第四章　設計與實做

第一節 修改資料結構

2006 年度學長們所研發的「深象」，其變數大都是宣告成全域變數，不過這樣不利於平行化。

因為平行化最重要的事情就是將資訊分享，讓其它的 Process，可以知道它所要負責搜尋的盤面、以及負責的走步。

所以平行化的第一個步驟，則是先設置一個 struct TREE，將搜尋所需要的變數，全部搬移到 TREE 裡面，這樣子當要分享資訊的時候，我們就將 TREE 裡面的資訊，複製到另一個 Process 的 TREE 的變數。

然而，並不是設置 TREE 結構就足夠了，因為我們是利用多行程 (Multi-Process)，的方式來做，每個 Process 之間的溝通就必須用共享記憶體 (Shared Memory)的方式，當複製盤面的時候，才能讓 Process A 將盤面複製資訊到 Process B 的 TREE。所以我們還需要設置一個 struct Shared，來存放共享的資料，比如說：TREE…等，當需要使用共用資料的時候，需要注意臨界區間(Critical Section)問題，變數 lock 就是用來限制 Shared 的存取。表 4 說明 TREE 資料內容，表 5 說明 Shared 所宣告的變數。

同形表也是需要一個需要修改的部份。當程式是循序版本的時候，並不會有臨界區間問題，如圖 12 舊版的深象同形表。不過當平行化之後，每個 Process 都會去存取同形表(使用 hashtable 實做)，所以需要使用 lock 來控制存取限制，雖然使用 lock 會造成負擔，但是為了解決臨界區間問題，這是必要的。圖 13 為修改

過後的同形表，新增一個欄位 lock，讓要存取同一個 entry 的 process 不會造成競爭狀況(race condition)。資料結構分別為表 6、表 7。

```
struct TREE{
    SEARCHINFO info;                  //盤面資訊….等
    SEARCHCONTROL control;            //搜尋控制：如深度
    Lock lock;
    struct TREE siblings[CPUS],       //紀錄幫忙的 process
                *parent;              //紀錄父親 process
    char stop;                        //停止搜尋
    int thread_id;                    //process id
    int nprocess;                     //紀錄有幾個 processes 在幫忙
};
```

表 4 TREE 資料內容

```
struct Shared{
    TimeControl time_control;              //時間控制
    PonderInfo ponder_info;                //後台思考
    unsigned int parallel_splits;
    unsigned int parallel_aborts;
    unsigned int max_split_blocks;         //每個節點最多用幾個 process 搜
                                           //尋
    unsigned int splitting;
    int smp_idle;                          //閒置 process 數量
    int smp_threads;                       //計算幾個 process 已經產生
    int initialized_threads;               //計算有幾個 process 已經初始化
    int max_threads;                       //最多 process 數量
    int split_at_root;                     //限制要不要在 root 分
    TREE *local[MAX_BLOCKS + 1];           //split block
    TREE *thread[CPUS];                    //每個 process 放置工作的地方
    lock_t lock_smp, lock_io, lock_root;   //lock
};
```

表 5 Shared 資料內容

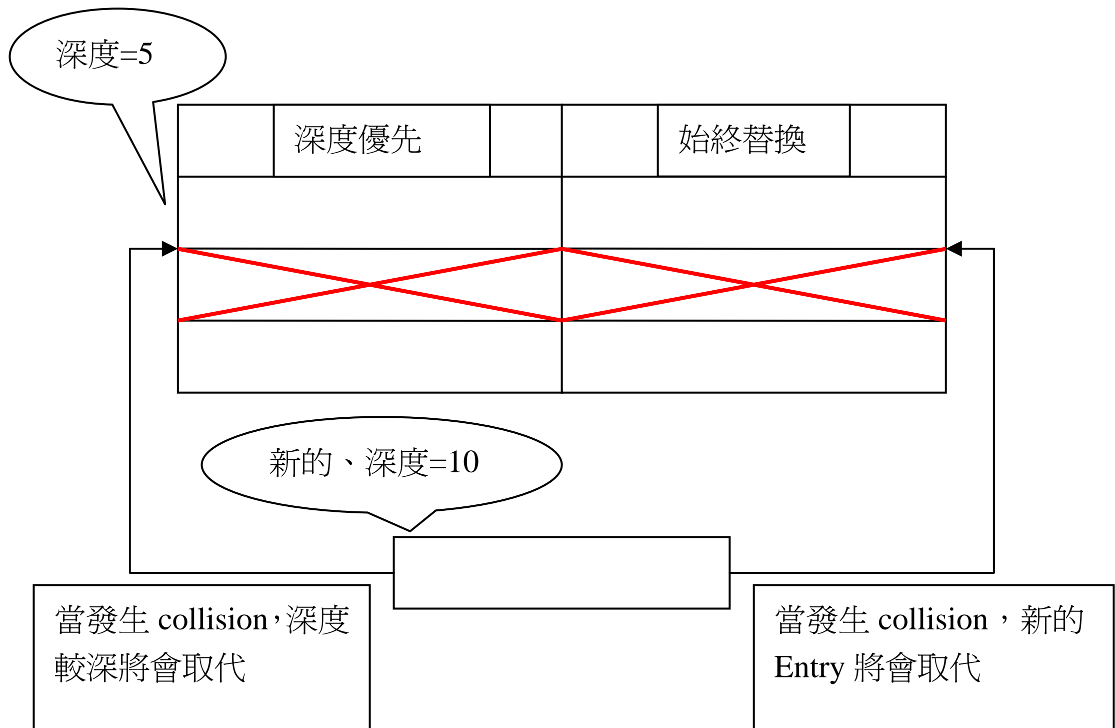

圖 12 舊版的深象的同形表

當新的 Entry 要插入同形表發生碰撞的時候，會先比較深度優先那一半的資訊，當新的 Entry 深度比較深，則發生碰撞的那一個位置則會更新為新的資訊，如果新的 Entry 深度比較淺，則保留同形表內的資訊。始終替換那一部份，則是遇到碰撞，都會更新成新的 Entry 的資訊。

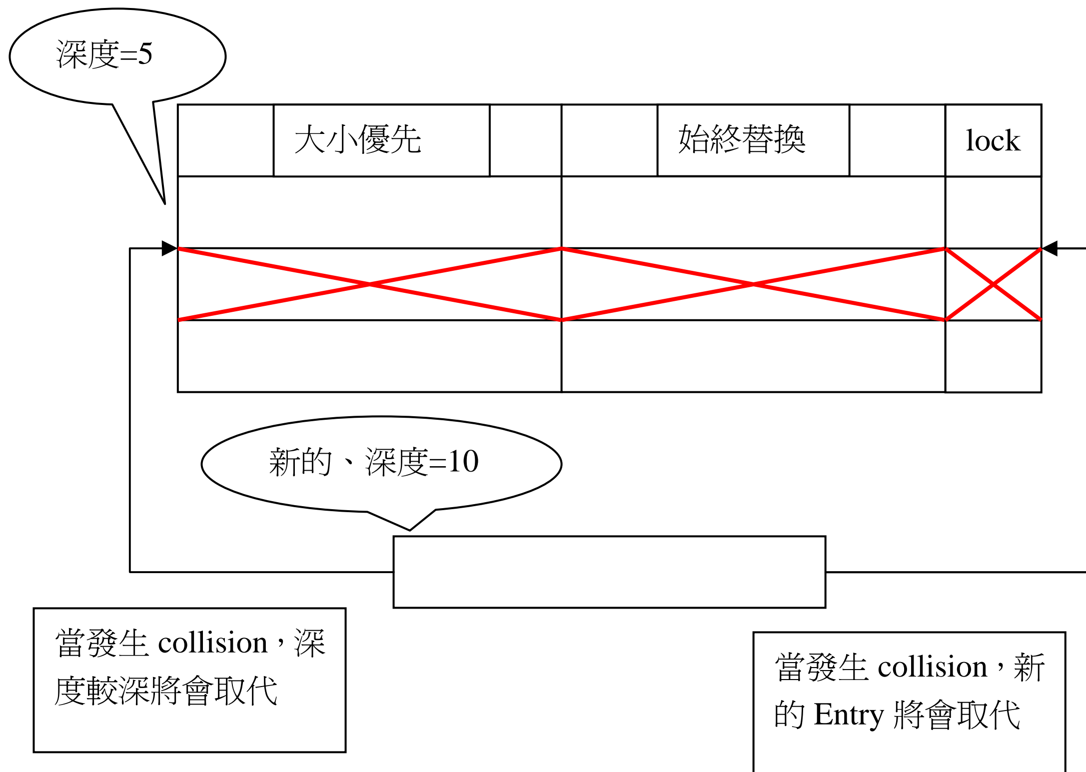

圖 13 修改過後的深象同形表

因為 Process 共用同形表，沒有使用 lock 會發生臨界區間(Critical Section)的問題，所以改良過後的同形表，如圖 13 所示，比圖 12 多了一個 lock。當 Process 要存取同形表其中一個 Entry，則會先去判斷有沒有 Process 正在存取，避免臨界區間的問題。

```
struct HashEntry{
        int key;
        short value;
        UCHAR depth;
        UCHAR flag;
        UCHAR extflag;
        short age;
        short bm;
};
```

表 6 舊版深象同形表的資料結構

```
struct HashEntry{
        int key;
        short value;
        UCHAR depth;
        UCHAR flag;
        UCHAR extflag;
        short age;
        short bm;
        int lock;          //lock，防止 race condition
};
```

表 7 深象同形表修改過後的資料結構

第二節 修改搜尋架構

之前 2006 年所研發的深象的搜尋架構如圖 14。

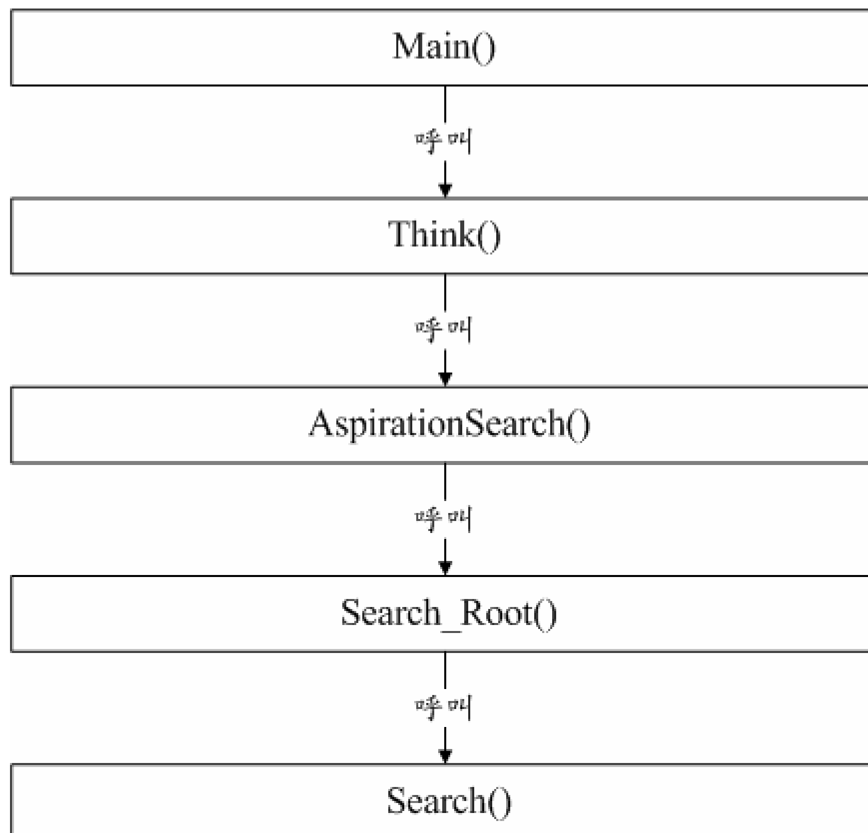

圖 14 舊版的深象搜尋架構

Think()：如同 Crafty 的 Iterate()，工作是疊代加深。

AspirationSearch()：期望搜尋，當發生 fail high 或是 fail low 的時候更改根節點 Alpha_Beta 窗口。

Search_Root()：初始化搜尋統計的變數，如：搜尋節點數，每秒搜尋幾個點(NPS：nodes per second)...等。之後呼叫 Search()。

Search()：使用 Nega_Scout 演算法搜尋。並且使用空步、以及查詢同形表(Transposition Table)。

當要將程式平行化，我們第一個步驟就是修改搜尋的架構。

因為深象的 Search_Root()並沒有多做什麼處理，就直接呼叫 Search()，由 Search()來搜尋所有的節點。所以我們先修改 Search_Root()的程式碼，讓 Search_Root()使用 Nega_Scout 演算法搜尋根節點的走步，除了根節點的走步就是使用 Search()來搜尋。將根節點以及其它子節點分別開來。

因為每個節點第一個走步的結果回傳之後，如果有 Process 閒置，就可以將工作交給它，一起搜尋。所以我們還在 Search_Root()、Search()呼叫 Thread()。

以下為修改後的虛擬碼。

```
Search_Root(tree, alpha, beta, depth){
    Lock(hashtable_entry);          //hashtable 是 global 變數，所以要 lock
    ProbeHash();                    //查詢同形表
    Unlock(hashtable_entry);
    a=alpha;
    b=beta;
    while(nextmove()){              //選擇走步
        DoMove(tree);
        value = -Search(tree, -b, -a, depth-1);   //呼叫搜尋
        if ((a < value) && (value < beta) && !firstmove){
            value = -Search(tree, -beta, -value,depth-1);
        }
        UndoMove(tree);
        if(value > a)
            a=value;
        if(a ≧ beta)
            break;
        b=a+1;
        if (shared->split_at_root && shared->smp_idle) //判斷要不要 split
            Thread(tree);  //將工作分出去，喚醒閒置的 process
    }
    return a;
}
```

當進入 Search_Root()，先搜尋同形表裡面是否有搜尋過的資訊，當沒有找到符合的資料，則呼叫 Search()搜尋底下的節點。當第一個走法已經搜尋完畢，則可以判斷到底要不要將工作分配出去。

```
Search (tree, alpha, beta, depth){
    if (!depth) //當深度為 0 時，呼叫寧靜搜尋
        return Quiescence(tree, alpha, beta, last_move);
    Lock(hashtable_entry);  //hashtable 是 global 變數，所以要 lock
    ProbeHash();            //查詢同形表
    Unlock(hashtable_entry);
    Null_Move_Search();     //空步搜尋
    a=alpha;
    b=beta;
    while(nextmove()){      //選擇走步
        DoMove(tree);
        value = -Search(tree, -b, -a, depth-1);   //呼叫搜尋
        if ((a < value) && (value < beta) && !firstmove){
            value = -Search(tree, -beta, -value,depth-1);
        }
        UndoMove(tree);
        if(value > a)
            a=value;
        if(a ≧ beta)
            break;
        b=a+1;
        if (shared->split_at_root && searched_moves && depth>SPLIT)
        {   //判斷是否要 split
            Thread(tree); //將工作分出去，並且喚醒閒置 process
        }
    }
    return a;
}
```

當進入 Search()，會先判斷搜尋深度是否為 0，當搜尋深度為 0 的時候則呼叫寧靜搜尋(Quiescence())，並且回傳分數。當搜尋深度不是 0 的時候，則進行搜尋。當第一個走步回傳分數之後，就可以判斷是否要將工作分配給閒置的 Process。

以上的演算法皆有查詢同形表，注意都要使用 Lock(hashtable_entry)，避免多個 Processes 同時存取，造成資料錯誤。

第三節 新增函式

除了修改搜尋的架構之外，還需要新增函式來完成平行化的工作，平行化的架構如圖 15。

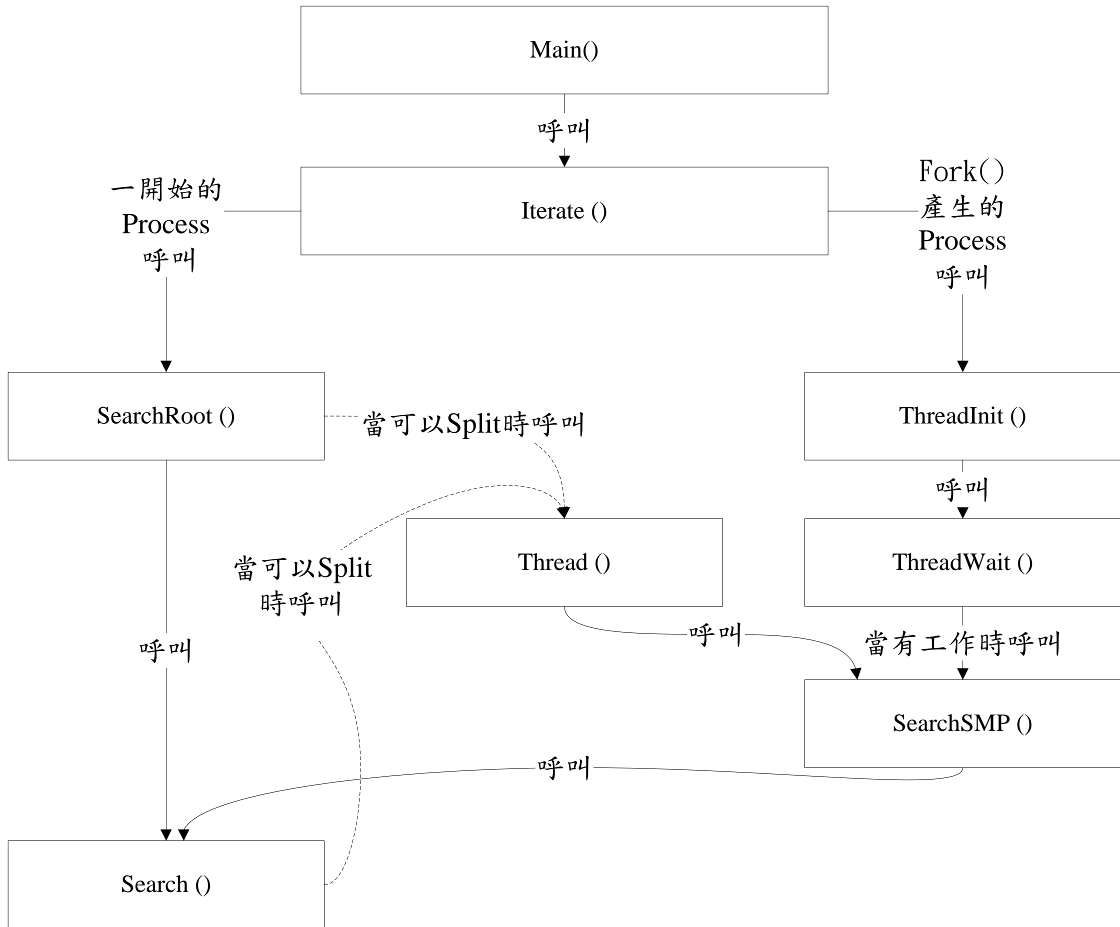

圖 15 平行化深象的搜尋架構

為了完成平行化的架構，我們新增了 SearchSMP()、ThreadInit()、ThreadWait()、ThreadStop()、Lock()、Unlock()這些函式。

ThreadInit()：各個 Process 初始化資料結構。

ThreadWait()：沒工作的 child Process，在此等待工作。

ThreadStop()：當發生 fail high 的時候，呼叫 ThreadStop()，暫停搜尋。

SearchSMP()：一起搜尋同一個 Tree 的 Process，在這個方法中，取得父親節

點的走步，並且呼叫 Search()，搜尋各個走步的變化。

Lock()：當要使用共用變數(shared)或是查詢同形表的時候使用，避免臨界區間的問題。

Unlock()：解除 Lock()。

```
Thread (tree) {
    Lock(lock_smp);
    nblocks=0;
    for (proc = 0; proc < max_threads && nblocks < max_thread_group;
      proc++) {
        if ( thread[proc] = = 0) {
            block = CopyToSMP(tree, proc);        //將工作複製給
                                                  //proc
            if (!block)
                continue;
            nblocks++;
            tree->siblings[proc] = block;         //紀錄將工作分配給
                                                  //proc
            block->thread_id = proc;
            block->parent = tree;
            tree->nprocs++;
        } else
            tree->siblings[proc] = 0;
        tree->search_value = tree->value;
        if (!nblocks) {
            Unlock(shared->lock_smp);
            shared->thread[tree->thread_id] = tree;
            shared->splitting = 0;
            return (0);
        }
    }
    for (proc = 0; proc < max_threads; proc++)
        //將工作分配，喚醒閒置的 process
        if (tree->siblings[proc])
            shared->thread[proc] = tree->siblings[proc];
    Unlock(lock_smp);
    ThreadWait(thread_id, tree);
}
```

Thread()檢查全部的 Process，如果是閒置(thread[proc] = = 0)，就將盤面資訊複製(CopyToSMP(tree, proc))給它，Process proc 發現它有工作，就會進入

SearchSMP()，進行搜尋。

```
ThreadInit (tid) {
    for (i = 0; i < MAX_BLOCKS_PER_CPU; i++) {
        memset((void *) local[j * MAX_BLOCKS_PER_CPU + i + 1], 0,
        sizeof(TREE));
        local[j * MAX_BLOCKS_PER_CPU + i + 1]->used = 0;
        local[j * MAX_BLOCKS_PER_CPU + i + 1]->parent = (TREE *) - 1;
        LockInit(local[j * MAX_BLOCKS_PER_CPU + i + 1]->lock);
    }
    ThreadWait(tid, 0);
}
```

ThreadInit()將資料結構初始化，把資料都設為 0。

```
ThreadWait (tree,waiting) {
        while(1){
                Lock(lock_smp);
                        smp_idle++;
                Unlock(tree->parent->lock);
                while (!thread[tid] && ! quit && (!waiting || waiting->nprocs))
                        Pause();                //等待工作
                if(quit)                        //當程式結束，跳出 ThreadWait()
                        return;
                value= SearchSMP(tree, alpha, beta, depth);
                Lock(lock_smp);
                Lock(thread[tid]->parent->lock);
                //合併資料，將資料傳給 parent
                CopyFromSMP(thread[tid]->parent, thread[tid], value);
                thread[tid]->parent->nprocs--;
                thread[tid]->parent->siblings[tid] = 0;
                Unlock(thread[tid]->parent->lock);
                thread[tid] = 0;
                Unlock(lock_smp);
        }
}
```

ThreadWait()讓沒有工作的 Process 在 ThreadWait()閒置，當程式結束的時候，就 return，最後 exit()。當有工作的時候，則進入 SearchSMP()，進行搜尋。

```
SearchSMP (tree, alpha, beta, depth){
    a=alpha;
    b=beta;

    while(1){
        Lock(tree->parent->lock);
        nextmove();   //選擇父親節點的走步
        Unlock(tree->parent->lock);
        DoMove(tree);
        value = -Search(tree, -b, -a, depth-1);
        if ((a < value) && (value < beta) && !firstmove){
            value = -Search(tree, -beta, -value,depth-1);  //呼叫 Search()
        }
        UndoMove(tree);
        if (value > a) {
            a = value;
        }
        if (value >= beta) {        //當發生 fail high，停止搜尋
            Lock(shared->lock_smp);
            Lock(tree->parent->lock);
            All thread Stop;
            Unlock(tree->parent->lock);
            Unlock(shared->lock_smp);
            break;
        }
        b=a+1;
    }
    return a;
}
```

SearchSMP()最主要的功能是讓那些一起搜尋的 Process 分別走它們父親節點的走步，並且搜尋底下子節點。