LECTURE 17
Speculative Parallelism and Computer Chess

Charles E. Leiserson

November 15, 2011
• Speculative Parallelism
• Alpha–Beta Search
• Computer–Chess Programs
OUTLINE

• Speculative Parallelism
• Alpha–Beta Search
• Computer–Chess Programs
Computing a Boolean Product

\[ p = \bigwedge_{i=0}^{n-1} A_i \]

```c
int big_and(int *A, int n) {
    int p = TRUE;
    for (int i = 0; i < n; ++i) {
        p = p && A[i];
    }
    return p;
}
```
**Boolean Product in Parallel**

\[ p = \bigwedge_{i=0}^{n-1} A_i \]

```cpp
int big_and(int *A, int n) {
    int p = TRUE;
    cilk::reducer_ptr<and_monoid_int> ptr(&p);
    cilk_for (int i=0; i<n; ++i) {
        *ptr = *ptr && A[i];
    }
    return p;
}
```
Optimization: Quit early if the partial product ever becomes FALSE.

```c
int big_and(int *A, int n) {
  int p = TRUE;
  for (int i=0; i<n; ++i) {
    p = p && A[i];
    if (p == FALSE) break;
  }
  return p;
}
```

Question: When is it worthwhile to parallelize such a loop?
Definition. *Speculative parallelism* occurs when a program spawns some parallel work that might not be performed in a serial execution.

**Rule of Thumb:** Don’t spawn speculative work unless there is little other opportunity for parallelism and there is a good chance it will be needed.
Theorem. Suppose that a program contains two parts A and B, and that after A executes, the probability is $\alpha$ that we need to execute B. Assuming a greedy scheduler, it cannot be worthwhile to speculate on B if the parallelism of B exceeds $\alpha P/(1 - \alpha)$, where P is the number of processors.

Proof. Let $T_P = (A_1 + B_1)/P + \max\{A_\infty, B_\infty\}$ be the time for the speculative execution, and let $T'_P = (A_1 + \alpha B_1)/P + A_\infty + \alpha B_\infty$ be the expected time for executing A and B (if necessary) in series. Then we have

$$T_P = (A_1 + B_1)/P + \max\{A_\infty, B_\infty\}$$
$$= (A_1 + \alpha B_1)/P + (1 - \alpha)B_1/P + A_\infty + B_\infty - \min\{A_\infty, B_\infty\}$$
$$= T'_P + (1 - \alpha)B_1/P + (1 - \alpha)B_\infty - \min\{A_\infty, B_\infty\}$$
$$= T'_P + (1 - \alpha)B_1/P + B_\infty - \alpha B_\infty - \min\{A_\infty, B_\infty\}$$
$$\geq T'_P + (1 - \alpha)B_1/P - \alpha B_\infty$$
$$> T'_P$$

if $(1 - \alpha)B_1/P > \alpha B_\infty$, or equivalently, if $B_1/B_\infty > \alpha P/(1 - \alpha)$. \qed
int big_and(int *A, int n) {
    int p = TRUE;
    cilk::reducer_ptr<and_monoid_int> ptr(&p);
    cilk_for (int i=0; i<n; ++i) {
        *ptr = *ptr && A[i];
    }
    return p;
}

How do we quit early if the partial product becomes FALSE?
Divide-and-Conquer Loop

```c
int big_and(int *A, int n) {
    if (n > THRESHOLD) {
        int p1 = cilk_spawn big_and(A, n/2);
        int p2 = big_and(A+n/2, n-n/2);
        cilk_sync;
        return p1 && p2;
    }
    int p = 1;
    for (int i=0; i<n; ++i) {
        p = p && A[i];
    }
    return p;
}
```
Short Circuit on FALSE

```c
static int is_false = FALSE;
int big_and(int *A, int n) {
    if (is_false) return FALSE;
    if (n > THRESHOLD) {
        int p1 = cilk_spawn big_and(A, n/2);
        int p2 = big_and(A + n/2, n-n/2);
        cilk_sync;
        return p1 && p2;
    }
    int p = 1;
    for (int i=0; i<n; ++i) {
        p = p && A[i];
    }
    if (p == FALSE) is_false = TRUE;
    return p;
}
```

**Notes:**
- Beware: nondeterministic code!
- Contains a benign race.
- Don’t forget to reset `is_false` after use!
- Is a memory fence necessary? **No!**
class Abort {
public:
    Abort();
    explicit Abort(Abort* p);
    int isAborted() const;
    void abort();
    void setPollingGranularity(int i);
};

IDEA: Poll up the cactus stack to see whether any internal node desires an abort.
• Speculative Parallelism
• Alpha–Beta Search
• Computer–Chess Programs
Two players: MAX □ and MIN ●.

The game tree represents all moves from the current position within a given search **ply** (depth).

At leaves, apply a static evaluation function.

MAX chooses the maximum score among its children.

MIN chooses the minimum score among its children.
**IDEA:** If MAX discovers a move so good that MIN would never allow that position, MAX’s other children need not be searched — *beta cutoff.*
IDEA: If MAX discovers a move so good that MIN would never allow that position, MAX’s other children need not be searched — *beta cutoff*.
**IDEA:** If MAX discovers a move so good that MIN would never allow that position, MAX’s other children need not be searched — *beta cutoff.*
**Idea:** If MAX discovers a move so good that MIN would never allow that position, MAX’s other children need not be searched — *beta cutoff.*
IDEA: If MAX discovers a move so good that MIN would never allow that position, MAX’s other children need not be searched — *beta cutoff*. 
**IDEA:** If MAX discovers a move so good that MIN would never allow that position, MAX’s other children need not be searched — *beta cutoff.*
**IDEA:** If MAX discovers a move so good that MIN would never allow that position, MAX’s other children need not be searched — *beta cutoff.*
**IDEA:** If MAX discovers a move so good that MIN would never allow that position, MAX’s other children need not be searched — *beta cutoff.*
**IDEA:** If MAX discovers a move so good that MIN would never allow that position, MAX’s other children need not be searched — *beta cutoff.*
**IDEA:** If MAX discovers a move so good that MIN would never allow that position, MAX’s other children need not be searched — *beta cutoff.*
**IDEA:** If MAX discovers a move so good that MIN would never allow that position, MAX’s other children need not be searched — *beta cutoff.*
**IDEA:** If MAX discovers a move so good that MIN would never allow that position, MAX’s other children need not be searched — *beta cutoff.*
IDEA: If MAX discovers a move so good that MIN would never allow that position, MAX’s other children need not be searched — beta cutoff.
**IDEA:** If MAX discovers a move so good that MIN would never allow that position, MAX’s other children need not be searched — *beta cutoff.*
**IDEA:** If MAX discovers a move so good that MIN would never allow that position, MAX’s other children need not be searched — *beta cutoff.*
**IDEA:** If MAX discovers a move so good that MIN would never allow that position, MAX’s other children need not be searched — *beta cutoff.*
**IDEA:** If MAX discovers a move so good that MIN would never allow that position, MAX’s other children need not be searched — *beta cutoff.*
IDEA: If MAX discovers a move so good that MIN would never allow that position, MAX’s other children need not be searched — *beta cutoff*. 
IDEA: If MAX discovers a move so good that MIN would never allow that position, MAX’s other children need not be searched — *beta cutoff*. 
IDEA: If MAX discovers a move so good that MIN would never allow that position, MAX’s other children need not be searched — *beta cutoff*. 
**IDEA:** If MAX discovers a move so good that MIN would never allow that position, MAX’s other children need not be searched — *beta cutoff.*
IDEA: If MAX discovers a move so good that MIN would never allow that position, MAX’s other children need not be searched — beta cutoff.
Alpha–Beta Strategy

- Each search from a node employs a window $[\alpha, \beta]$.
- If the value of the search falls between $\alpha$ and $\beta$, then the value of $\alpha$ is increased.
- If the value of the search falls above $\beta$, generate a beta cutoff.
- If the value of the search falls below $\alpha$, keep searching.
```c
int search( position *prev, int move, int depth ) {
    position cur;           /* Current position       */
    int best_score = -INF;   /* Best score so far      */
    int num_moves;           /* Number of children     */
    int child_sc;            /* Child's score          */

    make_move(prev, move, &cur);

    int sc = eval(&cur);     /* Static evaluation */
    if ( abs(sc) >= MATE || depth <= 0 ) { /* Leaf node */
        return (sc);
    }

    cur.alpha = -prev->beta;  /* Negamax */
    cur.beta = -prev->alpha;

    return best_score;
}
```
// Generate moves, hopefully in best-first order
num_moves = gen_moves(&cur);

for ( int mv = 0; mv < num_moves; ++mv ) {
    child_sc = -search( &cur, mv, depth-1 );
    if ( child_sc > best_score )
        best_score = child_sc;
    if ( child_sc >= cur.beta ) /* beta cutoff */
        break;
    if ( child_sc >= cur.alpha )
        cur.alpha = child_sc;
}
return best_score;
Theorem [KM75]. For a game tree with branching factor $b$ and depth $d$, an alpha–beta search with moves searched in *best-first order* examines exactly $b^{\lceil d/2 \rceil} + b^{\lfloor d/2 \rfloor} - 1$ nodes at ply $d$. □

The naive algorithm examines $b^d$ nodes at ply $d$. For the same work, the search depth is effectively doubled. For the same depth, the work is square rooted.
Opening Book

- Precompute best moves at the beginning of the game.
- The [KM75] theorem implies that it is cheaper to keep separate opening books for each side than to keep one opening book for both.
Iterative Deepening

- Rather than searching the game tree to a given depth $d$, search it successively to depths $1, 2, 3, \ldots, d$.
- With each search, the work grows exponentially, and thus the total work is only a constant factor more than searching depth $d$ alone.
- During the search for depth $k$, keep move-ordering information to improve the effectiveness of alpha-beta during search $k+1$.
- Good mechanism for time control.
**Observation:** In a best-ordered tree, the degree of every node is either 1 or maximal.

**IDEA** [FMM91]: If the first child fails to generate a beta cutoff, speculate that the remaining children can be searched in parallel without wasting work: “Young Brothers Wait.” Abort subcomputations that prove to be unnecessary.
• Speculative Parallelism
• Alpha–Beta Search
• Computer–Chess Programs
Quiescence Search

- Evaluating at a fixed depth can leave a board position in the middle of a capture exchange.
- At a “leaf” node, continue the search using only captures — **quiet** the position.
Null–Move Pruning

- In most positions, there is always something better to do than nothing.
- Forfeit the current player’s move (illegal in chess), and search to a shallower depth.
- If a beta cutoff is generated, assume that a full-depth search would have also generated the cutoff.
- Otherwise, perform a full-depth search of the moves.
- Watch out for zugzwang!
Other Search Heuristics

- Killers
  - The same move at a given depth tends to generate cutoffs.
- Move extensions — grant an extra ply to the search if
  - the King is in check,
  - certain captures,
  - singular (forced) moves.
- Zero-window search — a variant of alpha–beta, where $\text{alpha} = \text{beta}$. 
Transposition Table

• The search tree is actually a dag!
• If you’ve searched a position to a given depth before, memoize it in a hash table (actually a cache), and don’t search it again.
• Store the best move from the position to improve alpha–beta and minimize wasted work in parallel alpha–beta.
• Tradeoff between how much information to keep per entry and the number of entries.
• For each square on the board and each different state of a square, generate a random string.
• The hash of a board position is the XOR of the random strings corresponding to the states of the squares.
• Because XOR is its own inverse, the hash of the position after a move can be accomplished incrementally by a few XOR’s, rather than by computing the entire hash function from scratch.
Board Representation

• Bitboards
  ▪ Use a 64–bit word to represent, for example, where all the pawns are on the 64 squares of the board.
  ▪ Use POPCOUNT and other bit tricks to do move generation and to implement other chess concepts.
Performance-engineer a computer program to play **Khet 2.0** (like chess, but with lasers).

4-person teams, each with a dedicated 12-core computer.

Each team will receive a free game, courtesy of Dr. Michael Larson, CEO of Innovention Toys.

Games will be available as teams are formed.
Computer–Human Khet Match

Don Dailey + Core i7

VS.

Dr. Michael Larson

Today at 4:00 P.M. in Gates 7th-floor lounge