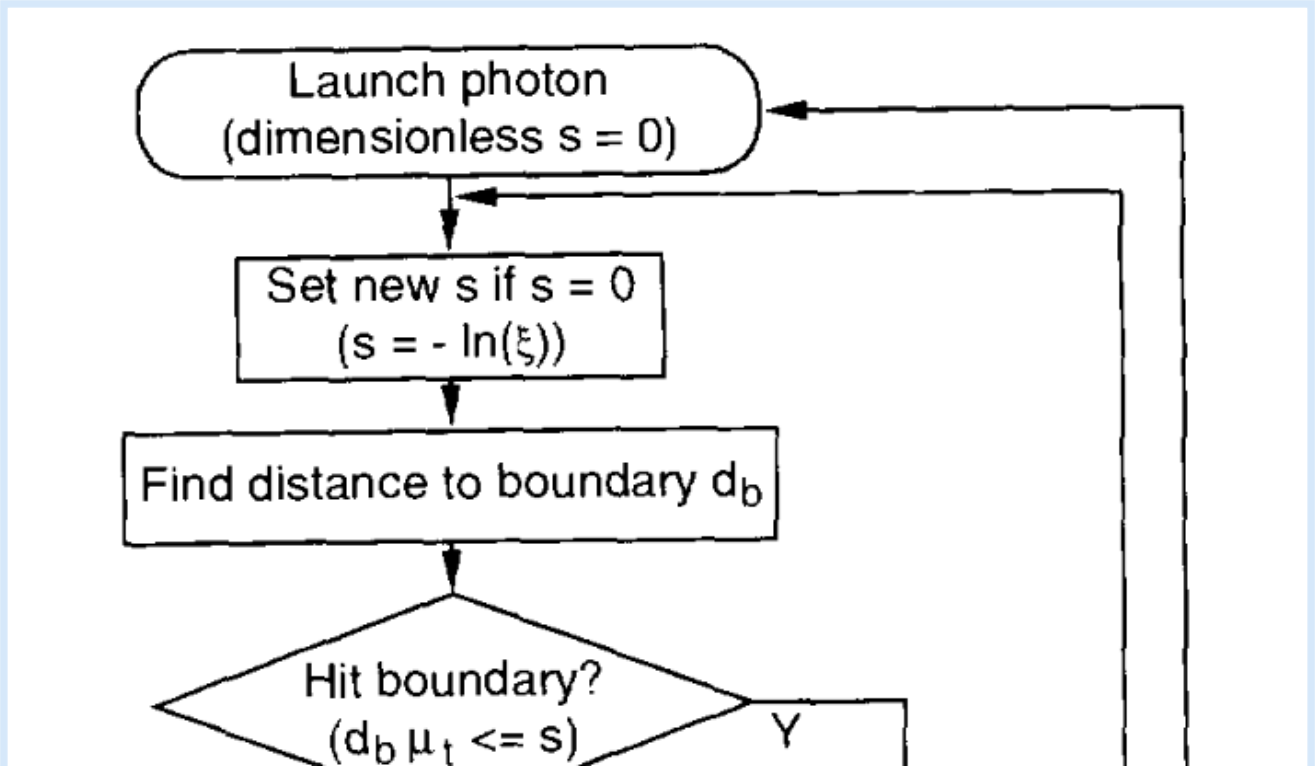
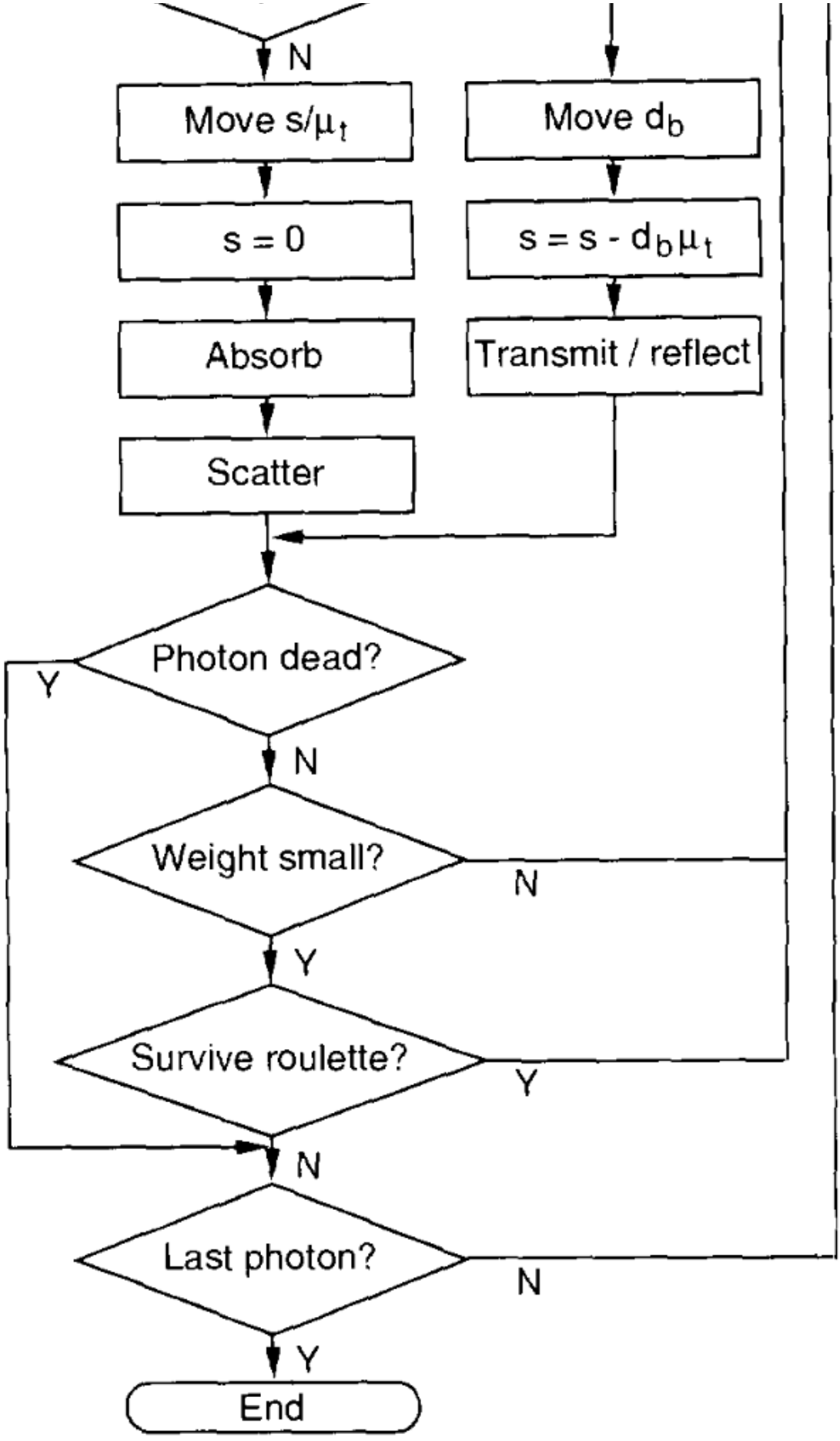
Parallel & Distributed Computing

lab1

MCML

- stands for Monte Carlo modelling of light transport in multi-layered tissues
- the program simulates transport of light by treating each transporting photon as an independent incident, thus makes parallelism a good choice when computing.
- 4 .c codes
  - mcmlmain.c : the whole mcml program starts from here, the main part of this code is the "**DoOneRun**" function, which does the Monte Carlo simulation of a photon
  - mcmlgo.c : defines detailed functions for computing each step of the transporting photon
  - mcmlio.c : deals with input and output parameters for structures defined for "tissue layer" and "photon"
  - mcmlnr.c : some additional functions, has not much to do with the main work
- flow chart of the simulation (ref. paper):




N
Move s/μ_t
s = 0
Absorb
Scatter
Move d_b
s = s - $d_b\mu_t$
Transmit / reflect
Photon dead?
Y
N
Weight small?
N
Y
Survive roulette?
Y
N
Last photon?
N
Y
End

- execution time of program fed with input file: example.mci
    - real 0m18.891s
    - user 0m18.874s
    - sys 0m0.008s

= =

lab2

single thread 執行 serial MCML 的效能分析...

1.使用 gprof＋gprof2dot 進行分析

- 各個Linux distribution都有gprof(GNU profiler)
- 在呼叫gcc連結並編譯程式時加上-pg選項,編譯器就會

1. 主程式入口處呼叫**void monstartup(lowpc, highpc)** 分配memory初始化profile的環境
2. 每個函式的入口呼叫**void _mcount()** 紀錄每個函數的caller和callee的位置
3. 程序退出时呼叫**void _mcleanup()**

- 使用-pg編譯時每個函式都會呼叫mcount,而它的功能就是在記憶體中保存一張函式調用圖,透過函式調用stack查詢callee/caller的記體位置,調用圖也保存了所有與函數相關的呼叫時間，呼叫次數等訊息。
- 跑完編譯出來的執行檔後會自動生成一個 gmon.out
- 此時再下 gprof [option] gmon.out > gprof_output 指令產生可讀的profile：
    - Flat profile:

```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total           
 time   seconds   seconds    calls   s/call   s/call  name    
 29.20    123.35   123.35 1655740118     0.00     0.00  Spin
 19.14    204.19    80.85 1794832151     0.00     0.00  StepSizeInTissue
 12.97    258.99    54.80 1655740118     0.00     0.00  Drop
```

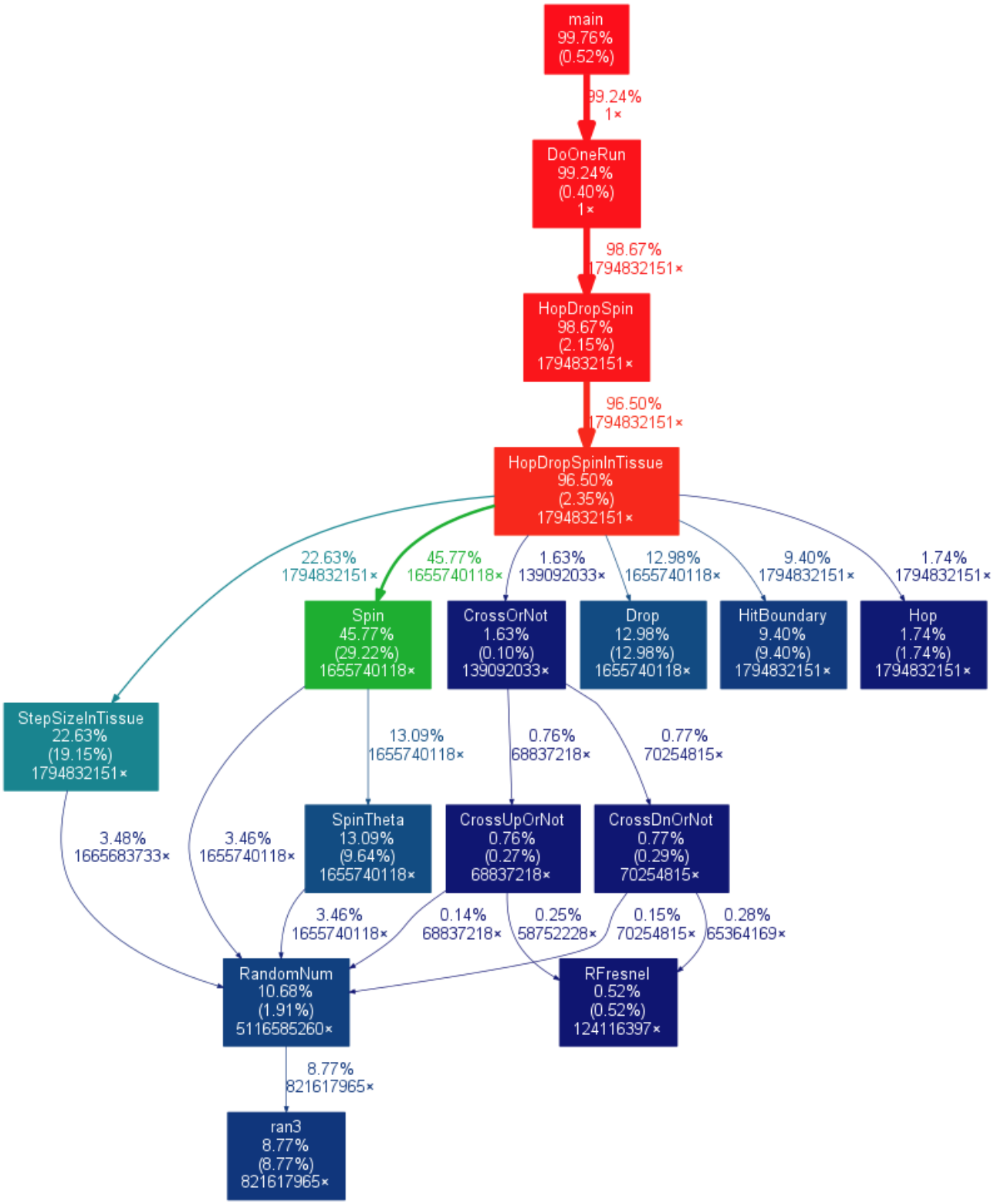
```
  9.63    299.68    40.69 1655740118     0.00     0.00  SpinTheta
  9.40    339.38    39.70 1794832151     0.00     0.00  HitBoundary
  8.76    376.40    37.01 821617965     0.00     0.00  ran3
  2.35    386.32     9.92 1794832151     0.00     0.00  HopDropSpinInTissue
  2.15    395.39     9.07 1794832151     0.00     0.00  HopDropSpin
  1.91    403.47     8.08 5116585260     0.00     0.00  RandomNum
  ......
```

- Call graph:

```
 Call graph (explanation follows)



granularity: each sample hit covers 2 byte(s) for 0.00% of 422.16 seconds

index % time    self  children    called     name
                                                 <spontaneous>
[1]     99.8    2.20  418.96                 main [1]
                1.70  417.26       1/1           DoOneRun [2]
                0.00    0.00       1/1           ShowVersion [62]
                0.00    0.00       1/1           GetFnameFromArgv [57]
                0.00    0.00       1/1           GetFile [56]
                0.00    0.00       1/1           CheckParm [52]
                0.00    0.00       1/2           ReadNumRuns [48]
```

```
                0.00    0.00       1/2           ReadParm [50]
-----------------------------------------------
```

- gprof2dot可將Call graph畫成樹狀圖



- 觀察上圖得知HopDropSpin呼叫次數最多且佔了CPUtime的98.67%,是效能瓶頸所在,然而它本身只佔了2%左右,說明了實際上耗費資源的是更底層的函式

- 以函式本身來說Spin是MCMLprogram中最花時間的函式,其次是StepSizeInTissue
- 由於gprof的原理是編譯時在user program的funtion裡加入指令,已經編好的shared library function的運行就沒辦法profile

2.MCML的程式特性

由前一題我們已經identify出耗費最多CPU時間的function,為了瞭解MCML是哪一種intensive,必須更深入的inspect各個function的細節

從Intel Vtune的profile得知Spin主要是Cosine運算(_cos_avx),而StepSizeInTissue則是Log運算(_ieee754_log_avx)

另一方面MCML的Top5 performance hotspots:

i. cos_avx
ii. ieee754_log_avx
iii. Spin
iv. StepSizeInTissue
v. Drop

由此推知MCML是computation intensive

執行HopDropSpin時photon 的sampling是一顆顆接續進行,且每顆之間又視為獨立,這樣的特性使得平行化處理是可行且有效率的

3. 比較 Valgrind 與 Vtune

- 執行時間:
  - Valgrind >>> 正常執行
    - 不同於Gprof,使用Valgrind時不需重新編譯程式加入外來指令,這是因為程式是跑在一個Valgrind模擬出的CPU上,而同時Callgrind會加入自己的instrumentation code,由於Valgrind完全模擬出user Program上每道instruction的執行,dynamic linked 的 library function也可以被profile,當然程式模擬出的CPU跟額外的監控也讓跑MCML的時間增加10倍之多
  - Vtune ~= 正常執行
    - intel的Vtune同樣的,不需要重新編譯,然而Vtune透過event-based sampling讀取自家CPU中PMU(performance monitoring unit)中的資訊,不需模擬一個執行環境因此大幅降低了profiling的overhead.
- 提供資訊:
  - Valgrind:
    - flat profile
    - caller/callee
    - caller map/ callee map
    - call graph

- Vtune:
  - Valgrind有提供的資訊Vtune都有
  - Top hotspots(function name, CPU time)
  - CPU utilization 以顏色區分效率評分

lab3

- openMP parallelize
  - machine

```
Linux cml22 3.16.0-4-amd64 #1 SMP Debian 3.16.7-ckt2-1 (2014-12-08) x86_64 GNU/L
inux
```

  - CPU : Intel(R) Xeon(R) CPU E5-2650 v2 @ 2.60GHz
  - serial execution:

```
Checking input data for run 1
OMP threads: 32
1023990 photons & 0 runs left, Now 20:13 10/24/16, End 20:13 10/24/16
1023900 photons & 0 runs left, Now 20:13 10/24/16, End 20:13 10/24/16
1023000 photons & 0 runs left, Now 20:13 10/24/16, End 20:13 10/24/16
1014000 photons & 0 runs left, Now 20:13 10/24/16, End 20:13 10/24/16
924000 photons & 0 runs left, Now 20:14 10/24/16, End 20:14 10/24/16
24000 photons & 0 runs left, Now 20:14 10/24/16, End 20:14 10/24/16
Real time:    32.00 sec.
User time:    31.85 sec =      0.01 hr.   Simulation time of this run.
```

  - parallel execution:

```
OMP threads: 32
1023990 photons & 0 runs left, Now 20:03 10/24/16, End 20:03 10/24/16
1023900 photons & 0 runs left, Now 20:03 10/24/16, End 20:03 10/24/16
1023000 photons & 0 runs left, Now 20:03 10/24/16, End 20:03 10/24/16
1014000 photons & 0 runs left, Now 20:03 10/24/16, End 20:10 10/24/16
Real time:    17.00 sec.
User time:   467.85 sec =      0.13 hr.   Simulation time of this run.
```

  - chages made:
    - add openMP commands in function DoOneRun in mcmlmain.c
    - change the while loop to for loop
    - rand() function in mcmlgo.c is not a threadsafe nor a reentrant function

    - use **rand_r()** instead of rand()
    - rand_r() takes a parameter as seed, for each thread I take its omp_thread_num as seed, to make numbers randomly generated
- Real time vs User time

	real	user
sequential	32	31.85
parallel	17	467

    - **realtime** records the total time of execution, real time drops to half when executed in parallel. The machine spawned up to 32 threads, meaning the speedup is much smaller than number of threads created. One possible reason is that 32 threads is too much in this case, the overhead of creating so much threads offsets the benefits of parallelising

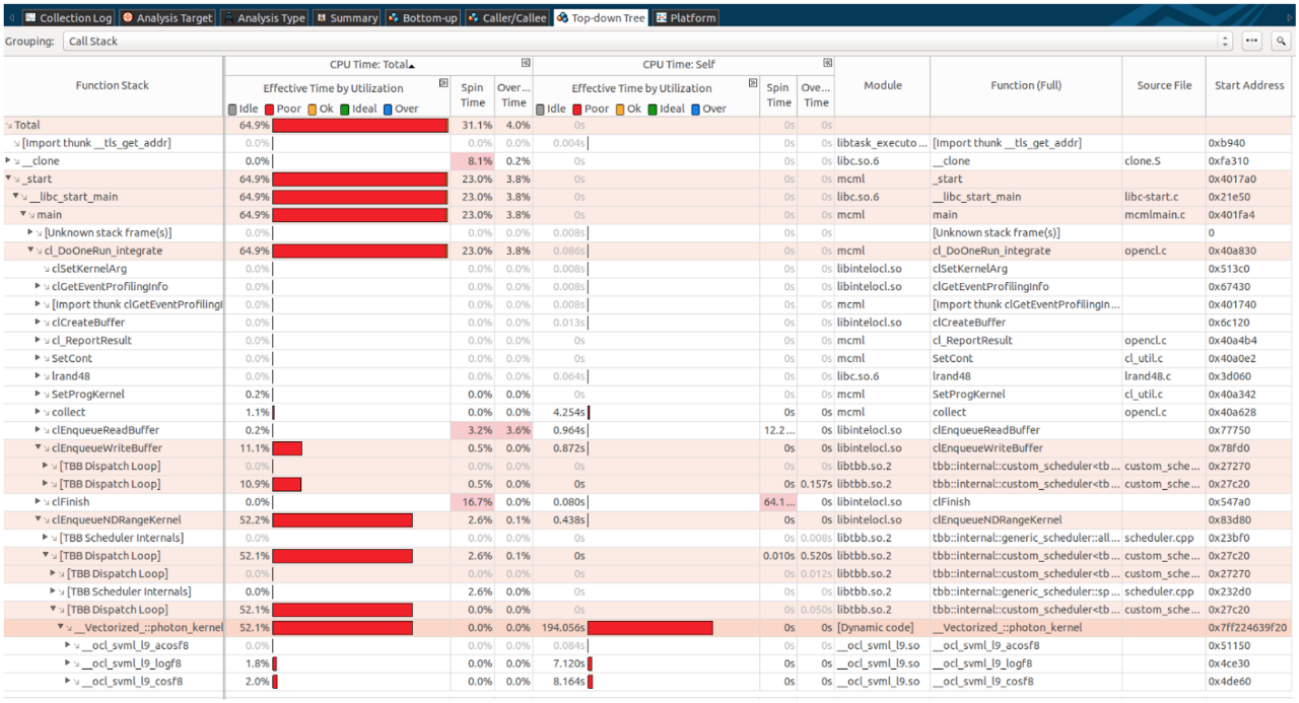
```
OMP threads: 12
1023990 photons & 0 runs left, Now 12:10 10/24/16, End 12:10 10/24/16
1023900 photons & 0 runs left, Now 12:10 10/24/16, End 12:10 10/24/16
1023000 photons & 0 runs left, Now 12:10 10/24/16, End 12:10 10/24/16
1014000 photons & 0 runs left, Now 12:10 10/24/16, End 12:12 10/24/16
924000 photons & 0 runs left, Now 12:10 10/24/16, End 12:11 10/24/16
24000 photons & 0 runs left, Now 12:10 10/24/16, End 12:10 10/24/16
Real time:     9.00 sec.
User time:   100.62 sec =     0.03 hr.   Simulation time of this run.
```

    - compared to execution results on the pdp course site, 12 threads yeild nearly 11x times speedup which is pretty ideal.
    - **usertime** records time spent in the cpu user mode, time spent in each thread is accumulated
    - for sequential execution the real time is >= to the user time (difference in for time for I/O,etc...)
    - for parallel execution the real time is >= **user time/num_of_threads**

LAB4

1.理想加速應該與實體/邏輯 CPU 核心數目(6/12)相同, 為什麼沒有達到6或12倍的加速？

(作業說明裡的參考圖)



觀察以上的profile得知,約有一半以上的執行時間花在TBB Dispatch Loop的處理,而此處的TBB Dispatch Loop是在clEnqueueNDrangeKernel之下,而clEnqueueNDrangeKernel的功能是控制opencl中host與device之間工作數據的傳遞,可推知此處在進行計算工作的分配

程式使用12個working thread的情況下,有一半的時間是host在做scheduling,顯示出過大的overhead,甚至使得ThreadBuildingBlocks變成效能瓶頸,因此達不到理想的平行加速。

2.opencl.c與photon.cl填空

opencl.c 中定義出三種device與host共享記體的模式**cl_mem_flags** 填空處大多是補齊各個buffer的宣告與給值

photon.cl 中填空處是實作mcml裡的hop

3.OpenMP和OpenCL哪個執行效能較佳？為什麼？

預設的memory共享模式是COPY_HOST_PTR,會從host_ptr複製一份data到device宣

告的memory,另一個模式ALLOC_HOST_PTR,是讓host從host-accessible memory中幫device allocate一塊memory,這兩種模式下opencl的執行效能都比openmp差

而相對的USE_HOST_PTR則是可以把host_ptr指向的memory cache起來讓device直接使用,在這個模式下,opencl的效能比openmp還好