第三章　平行化的構想及作法

第一節 簡介

最近二、三十年來，關於平行化的搜尋演算法文章如雨後春筍般發表出來，這些演算法大致上可以分成兩大類[4]：

1、以 Alpha-Beta 演算法為基礎：Parallel Aspiration Search、Dynamic Tree Splitting 等。

2、以非 Alpha-Beta 演算法為基礎：Parallel Search based on SSS* (Campbell, 1981; Campbell and Marsland, 1983)、Parallel Search based on ER 等。

表 2 將以 Alpha-Beta 演算法為基礎的平行演算法列出來。其中 n 代表 Process 個數。CPU 分配方式有兩種：集中式表示 master CPU 集中在某些 Process；分散式則表示每顆 CPU 地位相當。

演算法	發表時間	CPU 分配	速度提升	效能提升
Parallel Aspiration Search[3]	1978	集中式	≦6	
Mandatory Work First[2]	1979	集中式	≦6	
Tree Splitting[12]	1980	集中式	2.34 (n=3) 5.12 (n=27)	0.78 (n=3) 0.18 (n=27)
PV-Split[21]	1981	集中式	3.75 (n=5)	0.75 (n=5)
Key Node[18]	1983	集中式	12.75 (n=20)	0.63 (n=20)
UIDPABS[22]	1986	集中式	3.94 (n=8)	0.49 (n=8)
DPVS[24]	1987	集中式	7.64 (n=19)	0.40 (n=19)
EPVS[15]	1987	集中式	5.93 (n=16)	0.37 (n=16)
Waycoo[10]	1987	分散式	101 (n=256)	0.39 (n=256)
Young Brothers Wait[9]	1987	分散式	142 (n=256) 344 (n=1024)	0.55 (n=256) 0.33 (n=1024)
Dynamic Tree Splitting[14]	1988	分散式	8.81 (n=16)	0.55 (n=16)
Bound-and-Branch[11]	1988	分散式	12 (n=32)	0.375 (n=32)
Delayed Branch Tree Expansion[13]	1990	集中式	350 (n=1000)	0.35 (n=1000)
Frontier Splitting[19]	1993	分散式	3.32 (n=16)	0.20 (n=16)

αβ*[8]	1993	分散式	6.5 (n=8)	0.81 (n=8)
CABP[7]	1994	集中式	4.6 (n=9)	0.51 (n=9)
Jamboree[17]	1994	分散式	50 (n=512)	0.09 (n=512)
ABDADA[25]	1995	分散式	15.85 (n=32)	0.49 (n=32)
Dynamic Multiple PV-Split[20]	1995	分散式	32 (n=64)	0.50 (n=64)
APHID[4]	1996	集中式	6.04 (n=16)	0.337 (n=16)

表 2 以 Alpah_Beta 演算法為基礎的平行演算法

由表 2 中可以看出 PV-Split、Key Node、Dynamic Tree Splitting 演算法效能提升比率($=\frac{速度提升}{CPU數量}$)比較好。

本論文所採用的就是 Dynamic Tree Splitting 的演算法實做。並且參考也是使用 DTS 演算法的平行西洋棋程式”Crafty”。

Crafty 曾經在 2004 年第 12 屆 World Computer Speed Chess Championship 比賽獲得第二名，成績相當的優異，而且開放原始碼，所以我們就參考 Crafty 來改良深象。

第二節 DTS 演算法作法介紹

DTS(Dynamic Tree Splitting) 是由 Robert M. Hyatt 所提出的平行化演算法[14]。這個演算法每個 Process 都是同等地位(Peer-to-Peer)，並不需要多一個 Process

來分配工作(Master-Slave)。

當 N 層搜尋一開始的時候，先由一個 Process 從根節點展開第一個走步往下搜尋，其它的 Process 都是閒置的等待工作。當搜尋到第 N 層的時候，當最左邊的分支的分數回傳回來，Process 就將工作分配給其它閒置的 Process。之後第 N-1 到第一層也是一樣，最左邊的分支分數傳回來之後，閒置的 Process 就開始幫忙搜尋。

當有閒置的 Process 的時候，它會去幫助其它正在運算的 Process，以加快搜尋的速度。

表 3 列出了使用 DTS 演算法搜尋效能提升的數據[14]。

CPU 個數	1	2	4	8	16
速度提升(以 1 顆 CPU 速度為單位)	1.0	2.0 倍	3.7 倍	6.6 倍	11.1 倍

表 3 使用 DTS 搜尋效能提升

第三節 Crafty 分析

這一節會簡單的描述 Crafty 是如何平行搜尋。圖 11 為整體程式流程圖[16]。

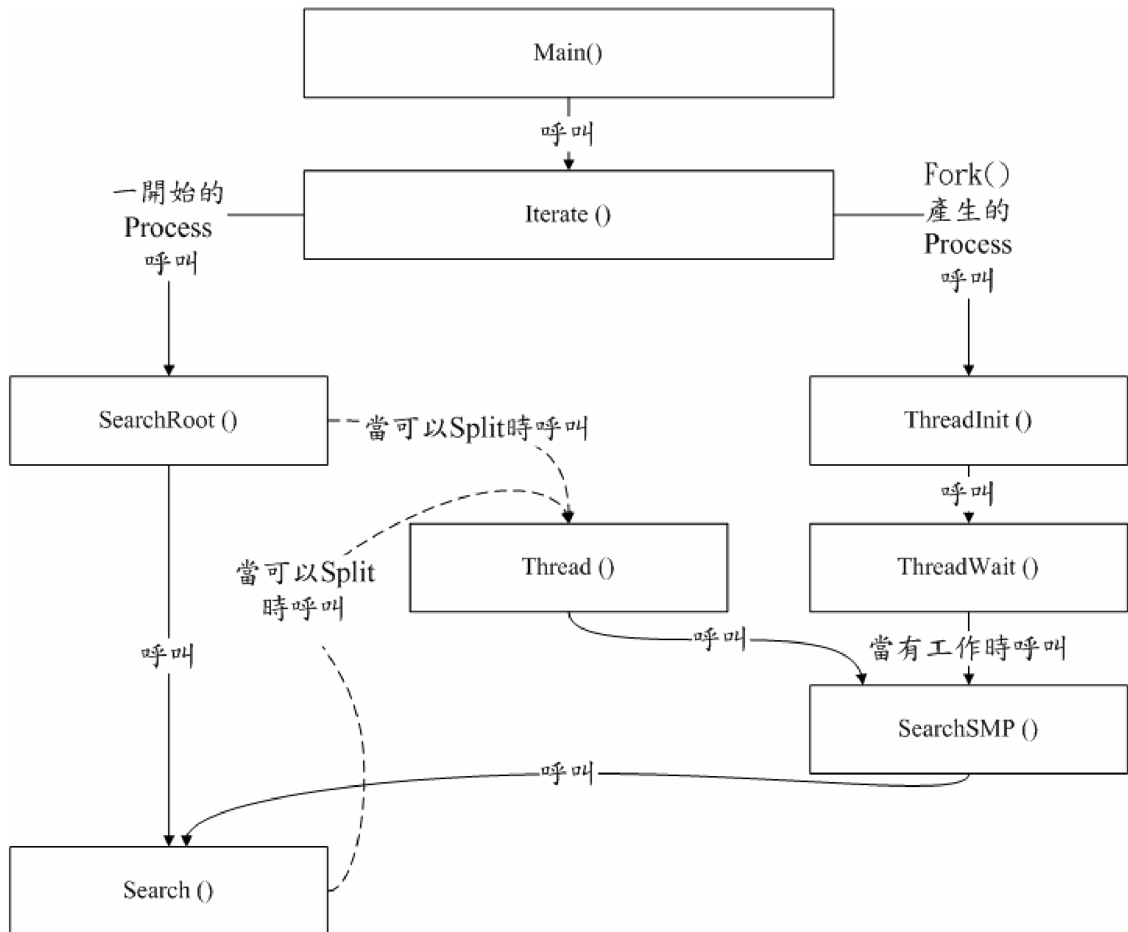

圖 11 流程圖

當 Process 進入 Iterate()會先判斷是否要產生其它 child Process，產生出來的 Process 會進入 ThreadInit()，初始化資料結構，之後進入 ThreadWait()，等待工作。

一開始的 Process 則是進入 SearchRoot()及 Search()。當發現有 Process 閒置的時候就進入 Thread()，將盤面資料複製給其它 Process，其它的 Process 發現有盤面資訊分配給它，則進入 SearchSMP()，取得 Parent Process 的走步，之後呼叫 Search()，搜尋底下子樹的變化。

以下說明各函式的功能：

Iterate()：這個函式在疊代加深之前，會將所需要的 Child Process 產生出來 (fork())，之後呼叫 SearchRoot()，進行搜尋。

SearchRoot()：這個函式最主要是搜尋根節點的走步，當第一個走步的結果回傳之後，可以將工作分配給其它閒置的 Process，進行平行搜尋。

Search()：使用 Nega_Max 演算法搜尋根節點以外的其它節點，並且使用同形表以及空步搜尋(Null-Move Search)來加快搜尋的速度。

Thread()：將工作分配給閒置的 Process(呼叫 CopyToSMP(A,B))，讓 Process 不再閒置。

SearchSMP()：一起搜尋同一個 Tree 的 Process，在這個函式中，取得父親節點的走步，並且呼叫 Search()，搜尋各個走步的變化。

ThreadInit()：初始化各個 Process 的資料結構。

ThreadWait() :閒置的 Process 在此函式中閒置，等待工作。當有資料複製給它的時候，Process 就進入 SearchSMP()搜尋。當父親節點的走步搜尋完畢，會將資料(例如：搜尋節點數量、最佳走步…等)合併傳給父親節點(呼叫 CopyFromSMP(A,B))。

CopyToSMP(A, B)：Process A 將盤面資訊複製給 Process B。

CopyFromSMP(A, B)：Process A 將搜尋資訊回傳給 Process B。

以下介紹各函式的虛擬碼。

```
Iterate (tree) {
	if (max_threads > smp_idle + 1)
		for (proc = smp_threads + 1; proc < max_threads; proc++)
			if( fork() = = 0){                    //產生 Process
				ThreadInit();
				exit(0);
			}
	for (; iteration_depth <= MAXPLY; iteration_depth ++){
		while (true) {
			value = SearchRoot (tree, alpha, beta, depth);
			if( value ≧ root_beta)              //當 fail high  調整 beta
				beta = beta + window;
			if(value ≦ root_alpha)              //當  fail low  調整  alpha
				alpha = alpha – window;
		}
	};
}
```

進入 Iterate(tree)會先判斷是否要產生 Process，當 max_threads>smp_idle，就產生 Processes，產生出來的 Process(我們之後稱它為 Child Process，以便與一開始的 Process 區分)，會進入 ThreadInit()，初始化資料，並且閒置等待工作。此處 max_threads 代表總共有幾個 Process 要運作。smp_idle 代表有幾個 Process 閒置。

```
SearchRoot (tree, alpha, beta, depth) {
    While(NextRootMove() ){          //從 Root 走步中取一個
        MakeMove();                  //走步
        if(firstmove){               //當搜尋第一步，以 full window 搜尋
            if(depth ≧ PLY)
                value = -Search(tree, -beta, -alpha, depth);
            else
                value = -Quiesce(tree, -beta, -alpha);
        }else{                       //使用 null window 搜尋
            if(depth ≧ PLY)
                value = -Search(tree, -alpha - 1, -alpha, depth);
        else
                value = -Quiesce(tree, -alpha - 1, -alpha);
            if ((value > alpha) && (value < beta))
                if (depth ≧ PLY)
                    value = -Search(tree, -beta, -alpha, depth);
                else
                    value = -Quiesce(tree, -beta, -alpha);
        }
        if (value > alpha) {
              if (value >= beta) {
                return (value);
              }
              alpha = value;
        }
        UnmakeMove();                //還原盤面
        if (shared->split_at_root && shared->smp_idle &&
    NextRootMoveParallel()   ){
            Thread();                //將工作分配出去，喚醒 process
        }
    }
    return alpha;
}
```

當 Process 產生完畢之後，一開始的 Process 就進入 SearchRoot()，開始進行

搜尋。

SearchRoot()使用 Nega_Max 演算法進行搜尋，當第一個走步搜尋完畢之後，判斷是否要在根節點 split、判斷有沒有 Process 閒置、以及是否還有走步，之後才呼叫 Thread()，將搜尋盤面資料複製給其它閒置的 Process，讓其它的 Process 幫忙搜尋。

```
Search (tree, alpha, beta, depth) {
    ProbeHash();                //查詢同形表
    Null_Move_Search();         //空步搜尋
    While(NextMove() ){         //選擇走步
        MakeMove();             //走步
        if( !moves_searched){   //當第一走步，以 full windw 搜尋
            if(depth ≧ PLY)
                value = -Search(tree, -beta, -alpha, depth);
            else
                value = -Quiesce(tree, -beta, -alpha);
        }else{                  //使用 null window 搜尋
            if(depth ≧ PLY)
                value = -Search(tree, -alpha - 1, -alpha, depth);
            else
                value = -Quiesce(tree, -alpha - 1, -alpha);
            if ((value > alpha) && (value < beta))
                if (depth ≧ PLY)
                    value = -Search(tree, -beta, -alpha, depth);
                else
                    value = -Quiesce(tree, -beta, -alpha);
        }
        if( value > alpha){
            if( value > beta){
                return value;
            }
            alpha = value;
        }
        UnmakeMove();                   //還原盤面
        if (smp_idle && !moves_searched &&
    min_thread_depth≦depth ){
            Thread();                   //將工作分配出去，喚醒 process
        }
    }
    return alpha;
}
```

Search()也是使用 Nega_Max 演算法進行搜尋，當第一個走步搜尋完畢之後，

判斷有沒有 Process 閒置、判斷目前搜尋深度是否大於 min_thread_depth，之後才呼叫 Thread()，將搜尋盤面資料複製給其它閒置的 Process，讓其它的 Process 幫忙搜尋。

```
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
                                                     //Process 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++)
        if (tree->siblings[proc])        //將工作分配給 proc，喚醒他們
            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。

```
SearchSMP (tree, alpha, beta, depth) {
	while(1){
		Lock(tree->parent->lock);
		move = nextmove(tree->parent,&tree->parent->sorter, a, beta,
depth, &tree->parent->extflag);		//取得父親節點的走步
		Unlock(tree->parent->lock);
		MakeMove();
		if(depth ≧ PLY)
			value = -Search(tree, -alpha - 1, -alpha, depth);
		else
			value = -Quiesce(tree, -alpha - 1, -alpha);
		if ((value > alpha) && (value < beta)){
			if (depth ≧ PLY)
				value = -Search(tree, -beta, -alpha, depth);
			else
				value = -Quiesce(tree, -beta, -alpha);
		}
		if (value > alpha) {
			alpha = value;
			if (ply == 1) {
				Lock(shared->lock_root);
				if (value > shared->root_value)
					shared->root_value = value;
				Unlock(shared->lock_root);
			}
			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;
			}
		}
		UnMakeMove();
	}
	return alpha;
}
```

SearchSMP()選擇一個父親節點走步之後，使用 Nega_Max 搜尋，當發現 fail high 的時候，通知其它在搜尋的 Process 停止搜尋。

```
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，把一些資料都預設為 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。