LECTURE 18
Speculative Parallelism & Computer Chess

Charles E. Leiserson
November 6, 2014
SPECULATIVE PARALLELISM
#define uint unsigned int

bool sum_exceeds(uint *A, size_t n, uint limit) {
    uint sum = 0;
    for (size_t i=0; i<n; ++i) {
        sum += A[i];
    }
    return sum > limit;
}
**Short-Circuiting**

**Optimization (Bentley rule)**
Quit early if the partial product ever exceeds the threshold.

```c
#define uint unsigned int

bool sum_exceeds(uint *A, size_t n, uint limit) {
    uint sum = 0;
    for (size_t i=0; i<n; ++i) {
        sum += A[i];
        if (sum > limit) return true;
    }
    return false;
}
```
#define uint unsigned int

bool sum_exceeds(uint *A, size_t n, uint limit) {
  uint sum;
  CILK_C_REDUCER_OPADD(sum, uint, 0);
  CILK_C_REGISTER_REDUCER(sum);
  cilk_for (size_t i=0; i<n; ++i) {
    REDUCER_VIEW(sum) += A[i];
  }
  CILK_C_UNREGISTER_REDUCER(sum);
  return REDUCER_VIEW(sum) > limit;
}

**Question**

How can we parallelize a short-circuited loop?
#define uint unsigned int

uint sum_of(uint *A, size_t n) {
    if (n > 1) {
        uint s1 = cilk_spawn sum_of(A, n/2);
        uint s2 = sum_of(A + n/2, n - n/2);
        cilk_sync;
        uint sum = s1 + s2;
        return sum;
    }
    return A[0];
}

bool sum_exceeds(uint *A, size_t n, uint limit) {
    return sum_of(A, n) > limit;
}

How might we quit early and save work if the partial sum exceeds the threshold?
#define uint unsigned int

uint sum_of(uint *A, size_t n, uint limit, bool *abort_flag) {
  if (*abort_flag) return 0;
  if (n > 1) {
    uint s1 = cilk_spawn sum_of(A, n/2, limit, abort_flag);
    uint s2 = sum_of(A + n/2, n - n/2, limit, abort_flag);
    cilk_sync;
    uint sum = s1 + s2;
    if (sum > limit && !*abort_flag) *abort_flag = true;
    return sum;
  }
  return A[0];
}

bool sum_exceeds(uint *A, size_t n, uint limit) {
  bool abort_flag = false;
  return sum_of(A, n, limit, &abort_flag) > limit;
}
#define uint unsigned int

uint sum_of(uint *A, size_t n, uint limit, bool *abort_flag) {
  if (*abort_flag) return 0;
  if (n > 1) {
    uint s1 = cilk_spawn sum_of(A, n/2, limit, abort_flag);
    uint s2 = sum_of(A + n/2, n - n/2, limit, abort_flag);
    cilk_sync;
    uint sum = s1 + s2;
    if (sum > limit && !*abort_flag)
      *abort_flag = true;
  }
  return sum;
}

return A[0];

bool sum_exceeds(uint *A, size_t n, uint limit) {
  bool abort_flag = false;
  return sum_of(A, n, limit, &abort_flag);
}

Notes:
- Beware: nondeterministic code!
- The benign race on abort_flag can cause true-sharing contention if you are not careful.
- Don’t forget to reset abort_flag after use!
- Is a memory fence necessary? No!

Short-Circuiting in Parallel
Speculative Parallelism

**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 worst-case greedy scheduler, it cannot be worthwhile to speculate on B if the parallel slackness of B exceeds $\alpha/(1 - \alpha)$.

Proof. Let $P$ be the number of processors, 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 $B_1/\alpha B_\infty > \alpha/(1 - \alpha)$. ■
ALPHA–BETA SEARCH
Min–Max Search

- 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 \textbf{MAX} discovers a move so good that \textbf{MIN} would never allow that position, \textbf{MAX}'s other children need not be searched — \textit{beta cutoff}.
**IDEA:** If $\text{MAX}$ discovers a move so good that $\text{MIN}$ would never allow that position, $\text{MAX}$’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 Pruning

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 \textbf{MAX} discovers a move so good that \textbf{MIN} would never allow that position, \textbf{MAX}'s other children need not be searched — \textit{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 Pruning

**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 \([\text{alpha}, \text{beta}]\).
- If the value of the search falls below \text{alpha}, keep searching.
- If the value of the search falls between \text{alpha} and \text{beta}, then increase \text{alpha} and keep searching.
- If the value of the search falls above \text{beta}, generate a beta cutoff and return.
```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;
};
```
// 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.
**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 Siblings Wait.” Abort subcomputations that prove to be unnecessary.
Abort Library

```c
void abort_constructor(Abort *self, Abort *parent);
int is_aborted(Abort *self);
void do_abort(Abort *self);
```

**IDEA:** Poll up the search tree to see whether any internal node desires an abort.
COMPUTER–CHESS PROGRAMS
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, ..., \( d \).
- 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.
Endgame Database

**IDEA:** If there are few enough pieces on the board, precompute the outcomes and store them in a database.

- It doesn’t suffice to store just win, loss, or draw for a position.
- Keep the distance to mate to avoid cycling.
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.
- Each side has the option of “standing pat.”
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 good move at a given depth tends to generate cutoffs elsewhere in the tree.
- 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.
Zobrist Hashing

- 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.
Transposition–Table Records

- Zobrist key
- Score
- Move
- Quality (depth searched)
- Bound type (upper, lower, or exact)
- Age
Typical Move Ordering

1. Transposition-table move
2. Internal iterative deepening
3. Nonlosing capture in MVV–LVA (most valuable victim, least valuable aggressor) order
4. Killers
5. Losing captures
6. History heuristic
Late-Move Reductions (LMR)

**Observation**
With a good move ordering, a beta cutoff will either occur right away or not at all.

**Strategy**
- Search first few moves normally.
- Reduce depth for later moves.
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.
More Good Stuff

http://chessprogramming.wikispaces.com/
Final–Project Preview

- Performance–engineer a computer program to play Leiserchess–2014 (like chess, but with lasers).
- 2 betas plus a final.
- Form your teams!
- **Major time saver:** Code walk of the Leiserchess codebase next week on Thursday.
- Bring your laptops!