Lecture 20
Speculative Parallelism & Leisерchess

Charles E. Leiserson
Speculative Parallelism
Thresholding a Sum

```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];
    }
    return sum > limit;
}
```
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_REDUCTOR_OPADD(sum, uint, 0);
    CILK_C_REGISTER_REDUCTOR(sum);
    cilk_for (size_t i=0; i<n; ++i) {
        REDUCER_VIEW(sum) += A[i];
    }
    CILK_C_UNREGISTER_REDUCTOR(sum);
    return REDUCER_VIEW(sum) > 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)
            return sum;
    }
    return A[0];
}

bool sum_exceeds(uint *A, size_t n, bool abort_flag) {
    return sum_of(A, n, 0, 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!
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.
PARALLEL ALPHA–BETA SEARCH
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 $b^{\lfloor d/2 \rfloor} + b^{\lfloor d/2 \rfloor} - 1$ nodes at ply $d$. The naive algorithm examines $bd$ nodes at ply $d$. For the same work, the search depth is effectively doubled. For the same depth, the work is square-rooted.
**Parallel Alpha–Beta**

**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 Mechanism

typedef struct searchNode {
    struct searchNode *parent;
    position_t position;
    bool abort_flag;
} searchNode;

**IDEA:** Poll up the search tree to see whether any internal node desires an abort.
Problem: In general, the game tree is not best-ordered, meaning that parallel alpha-beta search using the “young siblings wait” idea will waste work.
Alpha–Beta Search: Example
Alpha–Beta Search: Example
Alpha–Beta Search: Example
Alpha–Beta Search: Example
Alpha–Beta Search: Example
Alpha–Beta Search: Example
Alpha–Beta Search: Example

Second sibling provides cutoff.
Parallel recursive full-window searches.
Young Siblings Wait: Example

$S \geq 3$

$S = 3$

$S = 3$

$S \geq 7$

$S \geq 6$

2 3 0 7 0 8 6 1 0 6 0 0 7 6 0 0 7 5 0 0 4 0 0 2 0 0 2 0 0

© 2012–2018 by the Lecturers of MIT 6.172
Young Siblings Wait: Example

Parallel recursive full-window searches.
Parallel recursive full-window searches.

Cutoff from second child is not available to prune these searches.
IDEA: Allow children to update parent’s alpha/beta value concurrently.

- Children can poll for the alpha/beta value.
- Problem: Difficult to implement efficiently.
- Problem: Efficiency relies on lucky scheduling!
In practice, speculative alpha–beta search of a game tree will always waste some work.

Aim to balance two conflicting goals:
- Generate enough parallel work to get parallel speedup.
- Don’t do too much unnecessary work.
JAMBOREEE SEARCH
**IDEA** [K94]: After searching the first child, perform a **scout search** of the remaining children in parallel, and sequentially value any tests that fail.

- In other words, do `searchPV` serially, and do `scout-search` in parallel.

**Intuition**: It’s fine to waste work on a zero–window search, but not on a full–window search.
Recursive zero-window search for $S \geq 3$.  

© 2012–2018 by the Lecturers of MIT 6.172
Jamboree Search: Example
Jamboree Search: Example

Recursive zero-window search for $S \geq 3$. 
Jamboree Search: Example

Recursive zero-window search for $S \geq 3$. 

Recursive zero-window search for $S \geq 3$. 

© 2012–2018 by the Lecturers of MIT 6.172
Jamboree Search: Example

Test failed. Wait for preceding children to finish, then recursively value this tree.

© 2012–2018 by the Lecturers of MIT 6.172
Jamboree Search: Example

© 2012–2018 by the Lecturers of MIT 6.172
Recursive zero-window search for $S \geq 6$. 

© 2012–2018 by the Lecturers of MIT 6.172
Jamboree Search: Example

Recursive zero-window search for \( S \geq 6 \).

Test failed. Wait for preceding children to finish, then recursively value this tree.
Recursive full-width search.
JAMBOREE(n,α,β)
1 if n is a leaf then return STATICEVAL(n)
2 {c₀,c₁,...,cₖ} = Children(n)
   b = -JAMBOREE(c₀, -β, -α)
   if b ≥ β then return b
   if b > α then α = b
   parallel_for (cᵢ in {c₁,c₂,...,cₖ})
   7 s = -JAMBOREE(cᵢ, -α-1, -α)
   8 if s > b then b = s
   9 if s ≥ β then abort-and-return s
   10 if s > α then
   11 wait for completion of all cⱼ where j < i
   12 s = -JAMBOREE(cᵢ, -β, -α)
   13 if s ≥ β then abort-and-return s
   14 if s > α then α = s
   15 if s > b then b = s
return b
The Leiserchess codebase is already structured to support a simple parallelization of `scout_search`.

```c
static score_t scout_search(searchNode* node, int depth,
      uint64_t* node_count_serial) {
    ...
    cilk_for (int mv_index = 0; mv_index < num_of_moves;
               mv_index++) {
      // Get the next move from the move list.
      int local_index = number_of_moves_evaluated++;
      move_t mv = get_move(move_list[local_index]);
      ...
    }
    ...
}
```

Resulting search is not the same as Jamboree search, but it’s enough to get you started.
Tips for Parallelizing Leiserchess

- Simply parallelizing the loop will produce code with races! Consider how you can address them:
  - Synchronize concurrent accesses, e.g., using locks.
  - Make a thread-local copy when a computation is stolen.
  - Use a thread-local data structure, but don’t copy data between threads.
  - Decide the race is benign and leave it be.
- Avoid generating too much wasted work.
  - Duplicate the loop over the moves in scout_search, and make one copy parallel.
  - Switch from the serial loop to the parallel loop when the number of legal moves is high enough.
• 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.
**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.
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
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.
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.
https://www.chessprogramming.org/