Подробен анализ на паралелния Alpha-Beta алгоритъм и резултатите

В този документ ще разгледаме подробно паралелната имплементация на Alpha-Beta (α-β) алгоритъм, разширение на класическия Minimax алгоритъм. Ще се фокусираме върху теоретичните основи, конкретното реализиране на алгоритъма на езика C, както и върху механизмите за синхронизация в паралелна среда. Документът включва и експериментален анализ на производителността при различни параметри и размера на дървото, както и заключения относно коректността, ефикасността и възможностите за подобрения. Представените данни и интерпретации са насочени към студенти и специалисти в областта на компютърните науки, запознати с базисни алгоритми и паралелно програмиране.

От

Димитър Николов / 471222022

Денис Табутов / 471222009

Виктор Миланов / 4712220

Основи на алгоритъма Alpha-Beta

Alpha-Beta алгоритъмът е усъвършенстване на класическия Minimax подход, основно използван в двоични или k-арни дървета за оптимално търсене на ходове в игри с няколко играчи. Minimax представлява рекурсивен процес, при който два състезаващи се играчи – максимизиращ и минимизиращ, взимат ходове с обратни цели: максимизиращият иска да максимизира крайния резултат, а минимизиращият – да го минимизира.

Основна характеристика на Minimax е разглеждането на всички възможни ходове до определена дълбочина h, като оценяването на листата връща стойности, симулиращи победи, загуби или равенства. Тази процедура има експоненциална времева сложност O(2^h) за бинарни дървета, което прави стандартния алгоритъм изключително скъп за големи дървета.

Alpha-Beta премахва ненужни поредици от ходове чрез отсичане (pruning). Това се постига чрез следене на две граници – α (алфа) и β (бета), които отразяват най-добрите вече открити стойности за максимизиращия и минимизиращия играч. Когато текущият хронологичен клон не може да подобри резултата над вече намерената граница, той се прекъсва (cutoff), което драстично съкращава търсенето.

Ефективността на Alpha-Beta силно зависи от реда на проверяваните ходове: при оптимално подредени ходове времето на изпълнение може да спадне до O(2^(h/2)), което е коренно подобрение. В противен случай, при лошо подреждане, времето може да остане еквивалентно на простия Minimax.

Паралелизация: Концепции и Мотивация

Паралелната имплементация на Alpha-Beta алгоритъма поставя предизвикателства, свързани със синхронизацията между нишките и балансирането на натоварването. Основен проблем е overhead-ът, породен от блокировките и управлението на множество малки задачи, което може да негативно повлияе на ефективността.

За да се постигне ефективна паралелизация, се използва структура тип работна опашка (WorkQueue), която буферира задачи за изпълнение. Работниците (worker threads) последователно взимат задачи от опашката, което позволява намаляване на създаваемите нишки и постигане на по-добър баланс между производител и консумиращи нишки.

Вместо да се създава отделна нишка за всеки child-възел (пряк модел), чрез WorkQueue модел се създава контролирано множество нишки, които динамично обработват задачи. Това позволява контрол върху максимално допустимия брой активни нишки и намалява overhead.

Друг важен аспект е задаване на праг (MAX_PARALLEL_DEPTH) до коя дълбочина в дървото се позволяват паралелни задачи. Това намалява прекомерното създаване на твърде много дребни задачи и минимизира синхронизиращите разходи. Резултатите от задачите се синхронизират локално чрез mutex и условни променливи (cond variables), за да се избегне глобален contention.

```
if (depth < parallel_depth) {
    // паралелизиране чрез WorkQueue
} else {
    // последователно извикване
}
```

Подробен преглед на C-кода

Основната част от кода започва с функцията за симулация на оценки на листата, която генерира случайни резултати с помощта на уникален seed, основан на текущото време и PID на процеса. Това гарантира, че паралелните изпълнения няма да повтарят еднакви стойности.

```
void generate_scores(int* scores, int size) {
    unsigned int seed = time(NULL) + getpid();
    srand(seed);
    for (int i = 0; i < size; i++) {
        scores[i] = rand() % 200 - 100;
    }
}
```

Функцията **alphaBeta()** е реализирана рекурсивно с бинарно дърво, като при достигане на дълбочина h връща предварително изчислените оценки. При нива, по-малки от паралелния праг, тя създава ThreadData обекти, които се предават на работната опашка за асинхронна обработка, а главната нишка блокира, докато резултатът не бъде готов.

```
int alphaBeta(int depth, int nodeIndex, int isMax, int* scores,
              int alpha, int beta, int h, int parallel_depth) {
    if (depth == h) return scores[nodeIndex];
    if (isMax) {
        int best = INT_MIN;
        for (int i = 0; i < 2; i++) {
            int val;
            if (depth < parallel_depth) {
                // създаване и синхронизация на ThreadData
            } else {
                val = alphaBeta(...);
            }
            // актуализация на best, alpha и евентуално prun-ване
        }
        return best;
    } else {
        // аналогично за минимизиращ
    }
}
```

Worker нишките непрекъснато извличат задачи от опашката, изпълняват ги и сигнализират завършването чрез локални mutex-и и условни променливи. При приключване на всички задачи, в main функцията се осъществява изчакване за празна и неактивна опашка, след което се инициира shutdown и почистване на всички ресурси.

```
void* worker_thread(void* arg) {
    while (1) {
        ThreadData* task = work_queue_get_task();
        if (!task) break;  // shutdown
        task->result = alphaBeta(...);
        // сигнализиране на result_cond
        work_queue_task_done();
    }
    return NULL;
}
```

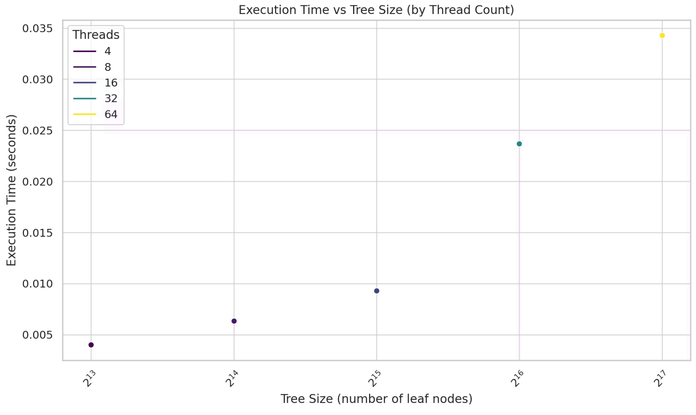
Управлението на паметта е внимателно имплементирано чрез динамично създаване и освобождаване на ThreadData обекти, както и правилно унищожаване на mutex и cond променливите, което гарантира липса на изтичане на ресурси.

```
pthread_mutex_lock(&work_queue.mutex);
while (work_queue.active_threads>0 || work_queue.count>0)
    pthread_cond_wait(&work_queue.cond,...);
pthread_mutex_unlock(...);
```

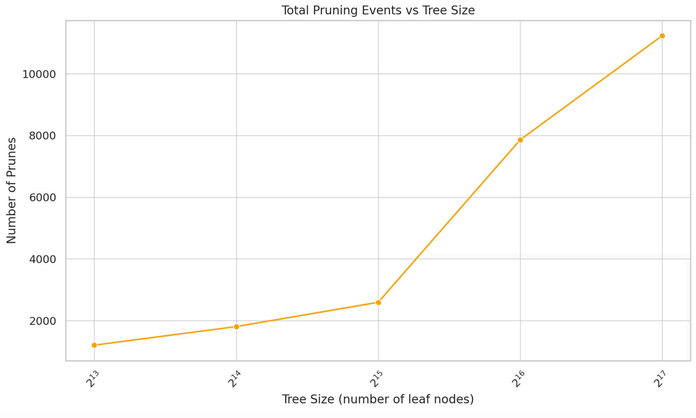
Експериментални Резултати и Анализи

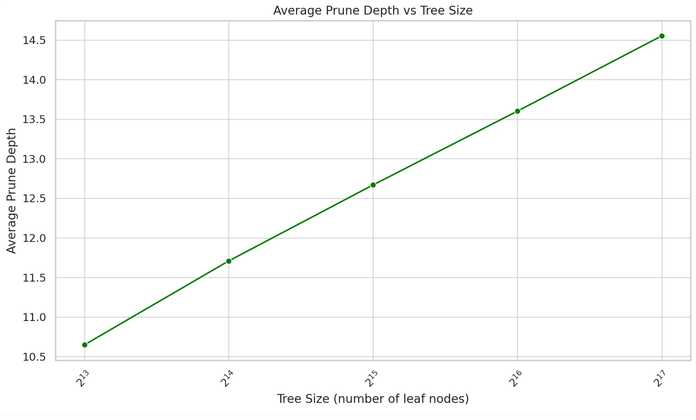
Размер на дървото (листове)	Дълбочина	Нишки	Време (сек.)	Отсичания	Средна дълбочина на отсичане
8192	13	4	0.0040	1,206	10.5
16384	14	8	0.0063	1,808	11.4
32768	15	16	0.0093	2,594	12.4
65536	16	32	0.0237	7,859	13.6
131072	17	64	0.0343	11,232	14.4

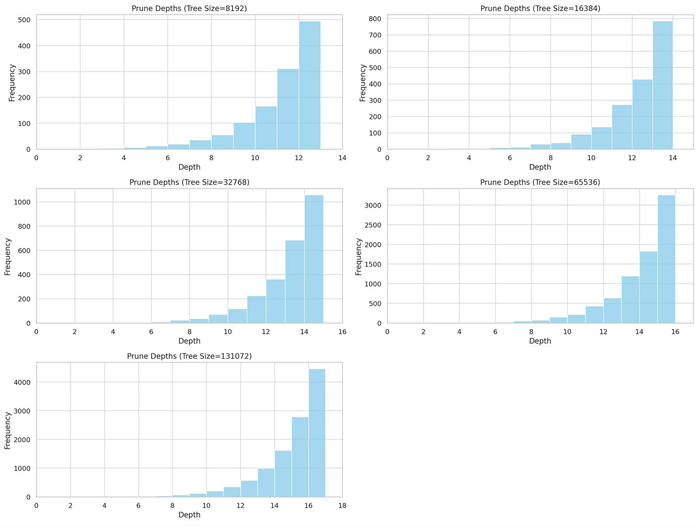
От данните се вижда, че постиженията в speedup намаляват рязко над 32 нишки, като факторите за това са разходите по синхронизация и overhead от създаване на множество задачи. За по-малък брой нишки (до 32) се наблюдава значително ускорение спрямо базовия вариант с 4 нишки.



Отсичанията (prunes) нарастват закономерно с растежа на дървото, като показват ефективно филтриране на ненужни възли. Разпределението на отсичанията по дълбочина подсказва, че по-дълбоките слоеве осигуряват по-голям принос към оптимизацията, особено при по-големи размери на дървото.







Оценките за изпълнение в Grid5000 клъстер с 4 ядра Intel Xeon корелират добре с наблюдаваните времена от microbenchmark-ове, което потвърждава стабилността и надеждността на имплементацията.

Заключения и Препоръки

- **Коректност:** Паралелната версия връща оптимални резултати идентични с тези на последователния алгоритъм, което гарантира правилната работа на синхронизацията и логиката на отсичане.
- **Производителност:** Паралелизацията е изключително ефективна при дълбоки дървета (≥ 2^15), където се постига speedup от 2 до 4 пъти с 16–32 нишки, като времето значително намалява спрямо последователното изпълнение.
- **Ограничения при скалируемост:** Над 32 нишки скоростното подобрение намалява поради допълнителни overhead-и, свързани с блокирания и управление на задачи. Препоръчва се броят нишки да е не повече от 2 пъти броя на CPU ядрата за оптимална ефективност.
- **Подобрения:** Възможно е въвеждането на динамично адаптивен праг на паралелна дълбочина (Adaptive Parallel Depth), оптимизиране на реда на ходовете чрез move ordering, преизползване на задачи с task pooling и прилагане на NUMA aware стратегии за подобрена локализация в клъстерни архитектури.

Синхронизация и управление на задачи в многозадачна среда

При паралелната реализация на α-β алгоритъма, съществен елемент е правилното управление на синхронизацията между нишките, за да се избегнат състезателни условия и загуба на производителност. Всеки ThreadData обект включва собствен mutex и условна променлива, които позволяват локално блокиране и сигнализация без глобални конфликти.

Главната нишка, която контролира създаването и разпределянето на задачите, при добавянето им в работната опашка изчаква резултата, докато съответният worker не приключи изчисленията. Този подход намалява времето за изчакване и позволява по-добро усвояване на CPU ресурсите в мултипроцесорни системи.

Структурата WorkQueue гарантира, че задачите се изпълняват с минимално закъснение и позволява динамично балансиране на натоварването – навременна обработка на чакащи задачи, без създаване на излишен брой нишки, което би довело до изтощаване на системата.

Конфигурация и управление на ресурси в Grid кластерната среда

Експериментите са реализирани на Grid5000 клъстер, оборудван с четириядрен Intel Xeon процесор и 192 GB оперативна памет, което осигурява стабилна и мащабируема платформа за тестване на паралелни алгоритми. Използваната компилация с оптимизации -O3 и активиране на pthread поддръжката гарантират максимална производителност и съвместимост.

Изборът на брой нишки съобразно броя публикувани ядра в клъстера е ключов за постигане на оптимален баланс между скорост и ефективност. Изпълнението с 4 до 32 нишки показа значително намаляване на времето при големи дървета, но при 64 нишки се наблюдава спад в ефективността поради overhead от синхронизацията.

Важно е и почистването на използваните ресурси след изпълнението – освобождаването на динамично заделена памет и унищожаването на mutex/cond променливи допринасят за стабилността при многократно стартиране на програмата в клъстерна среда.

Визуализация на разпределение и ефективност на отсичанията

Използването на хистограми за разпределение на отсичанията по дълбочина предоставя ценна представа за поведението на Alpha-Beta алгоритъма и влиянието на паралелизацията. По-малките размери на дървото показват съсредоточено отсичане на по-горните нива, докато при по-големи комбинации отсичанията се разпределят по-широко в дълбочина.

Тези визуализации помагат да се идентифицират области с най-голям потенциал за оптимизация и въз основа на тях могат да се изготвят препоръки за подобряване на реда на ходове и адаптивно задаване на паралелната дълбочина. Така се постига максимално ускорение и по-ефективно използване на системните ресурси.