基礎的管線化平行程式開發環境

A basic pipelined parallel programming environment

Chiang-Ping Lo(羅江萍)
Fu Jen Catholic University, Taiwan, ROC
Email: mack95j@csie.fju.edu.tw

Wen-Yan Kuo(郭文彥)
Fu Jen Catholic University, Taiwan, ROC
Email: wykuo@csie.fju.edu.tw

1.摘要

多核心處理器(multicore processors)已經無所不在(ubiquitous)。x86 處理器在 2005 年正式進入雙核心時代之後，多核心的應用，已經對一般終端應用產生重大的影響。

本研究利用 Pipeline 平行方式，透過 OpenMP 應用程式介面(API)，建立基礎的管線化平行程式開發環境。應用此開發環境，修改現有的序列式 C 語言程式，來達到 Coarse-Grained Pipeline Parallelism。透過串流應用程式 MPEG-2 Decoding 與資料壓縮程式 bzip2 二個實例，於四核心計算機上測試，證明序列式 C 語言程式透過 Pipeline Parallelism 方式，可利用平行處理提高執行效率。

關鍵詞：Pipeline Parallelism、OpenMP、Software Pipelining、Parallel Programming.

2.緒論

2001 年 IBM 推出首款雙核心處理器 Power4，開啟了處理器多核心化的序幕。昇陽(Sun)於 2004 年也推出雙核心 UltraSPARC IV 處理器，到 2005 年 Intel® 與 AMD 宣布將開發多核心處理器。對程式開發人員來說，開發平行處理的程式，越來越顯得重要。開發新的程式，可以運用平行演算法重新開發平行程式；但之前已開發的序列式語言程式，如何利用平行處理提高效能呢？

一般平行處理程式架構，可區分為三種類型，(1) Task Parallelism：主執行緒(main thread)負責子執行緒(child thread)間的協調合作，子執行緒負責資料處理，各子執行緒動作各自獨立，互相沒有關連性。(2) Data Parallelism：單一流程控制，將處理資料切割為多個資料元素(data elements)，多個執行緒，同時處理這些資料元素。 (3) Pipeline Parallelism：在程式流程迴圈內，切割多個步驟(stages)，利用 Pipeline 方式作為資料交換管道。此運算模式可以保留原程式流程先後順序的相依性，思考模式近似於序列式語言開發方式，所以進入 Pipeline 平行處理的門檻比較低。

然而使用 Pipeline Parallelism 方式進行平行處理，需考慮另一個問題，就是工作切割問題。工作切割方式基本分為(1) Fine-Grained：為 Instruction Level Parallelism (ILP)，於 Compile-Time 由編譯器分析可平行處理的指令。(2) Coarse-Grained：可透過分析工具，經由培訓執行(training run)或由程式開發人員指定特定程式區塊為平行處理區域。本研究利用分析工具或依照程式開發人員對程式熟悉程度，人為手動修改程式，來套用管線化平行程式開發環境，編譯產生平行處理程式碼。

為了測試本文的開發環境的實用性，於四核心計算機上實測了下列二程式：(1)串流程式 MPEG-2 Decoding，使用 3 核心，效能提升 2.62 倍。(2)資料壓縮程式 bzip2，使用 4 核心，效能提升 3.5 倍。由實驗結果得知，序列式 C 語言程式透過 Pipeline Parallelism 方式達到平行處理的

效能提升的數據。

後續的章節簡述如下：(a)相關研究：進行 Pipeline Parallelism 相關研究探討，比較與 Data Parallelism 之間的差異與工作切割等研究課題。(b)管線化平行程式開發環境：說明管線化平行程式開發環境的使用，使用 Triple-DES 加解密程式範例，示範如何修改舊有的序列式程式成管線化平行程式。 (c)實驗與分析：透過 MPEG-2 Decoding 與 bzip2 二個實例測試，比較管線化平行程式在本開發環境執行效率與原有的序列式程式相對比。(d)結論與未來工作：結論總結說明研究成果與未來研究可改進的地方。

3.相關研究

2001 年 IBM 推出首款雙核心處理器，開啟了處理器多核心化的序幕，單一顆處理器中包含兩個以上核心處理器架構 Chip Multi-Processors (CMPs)，現在已經很普遍在個人電腦中看到。本節將從平行處理程式架構的分類，探討為何使用 Pipeline 方式進行平行處理，與利用 Pipeline 方式處理資料交換與同步時，需注意哪些事項，最後討論工作如何切割等問題。

3.1.平行程式架構與平行化階層區分

不管是 CMPs 或是 SMP(symmetric multiprocessor)硬體架構，對於 Multiprogrammed 和 Multithreaded 的程式而言，都是為了充分利用硬體資源，減少執行時間，增加處理能力(throughput)。一般在說明平行處理的程式架構大略可區分三種樣式(paradigm)[4] [5] [11] ：

Master-Slave：Master Thread 起動多個 Slave Thread，分配每一個 Slave Thread 負責一項工作中的部分內容，當所有 Slave Thread 所負責的工作結束後，Master Thread 使用 Barrier 資料同步方式，匯集各 Slave Thread 的結果後，再繼續處理新的工作。Master Thread 主要控制程式流程而 Slave Thread 負責資料處理。此種樣式也稱為 Task Parallelism，如圖 1 所示。

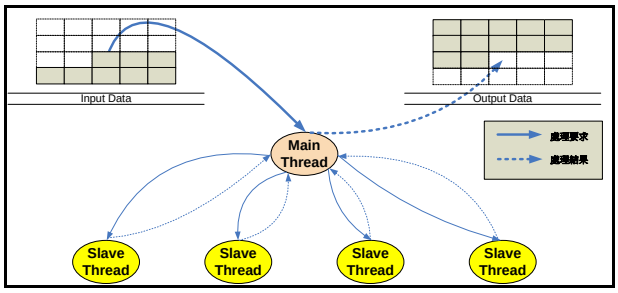


圖 1 平行程式架構-Master-Slave(Task Parallelism)

Workpile：將所有的工作放在 Pool 中，每一個 Thread 從 Pool 抓取工作進行處理，任一 Thread 運算的結果與其它 Thread 執行的工作，沒有相依的關係，直到所有 Thread 沒有任何工作可處理，所以適用於 Stateless[1]的工作性質。Data Parallelism 為 Workpile 其中的一種樣式，如圖 2 所示。

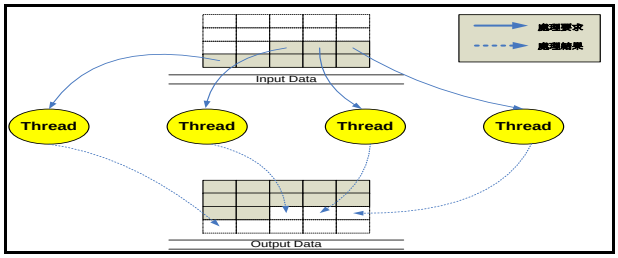


圖 2 平行程式架構-Workpile(Data Parallelism)

Pipeline：使用 Producer/Consumer 式，每一個 Thread 可能同時扮演 Producer 與 Consumer 角色。一個 Pipeline Stage 運算後產生的資料，傳遞給下一個 Pipeline Stage，如同搬運資料，由此站搬運到下一站。由於此種方式可保留原非平行程式資料處理先後的關係，所以適用於 Stateful[2]工作性質。此種方式也稱為 Pipeline

[1] Stateless：表示處理流程無先後關係，或被處理資料之間沒有相依性。例如 World Wide Web server，每一個 request 與前一個 request 是沒有任何關係的。

[2] Stateful：表示處理流程有先後關係，或被處理資料之間有相依性。例如 FTP server。當一個使用者要求傳輸一個檔案，server 假設已經驗證

Parallelism，如圖 3 所示。

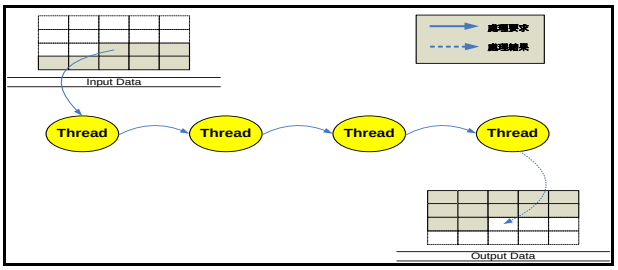


圖 3 平行程式架構-Pipeline(Pipeline Parallelism)

3.2.為何使用管線化

在說明為何使用管線化之前，先簡略說明如何從序列程式修改成為管線化平行程式。第一步在於確認程式中主要的迴圈(main loop)所在，於主迴圈內，將所有的工作切割成多個步驟(stage)。每一個步驟由一個執行緒負責處理，每一個執行緒在單一個核心(core)上執行。執行緒數量如果大於處理器核心數量，將無助於效能的提升[11] 。每一個步驟，只使用屬於自己輸入的資料，更新自己的處理狀態，擁有區域性(locality)操作特性[3] 。所有步驟平行處理，將處理後的結果輸出給下一個步驟。

由上述可知，管線化主要須從主迴圈切割工作著手。如果迴圈內切割出的工作有相依性的關係，可利用 Pipeline 作為相依性的傳遞。演算法則可保持原序列演算法，不必使用新的平行演算法，使得進入平行處理門檻降低，這是使用 Pipeline Parallelism 的好處之一。

管線化的另一個好處為，不論資料可不可切割，可利用切割步驟增加平行度。在 3.1 節討論平行程式架構中，Data Parallelism 是從處理資料切割著手，Pipeline Parallelism 則從程式處理步驟切割著手。表 1 彙整 Data Parallelism 與 Pipeline Parallelism 對處理資料與處理步驟之可切割性與可平行化間關係的比較。Task Parallelism 演算法，因無法沿用原序列演算法，故不列入比較中。

表 1 Data Parallelism 與 Pipeline Parallelism 平行度比較

處理步驟 \ 處理資料	可切割處理	不可切割
可切割多步驟	Data，Pipeline	Pipeline
不可切割	Data	無法平行化處理

當然 Pipeline 方式也有其缺點，如 Bridges et al.[1] 所探討的，各執行緒所分配的流程負載不平均時，則執行效率不理想。如 Software Pipeline(SWP) [6] 使用 Pipeline Parallelism 方式，利用 Loop 內的指令重新排列，建立 ILP 平行處理。雖然 SWP 使用 ILP 技術在效能上表現不錯；但是碰到指令 Latency 差異大時，效能則不是很理想。

Giacomoni et al.[2] 中提出，先進的硬體系統架構規劃上，從微處理器、路由器(router)等，都以 Pipeline 作為設計的基礎。軟體管線(software-pipelining)架構因此能夠配合硬體系統架構，提供更好的效能。所以不管從平行程式軟體架構與硬體系統架構來考慮，Pipeline Parallelism 架構，都適用於平行程式開發。表 2 彙整管線化之優缺點。

表 2 管線化優缺點比較

優點	缺點
(1.) 符合 Pipeline 硬體架構	(1.) 工作切割數量可能少於處理器數量
(2.) 不會發生 Data Races 與 DataLock 情況發生	(2.) 流程負載不平均，執行效率無法得到保證
(3.) 沿用序列式語言演算法，進入平行處理門檻降低	(3.) 需要額外資源與時間進行資料交換與同步處理
(4.) 利用 Pipeline 作為相依性的傳遞	(4.) 狀態同步，反應速度較慢
(5.) 可切割多步驟處理時，可增加平行度	(5.) 處理步驟無法切割時，無法進行平行運算

3.3.管線化資料交換與狀態同步

過使用者、進入他自己的目錄與檔案傳輸模式等狀態下，才能進行傳輸。

開發 Pipeline Parallel 程式需考慮資料傳遞的問題，各執行緒運算結果，透過 Pipeline 進行資料交換或狀態的同步。在資料交換的機制下，必須要有暫存的空間，來存放各管線化步驟(pipeline stage)產生的資料。避免下一個步驟尚未消化資料時，原本的 Producer 又產生新的資料，造成覆寫資料的狀況發生。所以在 Pipeline Parallel 程式中，相鄰步驟資料傳遞，皆需要兩倍的資料空間(double-buffering)[13] 。

在 Pipeline Parallelism，通常暫存空間使用 Queue 方式實踐。produce、consume 兩函式的實作，大至可區分兩種方式：方法一：利用共享記憶體(queue)，修改 Producer/Consumer 程式碼，在同步發生的時間點，將共享記憶體的資料各複製到自己的空間，所以修改程式的人員，必須對程式碼有一定的瞭解，修改難度比較高。方法二：在原始程式中建立包裹函式(wrapper function)，可以避免複雜的程式修改。使用 fork 方式，產生多個子程序，達到平行處理。由於各子程序皆由主程序碼複製一份程式碼，獨立執行，使用獨立的記憶體空間，使用相同的資料結構，所以不需修改任何指令。各程序透過 Inter-Process 溝通機制(如 pipes)。由於資料傳遞透過 Inter-Process 機制，執行效能可能會有所影響。上述的兩個方法，皆為處理 double-buffering 的問題。

Thies et al.[13] 對於 produce、consume 兩函式的實作採用方法二的方式，自動產生平行程式碼，但需程式開發人員指定可平行的程式區域。在四核心處理器的計算機上測試，效能提升為 MPEG-2：2.03 倍與 bzip2： 2.66 倍。本研究對於 produce、consume 兩函式的實作採用方法一的方式。在四核心處理器的機器上測試，效能提升為 MPEG-2：2.65 倍與 bzip2：3.5 倍。

3.4.管線化工作切割

使用 Pipeline Parallelism 方式，就必須面臨工作切割的問題，工作切割是否恰當，直接影響平行執行效率。工作切割可分為 Fine-Grained 與 Coarse-Grained 兩類，下面針對此兩類切割方式條列其特性與優缺點[18] 。

Fine-Grain Parallelism 之優缺點：(a)於單一迴圈時間內，使用漸進式方法完成工作項目。(b)不需深入理解整體程式碼演算法。(c)同時間有許多迴圈同時執行，效能提升有較好穩定性。(d)需要較多的同步處理。

Coarse-Grain Parallelism 之優缺點:(a)於 呼叫階層樹(Call-Tree Level)圖 4 中，執行於較高層級的程式碼，同時間執行迴圈範圍較大。(b)於同一執行時間內，有較多的程式碼被執行。(c)較少的同步處理，減少額外的資源使用。(d)需要深入理解整體程式碼演算法。

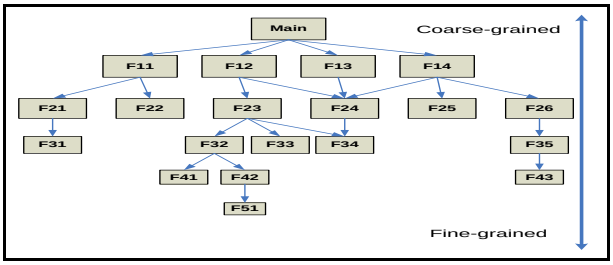


圖 4 呼叫階層樹(Call-Tree Level)

(資料來源引自 http://www.sgi.com/ ，2009)

如何切割各工作呢？自動化的 Fine-Grain Parallelism，是利用 ILP，從指令區段較小範圍追蹤可以平行處理的指令。Coarse-Grained 工作切割方式，可利用程式分析工具[7] [8] [11] [15] ，經過培訓執行，收集、分析執行過程中 Load、Store 指令，建立 Call Graph，紀錄所有函式的呼叫、被呼叫與返回次數。利用 Call Graph，觀察到處理過程中，各函式、資料相依性的關係。在單一 Iterate 之間沒有 Cyclic Dependences 關係存在的前提之下，單一 Iterate 中，相依性可利用 Pipeline 方式切割之間的關

係，進而達到平行處理。保留 Call Graph 相關資訊，若之後於不同執行緒執行，可作為次數平衡的參考。於 Rul et al. [11] 利用這些資訊自動產生平行處理程式或是透過開發人員指定平行處理區域 [1] [13] ，再產生平行處理程式碼。Decoupled Software Pipelining (DSWP) [9] 定位於編譯階段，自動產生平行處理的程式碼，故需要處理器硬體架構相關資訊，只能使用於特定的處理器上[9] [10] 。

4.管線化平行程式開發環境

本文提出一個基礎的管線化平行程式開發環境，主要在不修改原始程式演算法為前提下，讓序列式程式，使用 Pipeline 的方式，利用 OpenMP[19] API 實現平行化處理。利用平行程式開發環境的工作負載平衡功能，解決因工作切割不平均，造成執行效能不佳的問題。透過此開發環境，使用函式指標方式，設定各執行緒程式進入點，開發環境自動處理各執行緒之間資料交換與同步問題，程式開發人員只需專心於各 Pipeline Stage 工作分配與執行緒所執行的工作即可。

本節將利用 Triple-DES 加解密程式為範例，貫穿說明整個程式平行化流程與環境。使用 Triple-DES 為範例，是在不考慮執行效能前提下，單純只是為了說明管線化平行程式開發環境的具體應用方法所使用的範例。

4.1.Triple-DES 序列程式

在說明 Triple-DES 序列程式之前，先簡略介紹 Triple-DES 運作方式。說明 Triple-DES 必須先說明 DES。DES 全名為 Data Encryption Standard，是以 64 bit 金鑰對 64 bit 資料區塊進行加解密的演算法[12] 。而 Triple-DES 是 DES 的加強型，如圖 5 所示，分別使用 3 組 64 bit 金鑰對 64 bit 資料區塊進行加解密的處理。

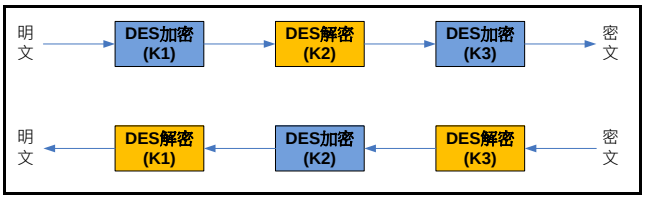


圖 5 Triple-DES 加解密處理步驟

Triple-DES 序列程式，如圖 6 所示。main 函式的處理流程為：將 3 組金鑰放入記憶體中，進入讀取明文資料的迴圈，讀取加密明文資料，進行 Triple-DES 加密處理，輸出加密後的密文。迴圈結束後，程式也隨即結束。

```
1.    int main( void )
2.    {
3.        for( i = 0; i < 3; i++)
4.        {
5.            des_set_key( &ctx[i], DES3_keys[i] );
6.        }
7.
8.        for( i = 0; i < TOTAL_READ_BLOCK_COUNT; i++ )
9.        {
10.           /* 讀取文字內容 */
11.           read_data_block( buf );
12.
13.           /* 加密 */
14.           des_encrypt( &ctx[0], buf, buf );
15.           /* 解密 */
16.           des_decrypt( &ctx[1], buf, buf );
17.           /* 加密 */
18.           des_encrypt( &ctx[2], buf, buf );
19.
20.           /* 寫出加密後的結果 */
21.           write_data_block(buf);
22.
23.       }
24.   }
```

圖 6 Triple-DES 序列式：main 函式

4.2.Triple-DES 管線化程式

此節將說明如何將 Triple-DES 序列程式修改成為管線化的平行程式(pipeline parallel)。首先說明管線化程式函式呼叫階層關係，如圖 7 之 Triple-DES 函式呼叫階層樹，此圖表達各函式之間呼叫樹狀階層性，為靜態資訊。圖 8 為 Triple-DES 管線化之加密程式流程圖，表達程式執行流程。透過這兩張圖例，將進一步說明如何修改 Triple-DES 成管線化的平行程式。

4.2.1.函式呼叫階層與處理流程

在圖 7 中，各函式處理方框右上角之英文字母為函式編號，以方便之後說明使用。函式處理方框內，小括弧內容，則對應到程式碼之

函式名稱。圖 7 表達了函式之間呼叫階層靜態資訊，圖 8 表達程式執行時，各函式執行的動態資訊。

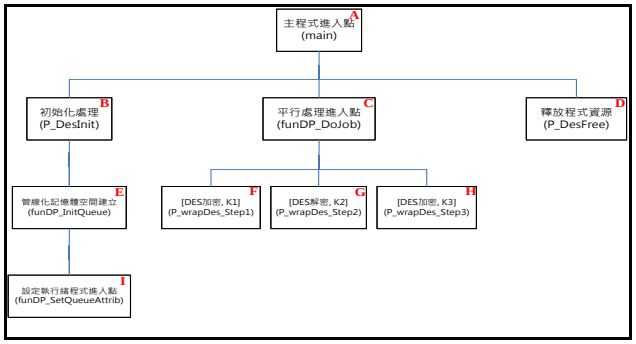


圖 7 Triple-DES 管線化之函式呼叫階層樹

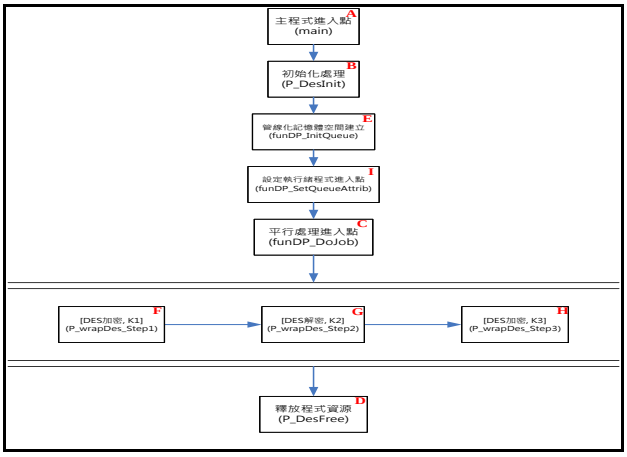


圖 8 Triple-DES 管線化之加密程式流程圖

對於 Triple-DES 管線化的工作切割，我們的規劃是將三次加解密處理，各自為單獨函式負責(F 函式，G 函式，H 函式)，各函式分別配置一個執行緒負責處理。當程式進入平行處理函式(C 函式)後，三次加解密處理步驟會平行化處理，直到所有要加密的資料處理完畢。此時會跳離平行程式區域，最後程式將所使用到的資源，歸還給作業系統(D 函式)。

4.2.2.Triple-DES 管線化之主程式進入點

Triple-DES 管線化的 main 函式可對應到圖 7 與圖 8 之 A 函式，其處理流程為：建立執行時所需的記憶體空間，同時在建立記憶體空間函式中，設定 Pipeline Stage 各執行緒之程式進入點。接著將 3 組金鑰放入記憶體中，進行 Triple-DES 加密平行處理。加密處理結束，釋放所有資源，程式結束。下列針對圖 9 程式碼進行說明

```
1.    int main( void )
2.    {
3.        P_DesInit(); /* B 函式 */
4.
5.        for( i = 0; i < 3; i++)
6.        {
7.            des_set_key( &ctx[i], DES3_keys[i] );
8.        }
9.
10.           /* 開始平行處理 */
11.           funDP_DoJob(); /* C 函式 */
12.
13.           P_DesFree(); /* D 函式 */
14.   }
```

圖 9 Triple-DES 管線化 main 函式：主程式進入點

建立平行程式記憶體空間與設定 Pipeline Stage 執行緒程式進入點

Line 03　建立平行程式所需的記憶體空間，包含 Queue 本身記憶體空間、Queue 內存放的資料元素。
於 P_DesInit 函式中，同時呼叫設定 Pipeline Stage 各執行緒之程式進入點的函式。
＊此函式涵蓋圖 7 與圖 8 之 B 函式，E 函式，I 函式。

3 組金鑰放入記憶體中

Line 05~08　將 3 組 64 bit 金鑰存放入 des_context 資料結構中，供後續加解密使用。

進入平行處理程式碼

Line 11　執行 funDP_DoJob 函式，即進入平行程式碼區域，由單執行緒切換為多執行緒狀態。平行處理加解密三個步驟。當所有明文資料皆加密完成，會離開函式，此時執行狀態由多執行緒切換成單執行緒狀態。
＊此函式涵蓋圖 7 與圖 8 之 C 函式，F 函式，G 函式，H 函式。

釋放記憶體

Line 13　將程式所有用到的資源歸還給作業系統，準備結束程式。

4.2.3.初始化處理

Triple-DES 管線化的 P_DesInit 函式可對應到圖 7 與圖 8 之 B 函式。此函式建立平行程式所需的記憶體空間。處理流程為呼叫開發環境所提供的 Pipeline Queue 初始化函式，建立 Queue 之資料元素(data element)的記憶體空間。下列針對圖 10 程式碼進行說明。

```
1.    int P_DesInit(void)
2.    {
3.        funDP_InitQueue(); /* E 函式 */
4.
5.        for(i = 0; i < TOTAL_BLOCK_SIZE; i++)
6.        {
7.            MpBlock = (LPBLOCK)malloc(sizeof(BLOCK));
8.            if(MpBlock == NULL)
9.            {
10.               printf("Initial data block memory space error.\n");
11.               return FALSE;
12.           }
13.       }
14.   }
```

圖 10 Triple-DES 管線化 P_DesInit 函式：建立資料元素記憶體空間

呼叫開發環境所提供的 Pipeline Queue 初始化函式

Line 03　funDP_InitQueue() 為開發環境所提供的函式，主要建立 Queue 本身記憶體空間與設定各執行緒程式進入點。

建立 Queue 的資料元素記憶體空間

Line 06~13　動態產生 Queue 的資料元素記憶體空間，TOTAL_BLOCK_SIZE 為 Queue 會使用到的資料元素數量。資料元素資料結構，即各 Pipeline Queue 傳遞的內容由程式開發人員自行規劃，開發環境會負責各 Pipeline Queue 的資料傳遞。

4.2.4.管線化記憶體空間建立

Triple-DES 管線化的 funDP_InitQueue 函式可對應到圖 7 與圖 8 之 E 函式。此函式建立 Pipeline Queue 的記憶體空間與設定 Pipeline Stage 各執行緒程式進入點。處理流程為建立 Queue 本身記憶體空間，設定程式執行時，會建立執行緒的數量與設定各執行緒程式進入點。下列針對圖 11 程式碼進行說明。

```
1.    int funDP_InitQueue(void)
2.    {
3.        /* 建立 Parallel Queue */
4.        funDP_CreateDPQueue(QUEUE_NUMBER, QUEUE_SIZE,
      PARALLEL_NUMBER);
5.
6.        omp_set_num_threads(PARALLEL_NUMBER);
7.
8.        // Q0 attrib setup, I 函式
9.        funDP_SetQueueAttrib(0, 0.1F, 0.2F, 0, P_wrapDes_Step1,
      QUEUE_NO_START_QUEUE, 1);
10.       funDP_SetSupportSwitch(0, FALSE);
11.
12.       // Q1 attrib setup
13.       funDP_SetQueueAttrib(1, 0.1F, 0.2F, 1, P_wrapDes_Step2, 1,
      2);
14.       funDP_SetSupportSwitch(1, FALSE);
15.
16.       // Q2 attrib setup
17.       funDP_SetQueueAttrib(2, 0.1F, 0.2F, 2, P_wrapDes_Step3, 2,
      QUEUE_NO_END_QUEUE);
18.       funDP_SetSupportSwitch(2, FALSE);
19.
20.           return TRUE;
21.   }
```

圖 11 Triple-DES 管線化 funDP_InitQueue 函式：建立 Queue 記憶體空間與設定執行緒程式進入點

建立 Pipeline Queue 記憶體空間

Line 04　funDP_CreateDPQueue 函式為開發環境所提供的函式，建立 Pipeline Queue 記憶體空間，傳入的參數順序分別為：QUEUE_NUMBER，建立 Pipeline Queue 的數量；QUEUE_SIZE：每一個 Pipeline Queue 要建立多少資料元素 (data element)；PARALLEL_NUMBER：程式執行時，建立執行緒數量。

設定建立執行緒的數量

Line 6　omp_set_num_threads()，此函式為 OpenMP API，設定當程式執行時，建立執行緒的數量。

設定 Triple-DES 加密第一步驟→加密，程式進入點

Line 09~10　funDP_SetQueueAttrib 函式，主要設定各執行緒程式進入點。此區段程式碼，執行緒被建立後，會執行 P_wrapDes_Step1 函式，即 Triple-DES 加密第一個步驟。

funDP_SetSupportSwitch 函式，設定此 Queue 所存放的工作是否讓其他執行緒取出執行。

＊此函數對應到圖 7 與圖 8 之 I 函式。

加密第二步驟與第三步驟同比照於第一步驟。

4.2.5.平行處理進入點

Triple-DES 管線化的 funDP_DoJob 函式可對應到圖 7 與圖 8 之 C 函式。進入此函式後，執行狀態由單執行緒切換成多執行緒狀態。其處理流程為呼叫 OpenMP API 提供 parallel for 平行指令，各執行緒被執行後，會取自己 Queue 的工作執行，或透過工作負載平衡，支援其他執行緒的工作。以本範例，進入此函式後，加解密三個處理步驟，將平行處理。下列針對圖 12 程式碼進行說明。

```
1.    #pragma omp parallel for
2.        /* Main process */
3.        for(MintPID = 0; MintPID < GintThreadNumber; MintPID++)
4.        {
5.            funDP_Main_Parallel_Process();
6.        }
7.
8.        /* Free memory */
9.        funDP_FreeDPQueue();
10.
```

圖 12 Triple-DES 管線化 funDP_DoJob 函式：執行平行程式程式區域

進入平行處理程式區域

Line 01~06　#pragma omp parallel for 為 OpenMP 提供的 API，執行狀態由單執行緒切換成多執行緒狀態，即開始執行平行程式。每一個執行緒

被執行時，會執行 funDP_DoParallelProcess 函式。

以本範例而言，表示 Triple-DES 加解密三個步驟函式，將平行處理。

釋放平行開發環境所使用的系統資源

Line 09　當執行到此函式，表示平行處理已經結束，執行狀態由多執行緒切換成單執行狀態。此函式會釋放開發環境所使用的系統資源。

在進入平行處理程式區域後，Triple-DES 加解密處理步驟會平行執行，如圖 8 之 F 函式，G 函式，H 函式。下一節將說明加解密處理步驟。

4.2.5.1.Triple-DES 加密步驟一

Triple-DES 管線化的 P_wrapDes_Step1 函式可對應到圖 7 與圖 8 之 F 函式。處理流程為取出 Queue 的資料元素，作為資料交換的空間。讀取明文資料，進行加密處理，將結果存放在資料元素中，最後再將資料元素的位址新增到下一個步驟的 Queue 中。若明文資料已經讀取完畢，發送已無工作訊息給各執行緒。如圖 13 的程式碼。

```
1.    int P_wrapDes_Step1(ElementType *A_Task)
2.    {
3.       LPBLOCK MpBlock = NULL;
4.       ElementType Node;
5.       while(1)
6.       {
7.          if(i < TOTAL_READ_BLOCK_COUNT)
8.          {
9.             funDP_QueueAction(QUEUE_ACTION_POP,0,&Node);
10.            MpBlock = Node.p;
11.
12.            /* Read data block */
13.            read_data_block( MpBlock->MchrBuf );
14.            des_encrypt( &ctx[0], MpBlock->MchrBuf,
      MpBlock->MchrBuf );
15.
16.            funDP_QueueAction(QUEUE_ACTION_PUSH, 1, &Node);
17.            i++;
18.         }
19.         else
20.         {
21.            /* End of file */
22.            funDP_EndQueueProcess(1);
23.            break;
24.
25.         }
26.      }
27.   }
```

圖 13 Triple-DES 管線化 P_wrapDes_Step1 函式：Triple-DES 第一步驟--加密

4.2.5.2.Triple-DES 加密步驟二

Triple-DES 管線化的 P_wrapDes_Step2 函式可對應到圖 7 與圖 8 之 G 函式。此函式負責 Triple-DES 加密第二步驟。處理流程為取出 Queue 的資料元素，此資料元素即上一步驟處理的結果。取出資料內容進行 Triple-DES 第二步驟，解密處理，處理完畢，傳遞給下一個步驟。如圖 14 的程式碼。

```
1.    int P_wrapDes_Step2(ElementType *A_Task)
2.    {
3.       LPBLOCK MpBlock = NULL;
4.
5.       MpBlock = A_Task->p;
6.       des_decrypt( &ctx[1], MpBlock->MchrBuf, MpBlock->MchrBuf );
7.
8.       return TRUE;
9.    }
```

圖 14 Triple-DES 管線化 P_wrapDes_Step2 函式：Triple-DES 第二步驟--解密

4.2.5.3.Triple-DES 加密步驟三

Triple-DES 管線化的 P_wrapDes_Step3 函式可對應到圖 7 與圖 8 之 H 函式。此函式負責 Triple-DES 加密第三步驟與密文的輸出。處理流程為取出 Queue 的資料元素，此資料元素即上一個步驟處理的結果。取出資料內容進行 Triple-DES 第三步驟，加密處理。處理完畢，輸出密文內容。如圖 15 的程式碼。

```
1.    int P_wrapDes_Step3(ElementType *A_Task)
2.    {
3.       LPBLOCK MpBlock = NULL;
4.
5.       MpBlock = A_Task->p;
6.       des_encrypt( &ctx[2], MpBlock->MchrBuf, MpBlock->MchrBuf );
7.       write_data_block(MpBlock->MchrBuf);
8.       funDP_QueueAction(QUEUE_ACTION_PUSH, 0, A_Task);
9.
10.      return TRUE;
11.   }
```

圖 15 Triple-DES 管線化 P_wrapDes_Step3 函式：Triple-DES 第三步驟--加密

上述就是利用管線化平行程式開發環境，將序列程式 Triple-DES 改寫成為管線化平行程式。程式開發人員必須將各管線化步驟處理的工作，修改成為獨立的函式，定義要傳遞資料元素資料結構。其他執行緒狀態同步與資料傳遞則交由開發環境自動處理即可。

5.實驗與分析

本節說明使用兩個程式測試管線化平行開發環境的測試結果，這兩個程式分別為 MPEG-2 decoding[17] ，為一般串流(stream)應用程式與 bzip2 [16] ，為資料壓縮程式。

測試使用的硬體、作業系統、編譯器、軟體分析工具為：(1)硬體設備：CPU：Intel® Core Quad Q6600 2.4GHz。記憶體：DDR2-800 2G*4(8 GB)。硬碟：Seagate STAT2 500GB 7200 RPM。(2)作業系統與編譯器：Microsoft Windows XP Service Pack 3，NTFS 檔案系統，編譯器為 Visual Studio 2005 Professional Edition，OpenMP API v2.0。(3)軟體分析工具 Intel® VTune™ 9.0。

5.1.MPEG-2 Decoding

MPEG-2 是 MPEG 工作小組於 1994 年發佈的視頻和音頻壓縮國際標準。MPEG-2 Decding (MSSG, mpeg2vidcodec_v12) 串流影像解碼處理步驟有先後順序的相依性，所以利用 Pipeline 作為相依性的傳遞，將序列串流影像解碼程式，修改成為管線化的平行程式。利用 Intel® VTune™分析工具，分析出序列程式瓶頸所在。最後比較序列程式與管線化平行程式，執行效率的差異。

利用 Intel® VTune™ 分析工具，測試 MPEG-2 Decoding 序列程式，產生 MPEG-2 Decoding Call Graph (圖 16)與 Call List (圖 17)。由圖 16 可以知道 putbyte，conv420to422，conv422to444 三函式為整個流程中佔用最多執行時間的函式。由圖 17 可知佔用整體執行時間的百分比為：putbyte 約為 38.2%、conv420to422 約為 18.2%、conv422to444 約為 12.8%。利用這樣的資訊，將整體工作切割，分別由 2 個執行緒、3 個執行緒或 4 個執行緒來執行。依各種工作切割方式，比較各執行結果。

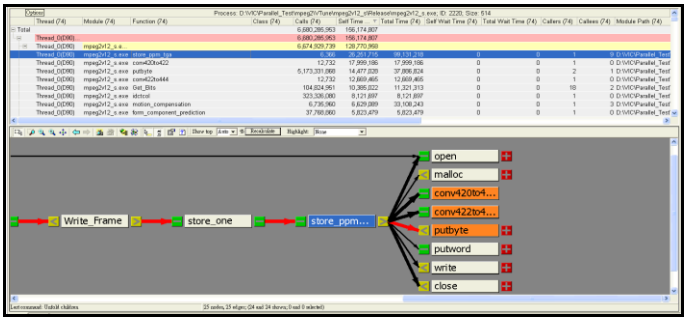


圖 16 MPEG-2 decoder serial code call graph

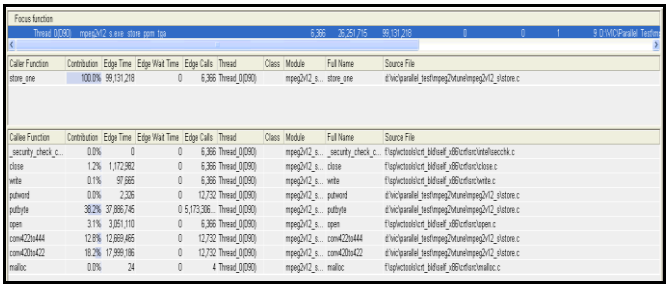


圖 17 MPEG-2 decoder serial code call list

5.1.1.Pipeline Stage 工作分配

透過分析工具可以知道各函式工作負載，此節將進行各 Pipeline Stage 工作負載分配處理。由原始程式碼可知，store_ppm_tga 函式內分別呼叫 conv420to422、conv422to444 與 putbyte 三函式，執行順序如同上述。於 2 個執行緒環境下，因 putbyte 函式(38.2%)與 conv422to444 函式(12.8%)執行時間約佔總執行時間一半，所以將兩函式分配給 1 個執行緒處理，其他函式分配給另一個執行緒處理。於 3 個執行緒環境下，因 putbyte 函式佔 38.2%執行時間，所以單獨 1 個執行緒處理此函式，conv420to422(18.2%)、conv422to444(12.8%)兩函式由 1 個執行緒處理，其他函式分配給另一個執行緒處理。於 4 個執行緒環境下，將 conv420to422 函式(18.2%)、conv422to444 函式(12.8%)與 putbyte(38.2%)函式，各單獨分配一個執行緒，其他函式分配最後一個執行緒。

5.1.2.執行結果與分析

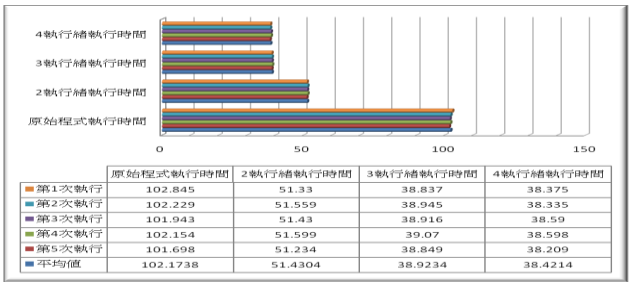
由上節說明的 Pipeline Stage 工作分配，以原始程式為比較基準，由圖 18 與圖 19 可知 2 個執行緒、3 個執行緒執行效能都不錯；但 4 個執行緒效能卻與 3 個執行緒幾乎相同，分析如下：

2 個執行緒環境：Thread1 分配 putbyte 函式

(38.2%)與 conv422to444 函式(12.8%)，約 50%的工作量；相對 Thread0 也是 50%的工作量，所以測試結果約提升 1.9 倍，符合推斷的預期結果。

3 個執行緒環境下：Thread2 分配的 putbyte 函式(38.2%)，是整個流程工作量最大的，雖然 Thread0、Thread1 執行效能高於 Thread2；但整體執行效能，是由 Thread2 執行效能決定。透過推算，Thread2 效能應該為 1/0.38=2.63 倍，而測試結果圖 19 為 2.62 倍，符合預期的結果。

4 個執行緒環境：由 3 個執行緒分析測試結果可知，執行效能是由 Thread2(putbyte 函式)執行效能決定，不可能再提升。除非對 putbyte 函式再做切割步驟動作。但 putbyte 函式為寫檔動作，已經無法再切割，所以執行效能同 3 個執行緒一樣。



	原始程式執行時間	2執行緒執行時間	3執行緒執行時間	4執行緒執行時間
第1次執行	102.845	51.33	38.837	38.375
第2次執行	102.229	51.559	38.945	38.335
第3次執行	101.943	51.43	38.916	38.59
第4次執行	102.154	51.599	39.07	38.598
第5次執行	101.698	51.234	38.849	38.209
平均值	102.1738	51.4304	38.9234	38.4214

圖 18 MPEG-2 decoder 執行時間

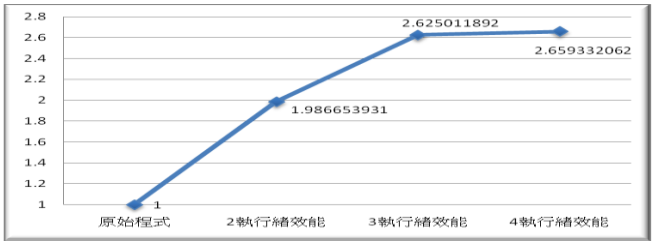


圖 19 MPEG-2 decoder 執行效能

由 Intel® VTune™分析工具，分析 MPEG-2 Decoding 序列程式取得的數據，再比對本研究使用 Pipeline Parallelism 測試結果數據可知，改寫成平行程式，其執行效能提升，符合預期的。這也表示，本開發環境下，Queue 之間資料的傳遞，使用資料元素地址的方式傳遞，可以減到很低的效能耗損。

5.2.bzip2

bzip2 為無失真資料壓縮演算法(lossless data compression algorithm)。使用 Burrows-Wheeler transform 與 Huffman encoding 演算法，與一般使用的 gzip 或 ZIP 相比較，壓縮比率高；但速度慢。對 bzip2 的測試，將與 pbzip2 [14] 做比較。pbzip2 使用平行演算法進行 bzip2 檔案壓縮處理。此測試可知由原序列演算法修改的管線化平行程式與使用平行演算法的平行程式，之間執行效能的差異。

5.2.1.Pipeline Stage 工作分配

此測試的 Pipeline Stage 工作分配，因同時資料可切割而可使用工作負載平衡，所以在工作分配時，以如何方便切割工作步驟為優先考量。下列描述各執行緒所負責工作的分配。

(1.) Thread0：readBlock 函式，負責讀取要處理資料至記憶體空間，透過 Queue 的資料元素傳遞到下一個步驟處理，接著繼續讀取、傳遞，直到檔尾，最後送出結束的通知訊息。

(2.) Thread1：bzBuffToBuffCompress 函式，接收 Thread0 傳遞來的資料，負責資料壓縮處理函式，此執行緒為整個流程，工作負載最重的步驟。

(3.) Thread2：handleMTFProcess 函式，接收 Thread1 處理後的資料。負責資料壓縮步驟中 Move To Front(MTF)步驟，當此步驟處理完畢，表示壓縮所有步驟皆已經完成。

(4.) Thread3：writeBlock 函式，只負責將壓縮好的資料，依照資料先後順序輸出到實體檔案中。

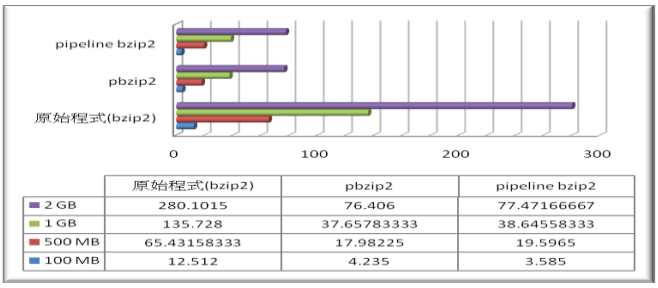
表 3 整理各執行緒所負責的工作範圍。此測試使用 100MB、500MB、1GB 與 2GB，4 個文字檔當作測試輸入檔，於 4 個執行緒環境下，每個測試檔，皆執行 5 次，取執行平均時間作為效能差異比較。

表 3 pipeline bzip2，4 個執行緒步驟分配

執行緒編號	序列執行時間百分比與步驟處理範圍
Thread 0	readBlock(1.1%)
Thread 1	bzBuffToBuffCompress(88.6%)
Thread 2	handleMTFProcess(10%)
Thread 3	writeBlock(0.3%)

5.2.2.執行結果與分析

由圖 20、圖 21 可知，整體來看，兩程式執行效能，pbzip2 使用平行演算法撰寫程式；而 pipeline bzip2 使用原序列程式演算法配合 Pipeline 方式即可達到平行處理功能，兩程式執行效能相近。



	原始程式(bzip2)	pbzip2	pipeline bzip2
2 GB	280.1015	76.406	77.47166667
1 GB	135.728	37.65783333	38.64558333
500 MB	65.43158333	17.98225	19.5965
100 MB	12.512	4.235	3.585

圖 20 pbzip2 與 pipeline bzip2 執行時間

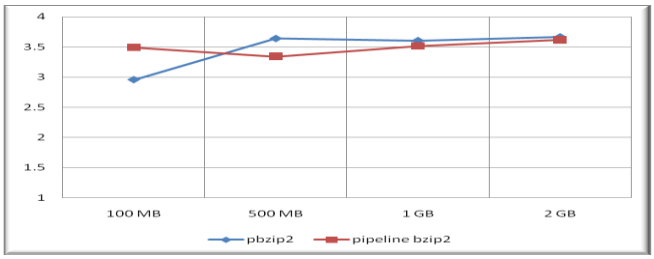


圖 21 pbzip2 與 pipeline bzip2 執行效能

由測試結果可知，使用原程式演算法，利用管線化平行程式開發環境產生的平行程式與使用平行演算法的平行程式，執行效差距甚小。

6.結論與未來工作

多核心處理器的普及化，漸漸改變了軟體開發的規則，下一代程式開發人員，必須瞭解多核心處理器架構與平行演算法，盡可能利用多核心的資源，開發出效能優異的程式。對於既有的程式或歷史悠久的單執行緒程式，要如何移轉到多核心硬體架構，享用多核心所帶來效能的提升呢？本文提出一個基礎的管線化平行程式開發環境。

本管線化平行程式開發環境，使用 Pipeline Parallelism 架構，利用 Pipeline 傳遞資料相依性的能力，達到保持原序列演算法，不必重新開發平行演算法，進而降低進入平行處理門檻。另一方面能夠克服處理資料無法切割時，改變切割工作處理步驟，增加平行度。當然 Pipeline Parallelism 也有其缺點，各執行緒狀態同步與資料交換，需花費額外的資源；工作切割是否平均，決定了整體執行效能。

本文使用二個程式測試管線化平行的效能：程式一： MPEG-2 decoding[17] ，使用 3 核心，效能提升 2.62 倍；程式二：bzip2 [16] ，使用 4 核心，效能提升 3.5 倍。由二個實驗結果得知，透過管線化平行程式開發環境，編譯產生的平行程式，其執行效能，可以被接受的。

在未來的研究與改進上，短期目標是，將修改原程式碼成為管線化平行程式的動作自動化；長期目標，在於研究如何自動切割管線化的工作，盡可能達到減少程式開發人員的介入修改，讓此開發環境更友善，更人性化，提供更好的執行效能。

7.參考文獻

[1] M. J. Bridges, N. Vachharajani, Y. Zhang, T. Jablin, and D. I. August. Revisiting the sequential programming model for multi-core era. In MICRO, 2007.

[2] J. Giacomoni, T. Moseley, G. Price, B. Bushnell, M. Vachharajani, and D. Grunwald. Toward a toolchain for pipeline parallel programming on CMPs. In Workshop on Software Tools for Multi-Core Systems, 2007.

[3] J. Giacomoni, M. Vachharajani, and T.

Moseley. FastForward for concurrent threaded pipelines. Univerity of Colorado at Boulder, Tech. Rep. CU-CS-1023-07, 2007.

[4] M. Gordon, W. Thies, and S. Amarasinghe. Exploiting Coarse-Grained Task, Data, and Pipeline Parallelism in Stream Programs. In Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems, Oct. 2006.

[5] S. Kleiman, D. Shah, and B. Smaalders. Programming with threads. SunSoft Press, Mountain View, CA, USA, 1996.

[6] M. S. Lam. Software pipelining: An effective scheduling technique for VLIW machines. In Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation, pages 318–328, June 1988.

[7] T. Moseley, D. Grunwald, D. A. Connors, R. Ramanujam, V. Tovinkere, and R. Peri, Loopprof: Dynamic techniques for loop detection and profiling, in Proceedings of the 2006 Workshop on Binary Instrumentation and Applications (WBIA), 2006.

[8] T. Moseley, D. A. Connors, D. Grunwald, and R. Peri, Identifying potential parallelism via loop-centric profiling, in Proceedings of the 2007 International Conference on Computing Frontiers, May 2007.

[9] G. Ottoni, R. Rangan, A. Stoler, and D. I. August. Automatic thread extraction with decoupled software pipelining. In MICRO, 2005.

[10] R. Rangan, N. Vachharajani, M. Vachharajani, and D. August. Decoupled software pipelining with the synchronization array. PACT, 2004.

[11] S. Rul, H. Vandierendonck, and K. De Bosschere. Function level parallelism driven by data dependencies. In Workshop on Design, Architecture and Simulation of Chip Multi-Processors, 2006.

[12] William Stallings. Cryptography And Network Security Principles and Practice 2nd edition. pp.65-75, 1995

[13] W. Thies, V. Chandrasekhar, and S. Amarasinghe. A practical approach to exploiting coarse-grained pipeline parallelism in c programs. In MICRO '07: Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture, pages 356–369, 2007

[14] GILCHRIST, J. Parallel bzip2 (pbzip2) data compression software,http://compression.ca/pbzip2/

[15] Intel® Software Network, Intel® VTune™ Performance Analyzer, http://www.intel.com/cd/software/products/asmo-na/eng/239144.htm

[16] Julian Seward, bzip2/libzip2(bzip2-1.0.5), http://bzip.org/

[17] MPEG Software Simulation Group, MPEG-2 Video Codec(mpeg2vidcodec_v12), http://www.mpeg.org/MSSG/

[18] Silicon Graphics International. http://www.sgi.com/

[19] The OpenMP Specication. http://www.openmp.org/.