CPU

Task

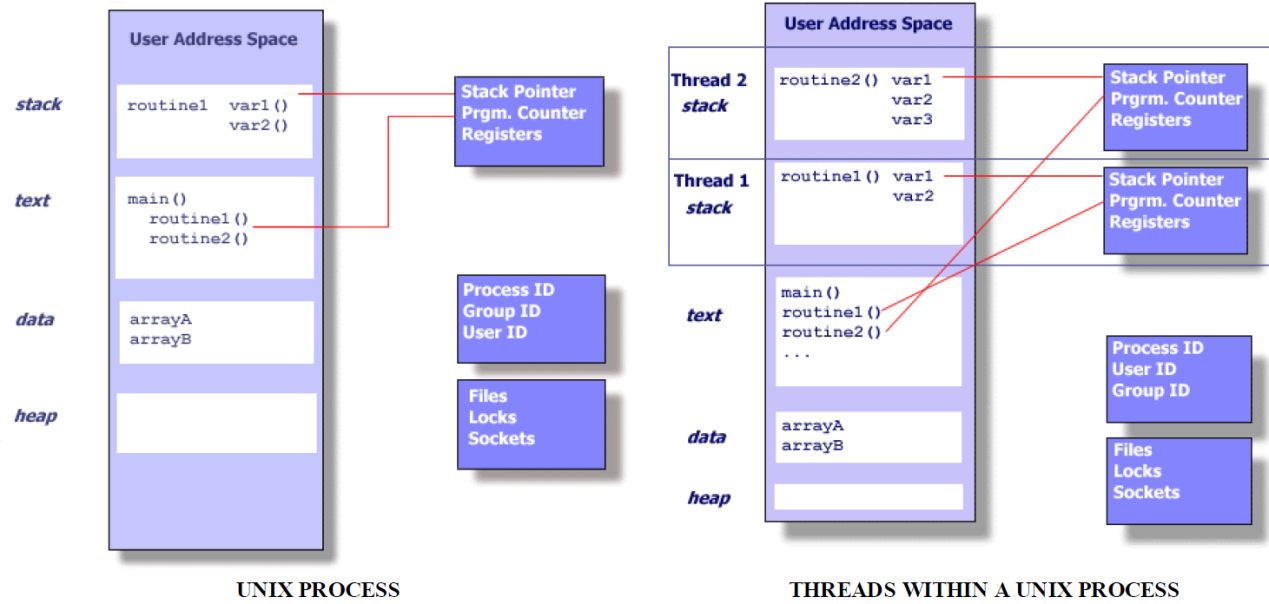
Task

Task

Thread是由軟體和硬體配合所提供的一種架構，並且由各個core獨立執行各個thread的，通常都是定義為**function** or **procedure**來呼叫和執行，從軟體來看是由Operating Systems、Thread Library、Virtual Machine來提供API讓程式呼叫，硬體來看的話是由SMT/hyper threading或GPU來實行實際上的切換，這兩著缺一不可，少了硬體切換速度會非常的慢，因為每次切換都必須用程式碼來自己複製自己的資料存起來，然後複製別人的資料進來；少了軟體則是只能夠在一個core上面執行，只能夠用到一個core的multithreading的資源而已，變成很浪費，所以threads是靠軟體還有硬體兩個配合才能夠正常運作的。



這張圖表示了process creation(task creation)和thread creation的差別，process creation 他需要把自己的resources複製一份，然而thread creation就不用，他們是用共資源的，如下圖詳細的圖說明。



一個process包含至少stack、text、data、heap，如果是process creation，他必須要把這些東西除了text之外全部都複製出另外一份！然而thread creation他只需要複製stack pointer、program counter、registers這樣子大大減少複製所需要的時間並且資料都是共用的。

所以到底差多少呢？根據實驗，平均一次fork需要約0.25ms；反之，平均一次pthread需要0.036ms。

stack、text、data、heap這些是做什麼的呢？一個程式會被分成好幾個區段(section)，以便管理，而其中最常用的就是這幾個區段，也是基本會有的區段。stack是function call的時候或著是有private variable和return value時會把資料push進去，直到要用的時候才會pop出來；text是存放程式碼，pc會指向這裡；data是存放global data或著已經定義好的string或陣列；heap是動態記憶體規劃malloc()時會從這邊取得記憶體空間來使用，所以個別的section有個別的作用，不然記憶體和程式碼都混在一起會不好管理，當然也有混在一起的方式，不過最常用的還是這邊看到的這種方式。

所以我們整理一下Process還有Thread的觀念

1. 一個Process可以有多個Thread。
2. 同一個Process內的Thread使用相同的Memory Space，但這些Thread各自擁有其Stack，換句話說local variable是各自獨立的。
3. 作業系統會根據Thread的優先權以及已經用掉的CPU時間，在不同的Thread作切換，以讓各個Thread都有機會執行。

最後我們來個小測驗來看看你是否真的了解觀念了：

1. How does the OS create a new process?
2. How does Pthread create a new thread?
3. What if you have many threads and only one CPU?
4. What if you have M threads and N CPUs?
5. Are all threads treated equally by the OS/Pthreads?
6. What if two threads want to access the same global variables at the same time?

小測驗的答案在這邊喔：

1. Create the needed sections and load the program/data
2. Just need to create new private sections (e.g. PC, reg, stack)
3. **Context switch periodically**: Save the state of one thread; suspend; load the state of another thread; run.
4. Complicated – no perfect solutions, but the OS tries to do its best by migrating threads and balancing the number of threads; **Sometimes users want to bind the threads to CPU**
5. The **priority** of a thread can be adjusted by the application
6. **Race condition**!
7. How to parallelize?

平行化有很多種，大致分為三種類型，分別為Task Parallelism、Data Parallelism、以及Pipeline，前面兩種是軟體層的，最後一個是硬體層面的。

* + 1. Task Parallelism

Task Parallelism就是依照工作、任務來平行化，所以每個cores可能會執行不同的程式碼，也是日常生活常常或碰到的情況，像是家裡要洗衣服、煮菜、打掃、倒垃圾，這些事情可以分開做，不用同一個人全部做完，家裡每一個人做一個工作，這樣子的話就可以使得這些工作很快速地就可以做完了。所以Task parallelism就是不同cores可以執行不同的工作，另外這些工作可能是有相關性的，像是煮菜之後可能會變髒，打掃的人就要來清理，但是這些都是可以交給不同人做。電腦程式裡面最常見的應該就是網路連線，每次有一個新的連線都會開啟一個新的thread去回應他，所以每一個人來連線的時候都當作是一個任務，分派一個thread去處理，但是連線可能會和其他連線有所關連，所以並不一定是無關的，像是通訊就需要和其他thread做溝通，這就是所謂的task parallelism。底下是範例pseudo code來說明依照不同cores來執行不同的task的程式大概會長怎樣，是怎樣的特性。

ifcores=1 then

do task A

else if cores=2 then

do task B

end if

* + 1. Data Parallelism

Data Parallelism，和Task Parallelism相反，Data Parallelism是依照資料來分派任務、工作，把資料切割成多份，讓不同的cores去處理那些資料，所以每個cores可能會執行相同的程式碼但是會處理不同的資料。生活中這個例子也很常見，像是老師如果要改考卷，課堂小考考完都會讓學生交換考卷互相改，這樣的話一個人就只需要改一份考卷，很快就可以改完了！這樣子依照資料來做切割，分給不同cores處理不同資料就是Data Parallelism。至於資料要怎麼切割，切的多細就是另外一個議題了。底下是範例pseudo code來說明依照不同cores來執行不同的data的程式大概會長怎樣，是怎樣的特性，最底下的foo(data [i])就是指要對data做的處理，執行到for-loop的時候由於if條件判斷不一樣，所以會執行不一樣部分的資料。在這個範例裡面切的就是Coarse-Grained的方式，切的顆粒度很大。

ifcores= 1

lower\_bound := 1

upper\_bound:= round(data\_length/2)

else if cores= 2

lower\_bound:= round(data\_length /2) + 1

upper\_bound:= d.length

for i from lower\_bound to upper\_bound step by 1

foo(data [i])

* + 1. Granularity

Granularity，顆粒度，也就是要把資料分割多細，一個常見的例子是影像處理，影像處理常常可以把像素切成N部分分別給N個cores去執行，切的時候有不同的切法，例如兩個cores的CPU，我們可以把一張圖片分成上下兩部分，第1部份給第1個core，第2個部份給第2個core去處理，這是屬於Coarse-Grained，也是上面範例所用的方式。第二種切割方式是依照第幾行來做切割，奇數行交給core 1去處理，反之偶數行交給core 2去處理，這樣資料切割比較細一點，也會比較集中。第三種方式是Fine-Grained，就是把資料切得很細的意思，實際上就是第奇數個像素交給core 1，反之第偶數個像素core 2去處理，這樣子分割是最細的方式，他的Data Locality也會最高，在執行的時候通常會比較能夠利用到cache的資源。Coarse-Grained的方式可以把資料獨立切割分很開，通常也比較好切，但是cache可能會比較浪費資源，有時cache miss會比較高；反之Fine-Grained的方式可以把資料切割分得很緊密相鄰，但是卻比較可能造成hot-spot，導致效率變差。

* + 1. Pipeline Parallelism

Pipeline，這種平行化的方式比較特別，源自於CPU，透過把複雜邏輯拆成好幾階段，使得每數個clock就可以完成一個階段，每個階段都會是獨立的，這種方式有點類似task parallelism，但是最大的不同之處就在於pipeline是針對一個工作，而task parallelism是分配不同工作給不同CPU，透過把一件複雜工作切成好幾個階段，每個階段完成每個階段該做的事情，像是我們如果要洗衣服，一整個流程會是先洗衣服(30分鐘)再脫水(5分鐘)再烘乾(15分鐘)，這三步驟如果都在一台洗衣機上面執行會需要一小時，所以如果有三台洗衣機，能夠洗的數量就是3批/小時，這樣子是Data Parallel的方式，如果現在分開這三步驟到三台機器上面，一台機器只負責一件事情，做完就換到下一台機器，雖然一次只能夠洗一批，使用piepeline可以洗的數量是1批/半小時。雖然看起來piepeline他洗得會比較慢，但是他卻有一個很大的優點是，他每半個小時就可以洗完一批，不像3台洗衣機要一個小時才能洗完一批。

我們舉個例子，如果現在有個非重複亂數產生器的演算法，他一次能產生出一個亂數，假設他的程式步驟分成三個獨立的步驟，分別是fun1, fun2, fun3這三個function來處理這三個步驟，若想要即時產生100萬個亂數，這時候就會使用pipeline的方式來平行化，這樣子平行化可以使得這個演算法加速最快3倍(要看執行最久的function要多久)，就可以盡量達到即時的效果。另外一個做法是可以透過一次執行三個非重複亂數產生器的程式，這樣子可以使得他一次吐出3筆亂數，但是這樣子卻會造成１、亂數可能重複，因為程式之間不知道對方產生了數字，２、除非事先執行，並且存起來，否則無法像pipeline那樣達到即時的回應。

其實在一般系統中，其實也很常利用pipeline parallelism的架構，因為它可以很有效的隔絕input/output還有message passing的問題，這也是現在在做平行化時很常用到的技巧，例如IO、中斷、FBP。然而我們會發現其實pipeline和task parallel真的很像，他們都是獨立出任務交給不同的core去跑，最大的不一樣就在於pipeline的**資料是會一直不斷地進來或出去**，因為是**把一個任務切割成好幾塊給不同core去執行**，然而task parallelism在執行階段只會執行有限次數的輸入輸出動作，因為是**把不同任務丟給不同core去執行**。最後在強調一次，pipeline parallelism和task parallelism性質非常接近，所以寫程式的時候不用特別在意是屬於哪一種，可以都歸類於task parallelism去思考如何平行化就好。

ifcores= 1

fun1

else if cores= 2

fun2

else if cores= 3

fun3

1. How to Use Pthread?

Pthread並不是標準的library，它是由POSIX提出的架構，所以需要額外使用第三方的library，不過現在的GNU/Linux都有包含libpthread這個library，只要在編譯階段多下一個參數–lpthread就可以使用pthread的library，如果是WINDOWS底下使用DevCpp的話則需要額外下載並安裝pthread的library，需要的話可以自行去尋找網路上的教學。

1. Naming Convention

pThread裡面的function有它的命名規則，單字之間會加底線，以下表格整理出了常用的一些function的名稱和命名方式。

|  |  |  |
| --- | --- | --- |
| 名稱 | 用途 | 範例 |
| pthread\_XXX | 管理thread的functions | pthread\_create()  pthread\_exit() |
| pthread\_attr\_XXX | thread attribute的相關設定和產生，是用來設定thread的屬性的 | pthread\_attr\_init()  pthread\_attr\_destroy() |
| pthread\_mutex\_XXX  pthread\_cond\_*XXX* | thread同步用的function | pthread\_mutex\_init()  pthread\_mutex\_lock()  pthread\_cond\_signal() , pthread\_cond\_wait() |
| pthread\_cleanup\_XXX | 用來指定一個特定function來清除在thread裡面曾經指定過的動態記憶體，避免發生memory leak。 | pthread\_cleanup\_push()  pthread\_cleanup\_pop() |

其中pthread\_create (tid, attr, func, args)，用途是創造新的thread，創造之後parent會繼續執行，child thread也會開始執行。這個function它包含了四個參數，tid是thread id，傳位址給他之後會透過這個變數回傳thread id給parent、attr是attribute，透過這欄來設定thread的一些屬性，例如是否joinable、是否可以由其他thread來中斷這個thread、等等、func是function，透過function pointer指定要用thread執行的function、最後是args，arguments，也就是要傳給thread的參數，建議使用global variable避免parent先離開導致private data消失。

1. Example : Create a Thread

#include <stdio.h>

#include <stdlib.h>

#include <pthread.h>

/\* Prints x's to stdout. Does not return. \*/

**void** \*print\_x(**void** \*unused)

{

**while** (1)

printf("x");

**return** NULL;

}

/\* The main program \*/

**int** main ()

{

pthread\_tthread\_id;

/\* Creating a new thread. The new thread

will run the print\_x function\*/

pthread\_create(&thread\_id, NULL, &print\_x, NULL);

/\* Print 'o' continuously to stdout \*/

**while** (1)

printf("o");

**return**0;

}

這個是一個簡單的範例示範pthread如何使用，結果應該會亂數顯示出o和x的組合。最上面的#include <pthread.h>宣告pthread裡面的function，print\_x這個function是負責print x，反之main負責print o，透過這樣的方式呈現這是平行執行的概念，main一開始需要定義thread\_id來儲存這個child thread的id，以後要對他做處理的時候才能處理，如果沒有紀錄的話就像是動態記憶體分配沒有assign變數來存位址會導致那塊記憶體變成孤兒一樣，這邊也需要有人記住thread的id，否則就沒有人知道他在哪了，thread\_create則是用來創造一個新的child thread，並且執行print\_x這個function。

Thread要離開的時候有幾個方式，第一個方式是透過return來離開，第二種方式是在child thread裡面呼叫pthread\_exit()來離開，pthread\_exit()可以是在print\_x()裡面被呼叫，或著是print\_x呼叫其他funciton，那個function呼叫pthread\_exit也會導致thread被終結掉，這是第一個和return不一樣的地方，另外一個不一樣的地方是，如果有用pthread\_cleanup\_push()指定任何一個function的話，return並不會執行任何cleanup function，如果使用pthread\_exit()，他會在離開前呼叫我們指定的cleanup function來釋放所用過的記憶體，避免memory leak。

1. Example : Passing Data

#include <stdio.h>

#include <stdlib.h>

#include <pthread.h>

**struct**print\_args {

**char** character;

**int** counter;

};

/\* Prints characters to stdout.

The parameter is used for passing data. \*/

**void** \*print\_c(**void** \*args)

{

/\* Cast the pointer to the right type we use \*/

structprint\_args \*ptr = (structprint\_args \*)args;

**int** i;

**for** (i = 0; i <ptr->counter; i++) {

printf("%c", ptr->character);

}

**return** NULL;

}

/\* The main program \*/

**int** main ()

{

pthread\_t thread1\_id;

pthread\_t thread2\_id;

**struct**print\_args thread1\_args;

**struct**print\_args thread2\_args;

/\* Create a new thread to print 20,000 '0's. \*/

thread1\_args.character = '0';

thread1\_args.counter = 20000;

pthread\_create(&thread1\_id, NULL, &print\_c, &thread1\_args);

/\* Create a new thread to print 10,000 '1's. \*/

thread2\_args.character = '1';

thread2\_args.counter = 10000;

pthread\_create(&thread2\_id, NULL, &print\_c, &thread2\_args);

**return** 0;

}

在這個範例中創造了兩個threads，分別print 0和1，透過這個範例示範如何透過struct傳遞多個參數給thread，傳遞參數時強烈建議要透過struct包起來，否則無法傳遞多筆資料，如果要用到陣列也是一樣，需要有變數代表陣列大小，所以需要struct包起來才能夠同時傳遞陣列和陣列大小。

main最開始定義了兩個thread\_id分別記錄兩個thread，並且創造出兩個local variable來儲存參數傳給兩個thread執行，分別透過pthread\_create創造child threads來執行print\_c，在print\_c裡面最一開始是透過型態轉換把泛指標(void \*)轉換成我們所傳遞的參數型態以取得資訊，由於ptr和i都是local variable所以他們在執行的時候兩個thread所看到的實際上是不同位址，不會衝突，接下來就是透過for-loop print指定數量的字元。

這個程式看起來好像完成了，創造threads、指定參數、各個thread執行時是獨立的不會影響彼此，thread function都保有自己的local private data不會導致race condition，但是這個程式實際上有一個很嚴重的bug！這個程式編譯過後執行會發現實際上什麼都沒顯示，甚至可能會發生segmentation fault！原因是因為在main最後直接就結束掉parent process，所以導致其他threads也被強迫關閉，並且還有一個很嚴重的問題是，我們是用local variable來指定threads的參數，如果parent先return了的話，這些local variable也會跟著消失，所以可能會導致segmentation fault！

想要修正bug的話可以透過pthread\_join(tid, retval)來等待這個thread結束，tid是thread id，指定是哪一個thread，retval是return value，透過這個指標來傳遞return value給parent。這樣會使得main在最後會依序檢查這兩個thread是否結束了，如果結束的話才繼續往下走，如果沒有結束會一直等到結束為止，這樣就可以避免上述的process exit和segmentation fault的情況發生了！

**int** main ()

{

…

/\* Make sure two threads are finished \*/

pthread\_join(thread1\_id, NULL);

pthread\_join(thread2\_id, NULL);

**return** 0;

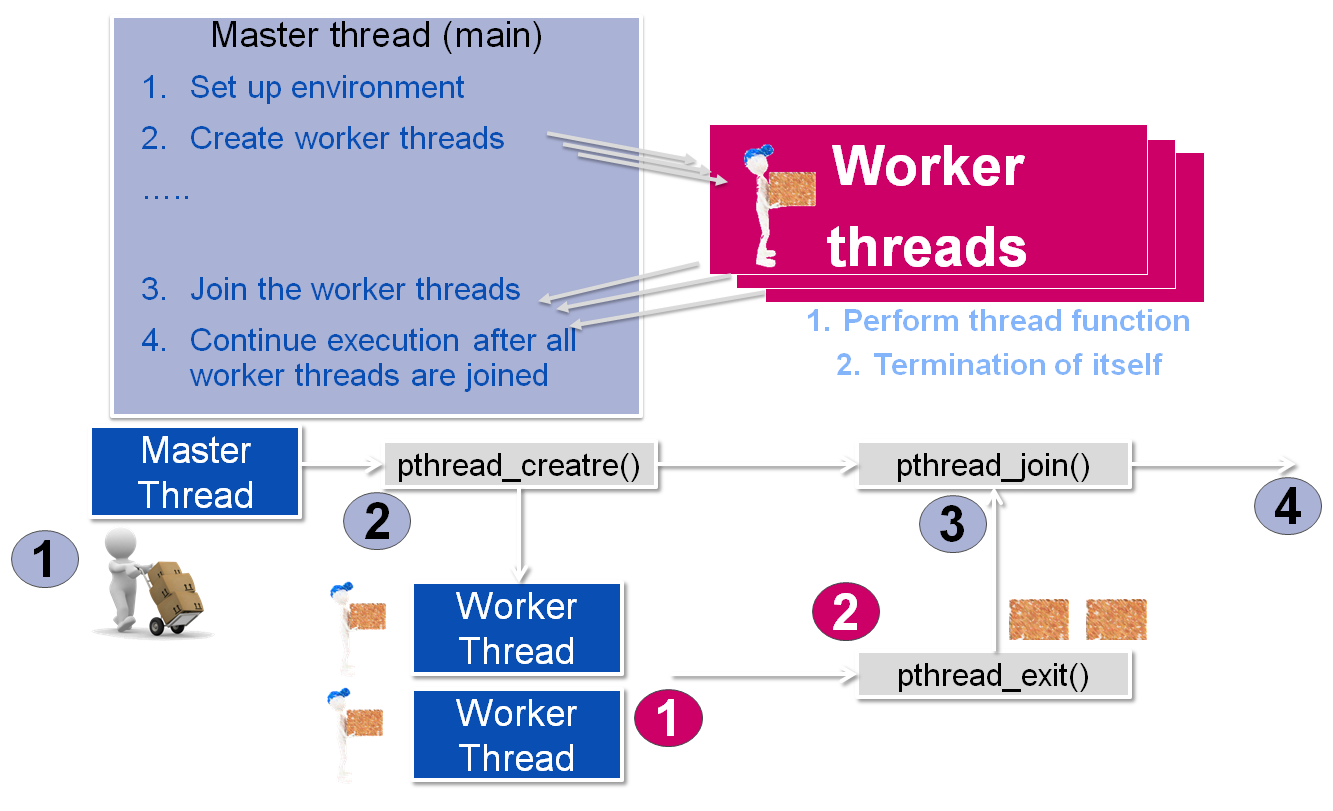
}

想要知道更多更詳細的pthread和linux在multithreading上是如何處理的可以參考這個檔案：

<http://www.advancedlinuxprogramming.com/alp-folder/advanced-linux-programming.pdf>

1. Parallelism Essence
   * 1. Master-Slave Model

寫一個平行化的程式有很多要注意的技巧，但是有一個很共同的地方就是最開始的程式一定都會是一個process然後產生其他threads/processes，而那些被產生的children也可能會再產生其他的，但是管理起來就會變得複雜，所以最常用的就是Master-Slave Model，也就是只有單一一個Master的Process然後產生其他Slave完成工作，前面提過的Task Parallelism和Data Parallelism皆是屬於這種Model，這種Model的好處就在於簡單直覺又很好用，能夠很清楚的知道程式執行模式以避免Race condition和其他問題，每次處理完之後就會join，Master就會在這時候把需要的資料從Slave那邊拿回來。像是計算積分的演算法平行化之後就是一個Master Slave的Model，Master把資料切成N等份，然後分給N個cores去執行，最後join的時候再把N段的區間和加總就是總積分的面積了。



* + 1. Degree of Parallelism

Degree of Parallelism(DOP)是一個公制單位，描述電腦或CPU在一個時間區間可以同時執行的processor數量，用來描述一台電腦執行平行程式的效能。如果把離散的時間段接起來，可以變成一個用來描述processor的數量對應時間軸的圖表，稱做Parallelism Profile。

DOP可以用來觀察自己的程式在平行化之後是否有利用到每個processor core的資源，以利改善自己程式的效能和找到瓶頸，這也是為什麼我們這麼需要工具來debug。

* + 1. Data Division

談到DOP，現在換來談談一個程式的效能是否平行化之後可以提升呢?又能提升多少?這邊舉一個平行度很高的影像灰階化的程式做為範例，三條線分別代表三種不同的演算法來平行化這個程式，現在我們來看一下底下這張圖，橫軸是cores垂直軸是相對於1個core的執行速度提升幾倍，你能知道這台電腦的DOP是多少嗎?也就是同時有多少個core在執行。

答案是4個cores，你答對了嗎?我們會發現到達4個cores的時候加速最多，到5的時候馬上就降下來，所以把一個程式拆越多，並不一定會得到效能提升，有時候差那一點的數字就會導致程式效能下降很多，當cores的數量變多的時候會發現Performance 2和Performance 3的效能又回復了一點，這也許是因為當數量又多起來的時候，系統排程可以把他排的效能比較好一點，但是Performance 3效能卻又沒有提升，這是為什麼呢?

原因是因為他的演算法的關係，其實Performance 1~3分別就是資料要切的多細，也就是從Coarse-Grained到Fine-Grained。概念上的圖如下圖，寬度5 pixel，高度8 pixel，每一個格子都是一個pixel，然後不同顏色代表不同的core負責執行的。

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 1 | 1 | 1 | 1 | 1 |
| 1 | 1 | 1 | 1 | 1 |
| 2 | 2 | 2 | 2 | 2 |
| 2 | 2 | 2 | 2 | 2 |
| 3 | 3 | 3 | 3 | 3 |
| 3 | 3 | 3 | 3 | 3 |
| 4 | 4 | 4 | 4 | 4 |
| 4 | 4 | 4 | 4 | 4 |

Performance 1

Coarse-Grained

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 1 | 1 | 1 | 1 | 1 |
| 2 | 2 | 2 | 2 | 2 |
| 3 | 3 | 3 | 3 | 3 |
| 4 | 4 | 4 | 4 | 4 |
| 1 | 1 | 1 | 1 | 1 |
| 2 | 2 | 2 | 2 | 2 |
| 3 | 3 | 3 | 3 | 3 |
| 4 | 4 | 4 | 4 | 4 |

Performance 2

In Between

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 1 | 2 | 3 | 4 | 1 |
| 1 | 2 | 3 | 4 | 1 |
| 1 | 2 | 3 | 4 | 1 |
| 1 | 2 | 3 | 4 | 1 |
| 1 | 2 | 3 | 4 | 1 |
| 1 | 2 | 3 | 4 | 1 |
| 1 | 2 | 3 | 4 | 1 |
| 1 | 2 | 3 | 4 | 1 |

Performance 3

Fine Grained

由於執行都是按照pixel的順序從左上到右下，或著以記憶體的配置位址來看的話就是從第一個pixel到最後一個pixel。講到這邊妳知道了這三個演算法的不同之處在哪裡了，聰明的你觀察出來這三個演算法造成剛剛的speedup結果不一樣的原因了嗎? 答案是Data Locality，由於CPU會把資料Cache住，加快程式執行速度，Cache的時候他一次會Cache一定數量的資料稱為Cache Line，由於Cache Line的關係加上其他程式和OS的影響，導致這三種演算法產生的結果會不一樣，Performance 3把資料切的最細小，導致他執行效率沒辦法像其他兩個一樣提升的原因就是因為切得太細！相鄰的資料都不是自己要處理的，導致Cache Line的使用效率非常的差，假設Cache Line大小是4 pixel(這邊為了說明方便，以pixel來做為單位，正常的單位是word)，core 2執行的時候會依序拿取array的第2, 7, 12, 17個pixel，cache在load第2個pixel的時候就會一次讀4個以加快讀取速度，然而core 2下一次要的資料pixel 7卻不在裡面，所以在load pixel 7的時候cache就miss，重新load pixel 7,8,9,10，我們會發現這樣子每次處理到下一個pixel時都會發生Cache Miss，導致Cache的使用率很差！其他兩個演算法的方式不會產生此問題的原因也是依此類推，因為他們一個core要處理的pixel通常都會是相鄰的pixel所以Cache的Hit rate比較高。

不過這邊只是一個特例，通常Fine-Grained不會把資料切到這麼細，一般的Fine-Grained會配合Cache的大小去調整，以利用Cache的效能，這次的例子只是一個極端的例子示範資料要如何做切割。

程式碼如下：

Algorithm 1 (Coarse-Grained):

start = param->num \* h / param->partitions;

**if** (param->num == param->partitions - 1)

end = h;

**else**

end = (param->num + 1) \* h / param->partitions;

**for** (y = start; y < end; y++){

**for** (x = 0; x < w; x++){

….

}

}

Algorithm 2 (In Between):

**for** (y = param->num; y < h; y += param->partitions){

**for** (x = 0; x < w; x++){

….

}

}

Algorithm 2 (Fine-Grained):

**for** (y = 0; y < h; y++){

**for** (x = param->num; x < w; x += param->partitions){

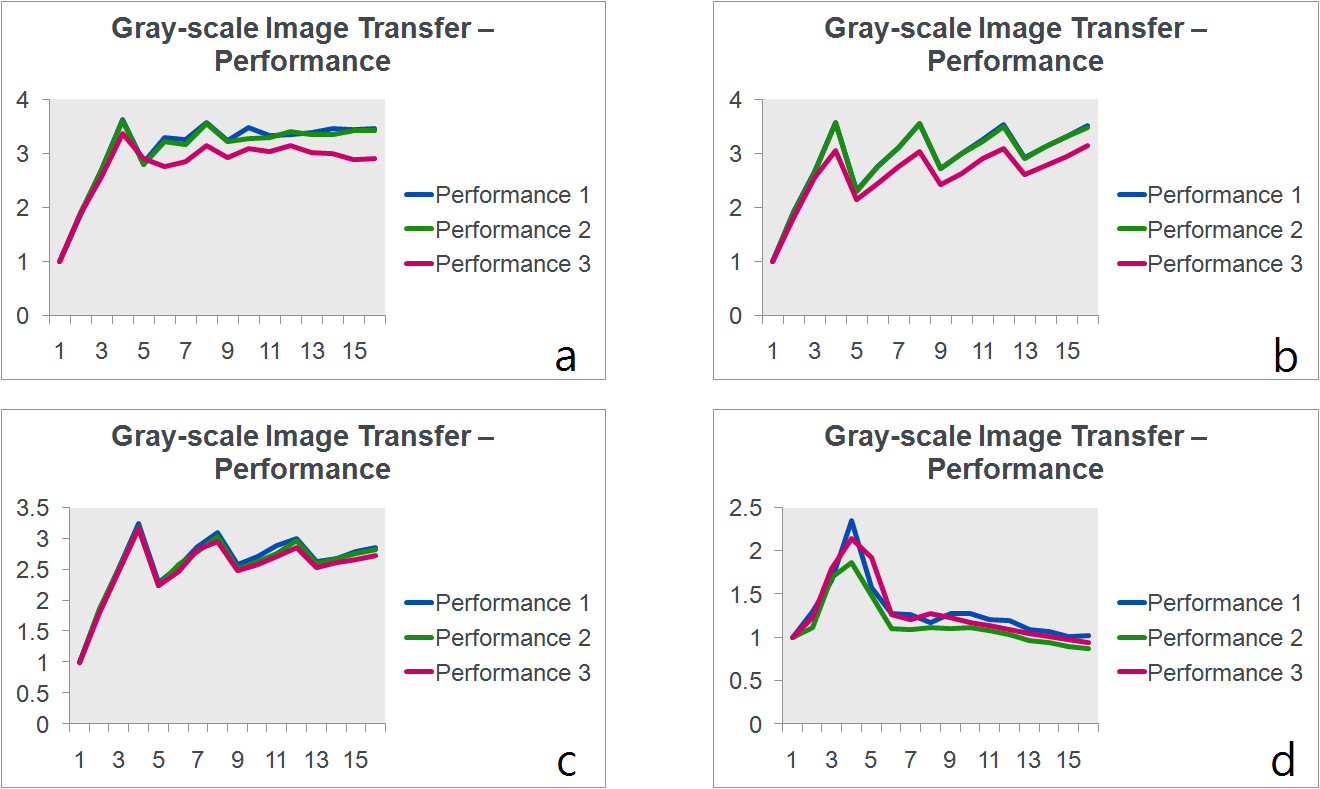
….

}

}

* + 1. The Devil is in The Details

接下來再看看這張比較圖，全部都是剛剛的灰階演算法，左上角的圖就是剛剛的圖。這些圖有什麼不一樣呢？是什麼造成這些結果不一樣？



我們會發現右上角圖表b和左下角圖表c的兩個圖表有個共同現象，每當core的數量到達4的倍數的時候效能提升的倍率就會達到區域最大值，造成這個現象的原因是什麼呢？就留給你自行理解吧。

其實造成這四張圖表不一樣的原因就是圖片大小！輸入的資料大小多或寡，也會影響平行化程式的效能！d圖充分表現出來當資料量太小的時候，如果還硬是把thread的數量拉到超過10個的話，得到的效能提升倍率會是微乎其微，但是很神奇的，剛剛說過資料切太細的Fine-Grained的演算法在這邊反而是不算太差的演算法，其他三張圖紅色的線都是最下面，只有這張圖紅色的線是在中間，甚至有一度超過藍色的線。

我們可以發現，當資料量太小的時候，其實不論什麼演算法都很難決定誰比較好，原因是因為thread creation的overhead已經大到佔總執行時間相當的比例，並且資料量越小執行時間的變動越大，這邊雖然從圖中看不出來，但是依照數據，這邊是取1000次的平均(mean)，但是這1000筆資料的標準差(standard deviation)隨著圖片大小越小而增加，這也是寫平行程式要考慮到的一環，要讓自己的程式效能好是需要依照各個參數下去調整的。另外一個原因是資料量小到可以整個塞進cache使得他的hit rate在不同演算法都差不多的關系所導致的，這點在c圖也可以看得出來，三條線一直都是在差不多的位置，並沒有明顯的差別。

說明到這邊，你看得出來圖片大小的順序依序是多少嗎？答案是a＞b>c>d。

小結，探討一個程式在電腦上能夠獲得多少效能常用的公制是DOP(Degree Of Parallelism)，並且透過Parallelism Profile圖可以觀察出在不同時間區段CPU的使用率是多少，想要得到Parallelism Profile圖就必須要透過工具才行，這也是接下來我們要進入到的單元，一個程式切多少個thread去執行會影響到程式的speedup，除此之外data要怎麼切或著task要怎麼分派也會大大的影響程式效能，Data Locality是解決效能瓶頸的關鍵，要提高cache的hit rate才能使程式的執行效率比較好，最後是輸入資料量的大小也會影響程式效能，當資料量太小的時候就不宜使用太多thread，因為thread creation overhead會佔了程式執行時間非常大的一部份。

1. Installing required software
   * 1. Download Vtune– Parallel Studio XE 2013 from Intel website (US version).

http://software.intel.com/en-us/intel-parallel-studio-xe

* + 1. An e-mail address is necessary for registration. You will have a 30-day license and a download URL.



* + 1. Install Vtune.
    2. Download Pthread. (For windows only)
    3. Install Pthread in a new directory C:\pThread. (Or any directory you want)

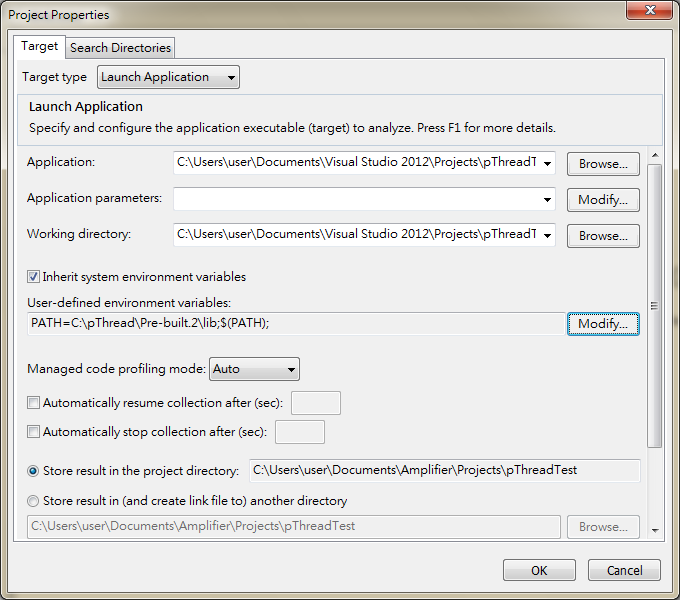
http://www.javaforge.com/doc/103669

(Only Windows needs to install the Pthread library.)

* + 1. Now, it’s time to get our serial code. [Optional]
    2. You can use free trial without serial code!!! The **serial code license file** is attached in the email. If you couldn’t find it, you can register by the steps in appendix G.

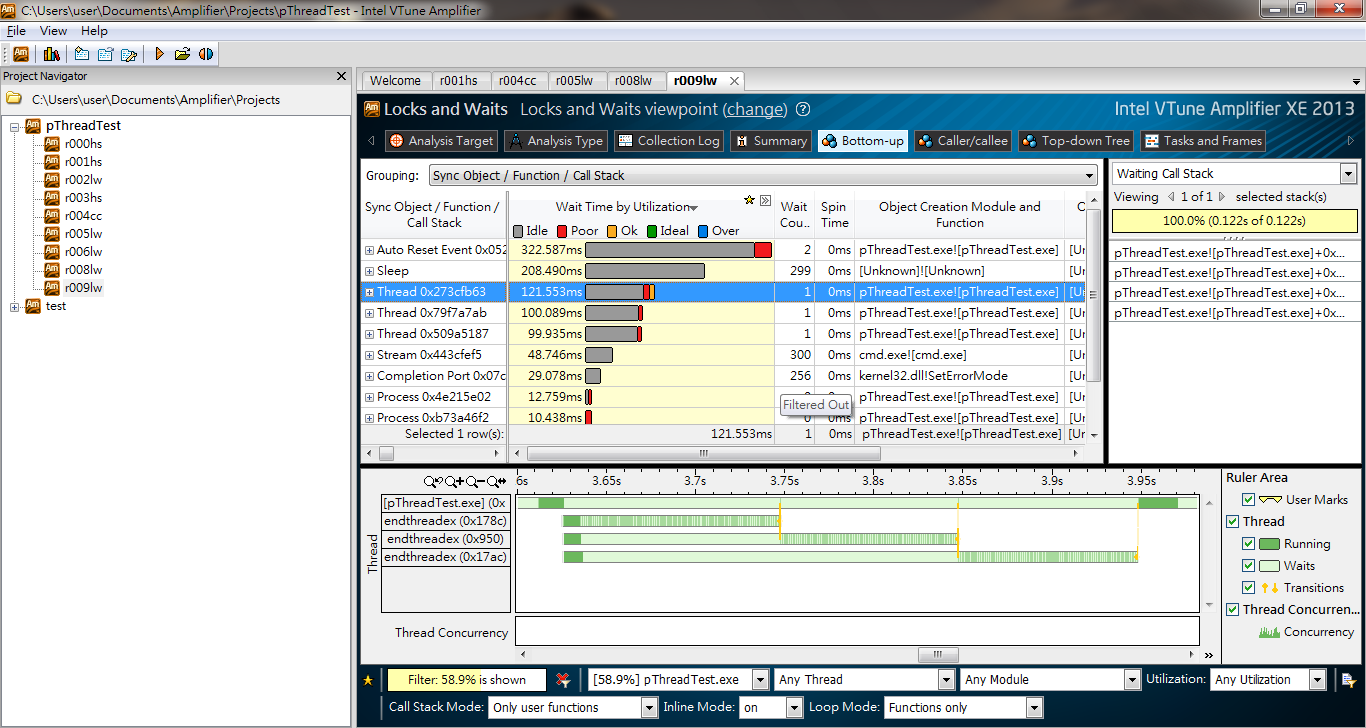
1. An example of profiling a parallel program
2. Open Intel VTune Amplifier XE 2013 and create a new project.
3. Open Project Properties. Browse the application we want to profile. For Windows user, modify the environment variables, add the path where you install Pthread library.(Pathdepends on where library is)

PATH=C:\pThread \Pre-built.2\lib;$(PATH);

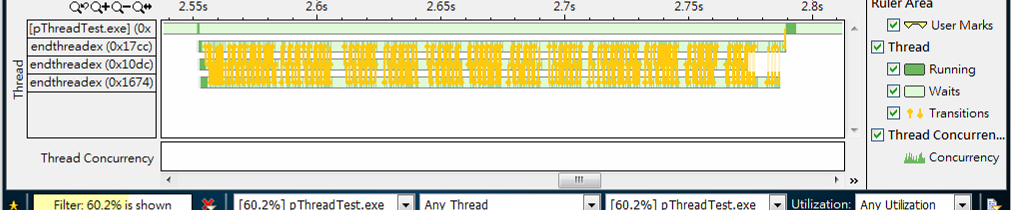


**Only Windows User Need to set PATH!!!!**

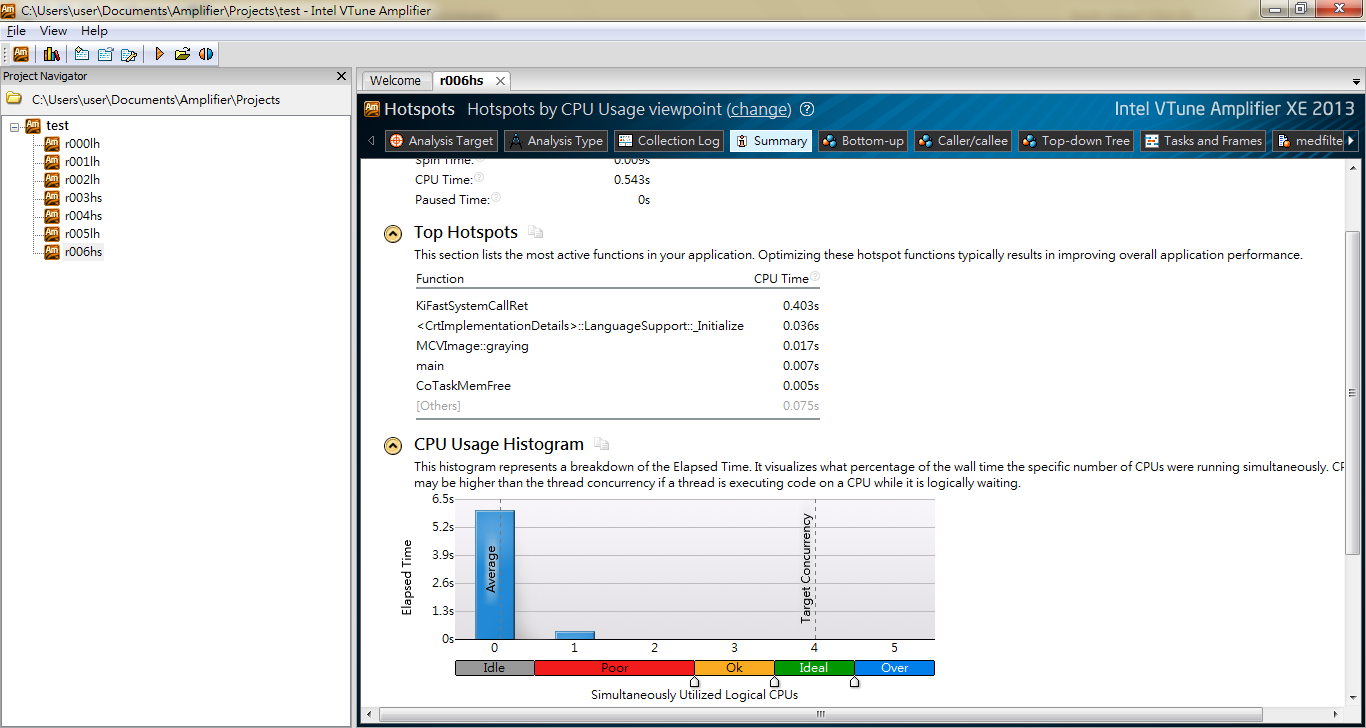
1. Create a New Analyze.
2. Choose Locks and Waits in analyze type and then press Start.
3. You can mark a timeline whenever you want as a debugging point.
4. After running, select Bottom-up tab. On the bottom are filters.(Solid line) Set process filter to the execution process name. There is a Thread Concurrency Area just above filter-chosen area. (Dotted line) Select a region where there are multi threads running, and then choose zoom-in-and-filter-in-by-selection.



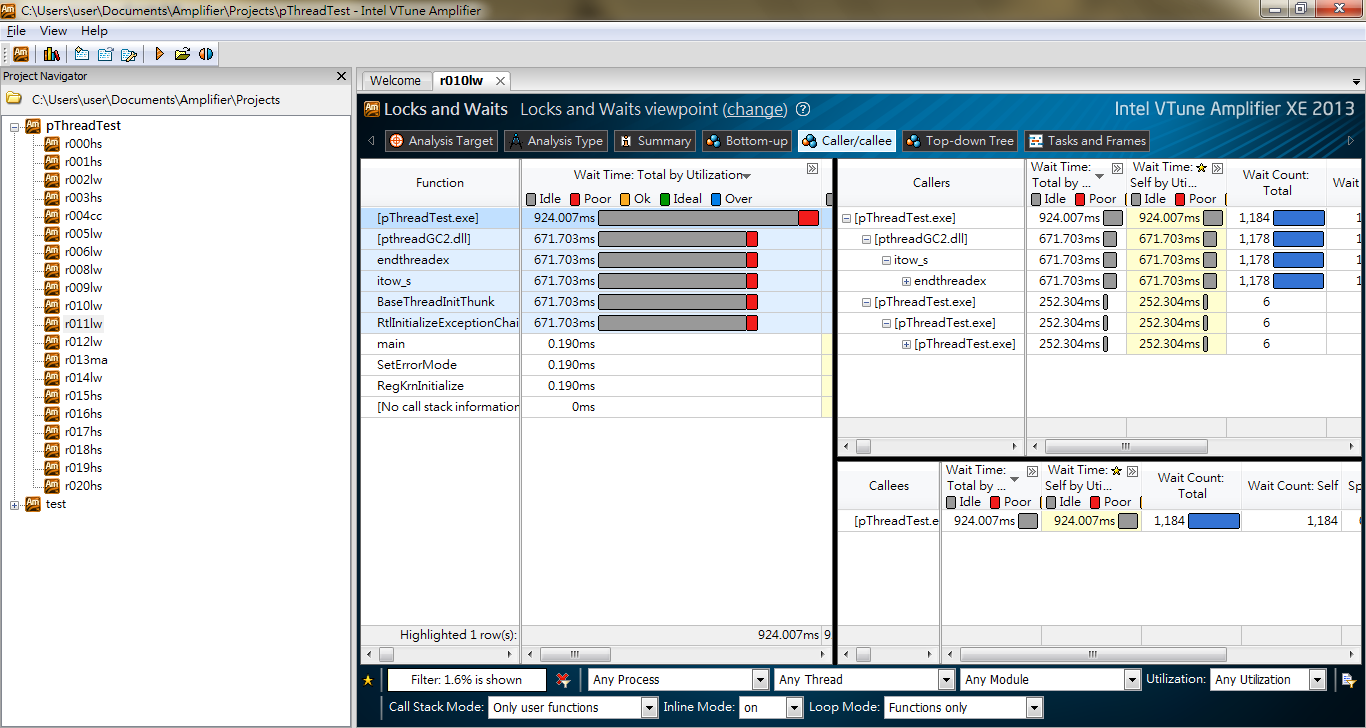
1. We can tell there is too much time waste in Waits (light green bar) on above graph. And each thread holds lock till they finish. This is called coarse-grained.In order to improvethe performance, we may use fine-grained.
2. Look at the graph below. It’s the same program with fine-grained. We can see many transitions between threads. They finish at almost the same time. And most important at all, the execution is 29% less because of the lesser waits.



1. With this tool, you can also see the summary tab for how parallel the program is.



1. With this tool, you can also see the Caller/callee tab or Top-downTree tab for how many time spent in what functions by functiontree.And then decide which function is bottle-neck.



P.S. For Linux User, user needs to use sudoamplifiler-gui to start! Otherwise, there’s no function call stack.

1. Homework

附檔中有nqueens\_bestSerial、nqueens\_naive、nqueens\_parallel，其中第一個是最佳的遞迴寫法，第二個是最直覺的遞迴寫法，第三個是把直覺遞迴寫法平行化。請找出nqueens\_parallel的效能瓶頸在哪裡，比較nqueens\_bestSerial是否比較快，請使用Vtune分析這個程式的效能瓶頸(5分)。

**註：請輸入n=14作為報告中執行的size**

議題，Thread要分出多少個，階層要切多細：

請改善nqueens\_parallel這隻程式，讓他效能變更好，並比較以下

1. 用Coarse Grained的方式平行化和Fine Grained的方式比起來，哪一種比較適合DFS的問題。Coarse Grained就是只針對root底下的第一層做平行化，Fine Grained就是每一層都要做平行化，或介於兩者之間該切到多細執行會比較快?(3分)
2. 什麼情況和什麼時候分多少個thread會比較快，也就是比較n、level(到第幾層還要分成thread)對於執行時間的影響，請比較nqueens\_bestSerial和你的程式的結果 (2分)

關於八皇后問題請參考此網站的敘述。

<http://zh.wikipedia.org/wiki/%E5%85%AB%E7%9A%87%E5%90%8E%E9%97%AE%E9%A2%98>

附檔附上N皇后的程式。

**[Hint]想要有加速的解決辦法和best serial的程式有關聯，一定會用到。**

底下是Coarse和Fine Grained的極端範例，第a題問的就是到底該分得多細?下圖一個顏色就是一個Thread，左邊是3個，右邊是7個threads。

Coarse Grained Fine Grained

# Homework Report Example

1. Vtune的圖，請分析和解釋圖所表現的意義還有效能瓶頸

[以下兩題不用Vtune]

1. 請畫出x軸為輸入的n的大小；y軸為執行時間的圖。比較不同level的差別。例如level=0代表上圖的coarse grained、level=3代表上圖的fine grained。比較時請比較nqueens\_parallel.c就好了。
2. 請畫出x軸為輸入的n的大小；y軸為執行時間的圖。比較n=14、15、16的時候level變化的差別，共三張圖。比較時請比較nqueens\_bestSerial.c 和nqueens\_parallel.c。

P.S. 程式執行結果顯示出來的時間單位是M cycles(百萬cycles)

**APPENDIX**

* 1. **GCC with Pthread in WINDOWS:**

gcc [file name] –o [output file name] -lpthreadGC2 -L"C:\pThread\Pre-built.2\lib" -I"C:\pThread\Pre-built.2\include"

* 1. **GCC with Pthread in Linux:**

gcc [file name] –O1 –g –Wall –o [output file name] -lpthread

* 1. **Dev Cpp with Pthread:**

Tool->Compiler Options->Compiler

In “Add these commands when calling compilers”:

Add : -O2

In “Add these commands to the linker command line”:

Add : -lpthreadGC2

Tool->Compiler Options->Directories

In “Binaries”:

Add : C:\pThread\Pre-built.2\lib

In “Libraries”:

Add : C:\pThread\Pre-built.2\lib

In “C Includes”:

Add : C:\pThread\Pre-built.2\include

In “C++ Includes”:

Add : C:\pThread\Pre-built.2\include

* 1. **Visual Studio with Pthread:**

In project properties

VC++ Directories -> Include Directories:

Insert “C:\pThread\Pre-built.2\include;” at the beginning.

VC++ Directories -> Library Directories:

Insert “C:\pThread\Pre-built.2\lib;” at the beginning.

Linker -> Input -> Additional Dependencies:

Add : pthreadVC2.lib

Debugging -> Environment:

Add : PATH=C:\pThread \Pre-built.2\lib;$(PATH);

* 1. **Coarse-Grained (in data size) Example:**

void \*thread\_function(void \*arg) {

int i;

pthread\_mutex\_lock(&m\_lock);

for (i =0; i <100; i++) {

printf("Thread %d says hi!\n", \*(int \*)arg);

Sleep(1);

}

pthread\_mutex\_unlock(&m\_lock);

return NULL;

}

* 1. **Fine-Grained (in data size) Example:**

void \*thread\_function(void \*arg) {

int i;

for (i =0; i <100; i++) {

pthread\_mutex\_lock(&m\_lock);

printf("Thread %d says hi!\n", \*(int \*)arg);

pthread\_mutex\_unlock(&m\_lock);

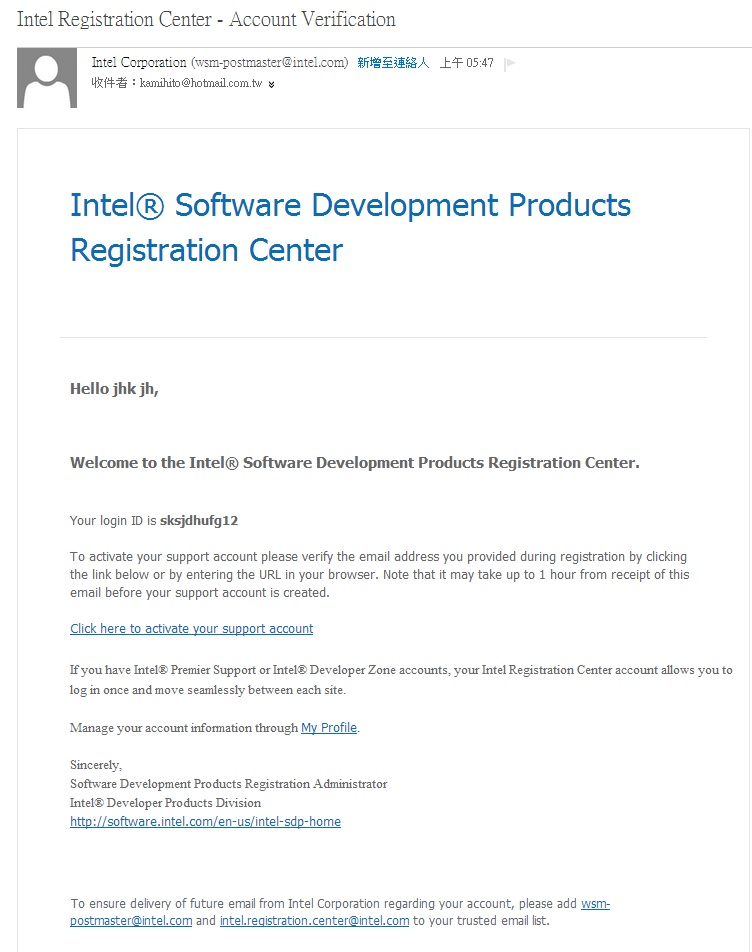
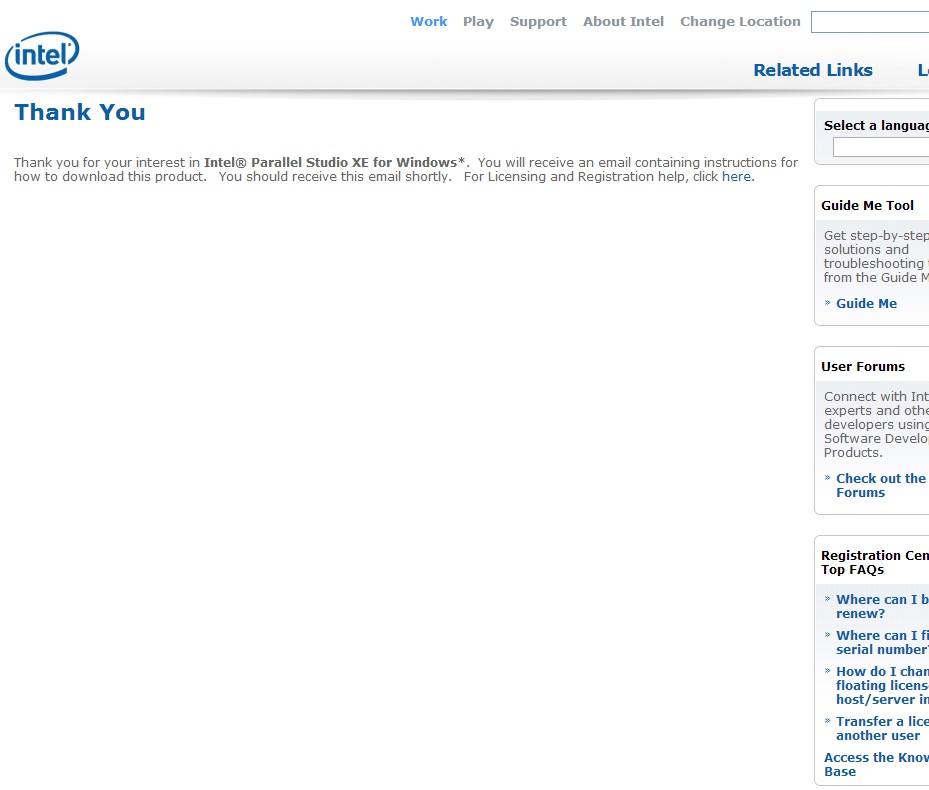
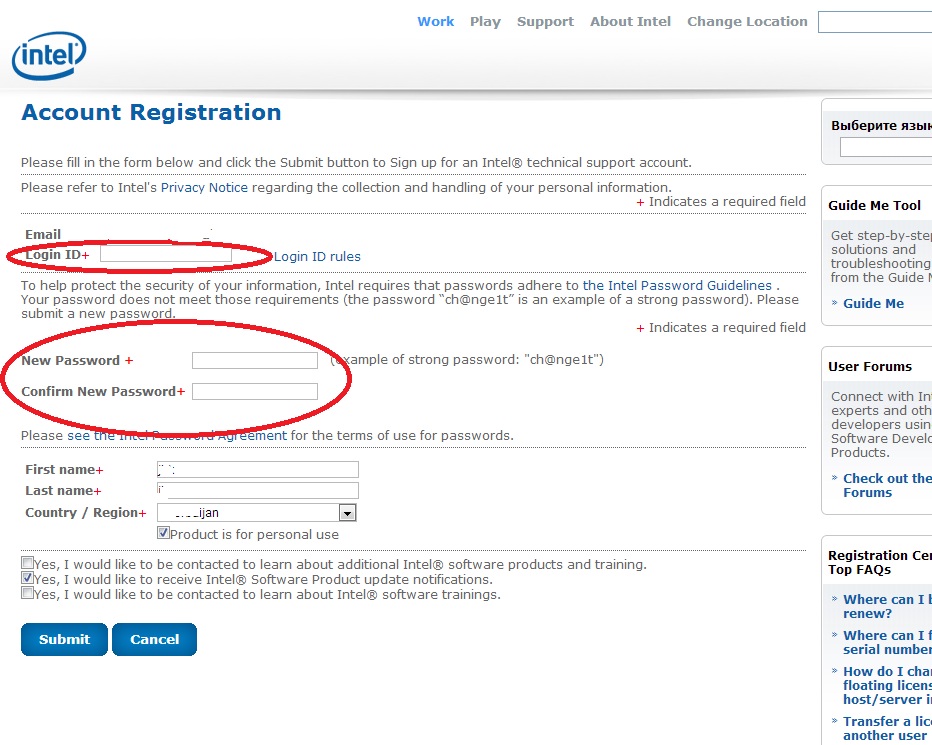
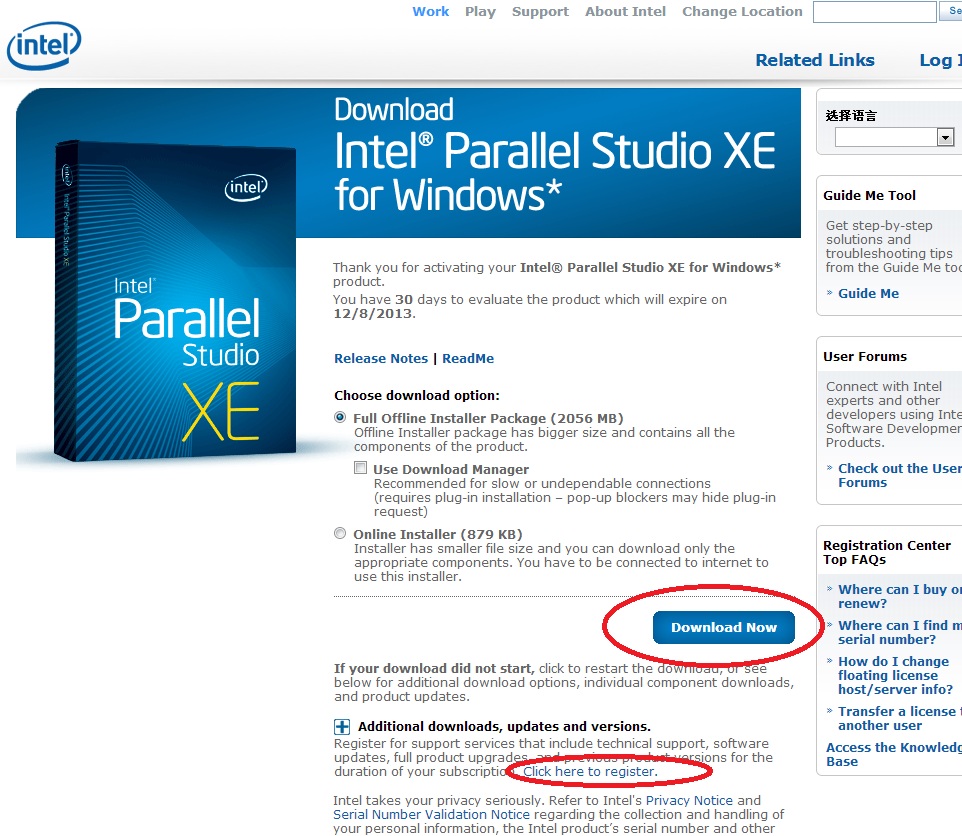
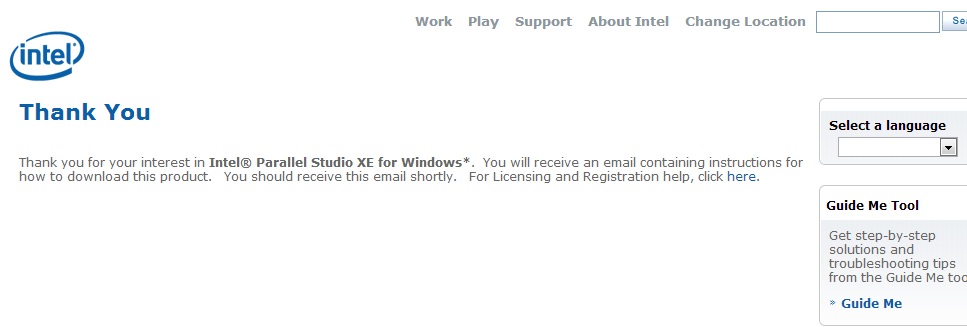
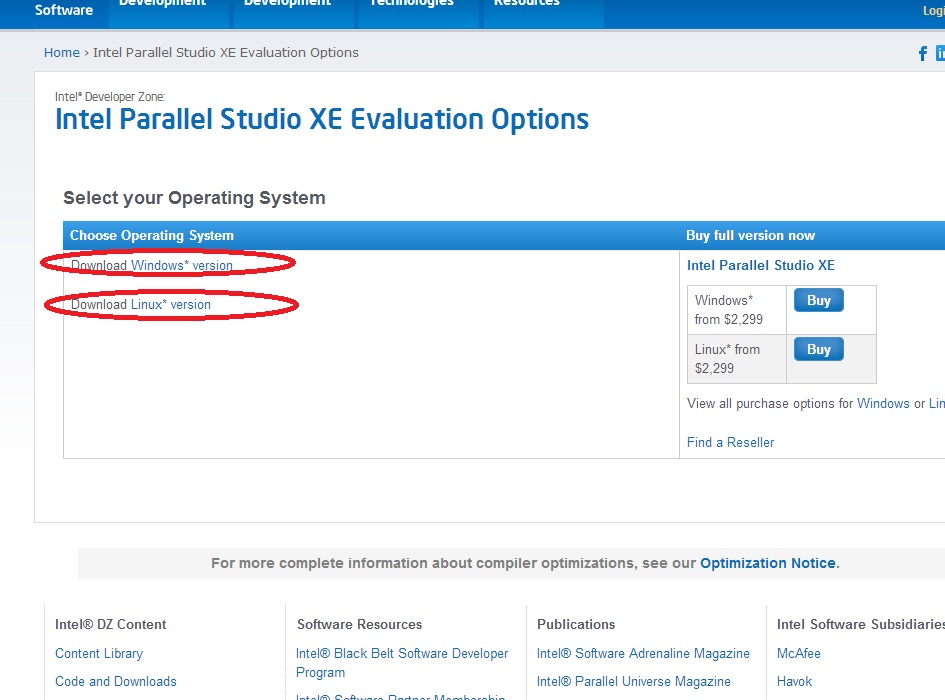
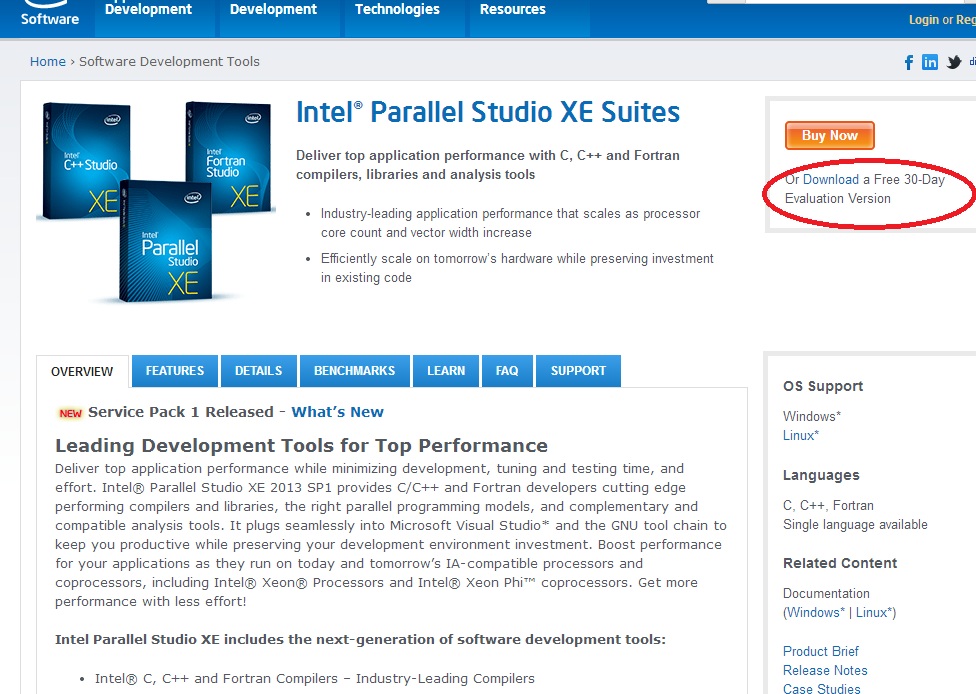
Sleep(1);

}

return NULL;

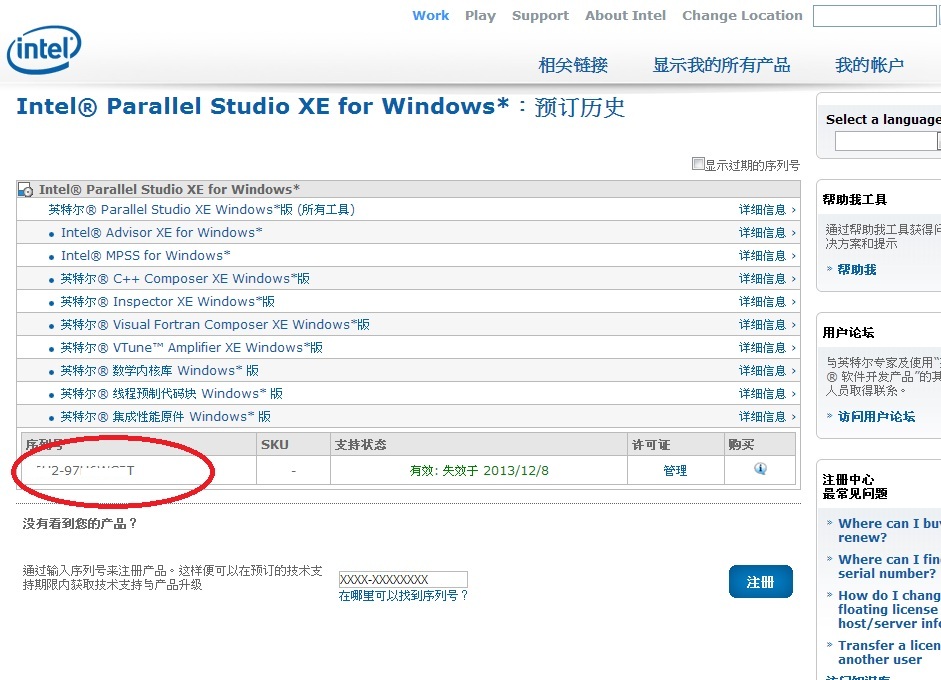
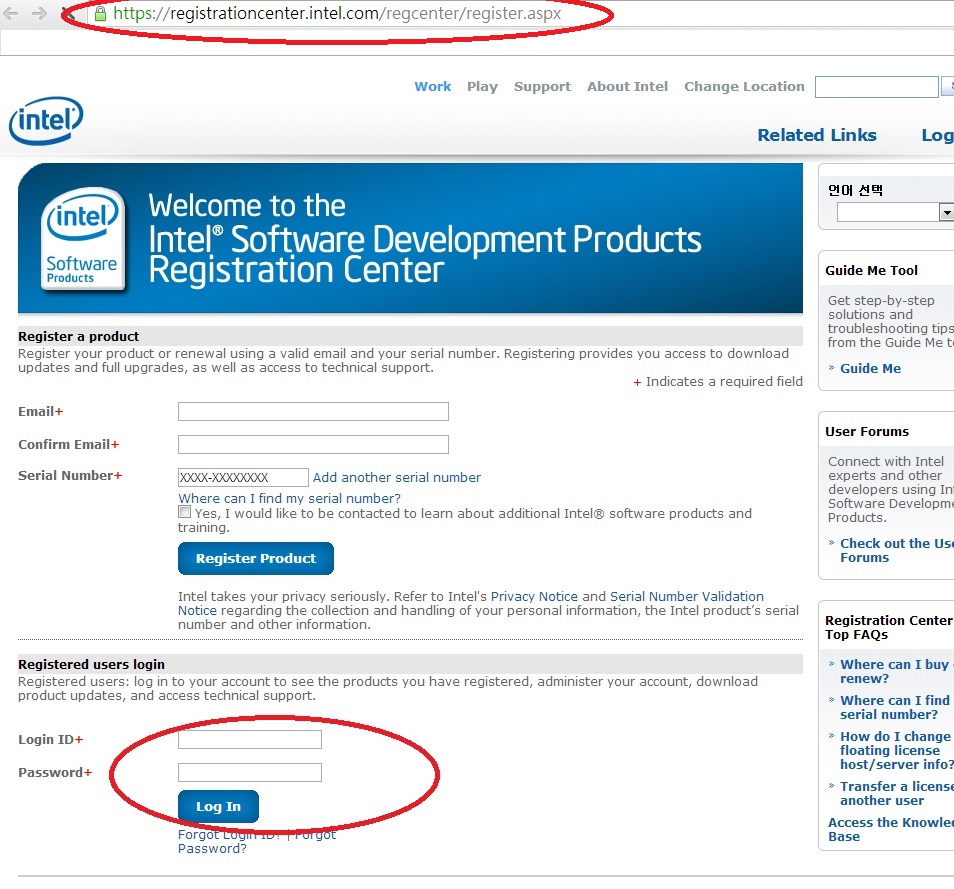
}

* 1. **Registration Steps:**

****

Go to:

https://www.facebook.com/l.php?u=https%3A%2F%2Fregistrationcenter.intel.com%2Fregcenter%2Fregister.aspx&h=7AQFovUuW

****