Speculative Parallelism in Cilk++

Ruben Perez & Gregory Malecha

MIT

May 11, 2010
Lots of search algorithms are *embarrassingly parallel*.

- Search down all the paths of a tree.
- Search multiple disjoint branches in parallel.
  - No communication overhead.
  - Very little memory contention.
- Since we often only need a single answer, we can stop once we’ve found any solution.

Cannonical algorithms are in game-search, minimax and $\alpha-\beta$ pruning.
2 Player Game Trees

- Nodes represent game states.
- Edges represent transitions between states.
- Leaf states are scored by a heuristic.
- Goal is to maximize the heuristic against an adversary.
Parallelizing MiniMax
Parallelizing MiniMax
Parallelizing MiniMax

```
max

3 0

min

max
```
Parallelizing MiniMax
Parallelizing MiniMax

```
max

3  0

min

2  -1

max
```
Parallelizing MiniMax

```
3
\rightarrow max
|   |
|\downarrow|   |
0 \rightarrow max
|   |
|\downarrow|   |
-1 \rightarrow max
|   |
|\downarrow|   |
2 \rightarrow min
|   |
|\downarrow|   |
-1
```
Parallelizing MiniMax
Improving MiniMax

- MiniMax exhaustively searches the game tree to a certain depth (iterative deepening).
- Not efficient for most games due to large branching factor.
  - Can’t search deep enough for the computer to be smart.
- **Intuition** Keep track of the range of feasible scores and prune branches that fall outside the range.
Keep additional information with each game node: $\alpha$ and $\beta$.

- $\alpha$ is the lower bound on the player’s score.
- $\beta$ is the upper bound on the player’s score.

This pruning allows us to search twice as deep on average.
Keep additional information with each game node: $\alpha$ and $\beta$.

- $\alpha$ is the lower bound on the player’s score.
- $\beta$ is the upper bound on the player’s score.
- This pruning allows us to search twice as deep on average.

Let’s see an example...
Pruning Example

$[-\infty, \infty]$
Pruning Example

[3, ∞]

max

3

min

max
Pruning Example

[3, ∞]

max

3
0

min

max

Ruben Perez & Gregory Malecha (MIT)
Pruning Example

[3, ∞]

max

min

max

3 0 [3, ∞]
Pruning Example

\[ \begin{array}{c}
\text{max} \\
3 \\
\text{min} \\
0 \\
\text{max} \\
2 \\
\end{array} \]

\[ [3, \infty] \]
Pruning Example

```
[3, ∞]
```

```
max
```

```
3 0 [3, 2]
```

```
min
```

```
2 Prune!
```

```
max
```
Outline

1. A Recipe for Speculation
   - Porting the Example
   - Implementing abort

2. Evaluation
   - Performance
   - Complexity Porting Old Code

3. Conclusions
When we speculate, we don’t want to have to commit to something finishing.

- Need a way to abort currently running computations that we don’t need anymore.
- For consistency, we’d like to be able to abort computations that are speculating.

We also need a way to combine the results.

- Reducers would work for this, but they will need to be able to call abort.
- Older versions of cilk had another mechanism for this.
Speculation is just based on spawn.

Combining results done through inlets.

- inlets are functions that merge results together (similar to reducers).
- inlets can also make the choice to abort computations.

Let’s look at a simple example.
Simple Example: Native `abort` & Inlets

Imagine you have two computations that should yield the same result, but one could take significantly longer to compute.
Simple Example: Native abort & Inlets

- Imagine you have two computations that should yield the same result, but one could take significantly longer to compute.
  - Speculatively execute both and abort when the first finishes.

```c
int long_computation_1(void* args);
int long_computation_2(void* args);

int first(void* args1, void* args2) {
    int x;
    inlet void reduce(int r) {
        x = r;
        abort;
    }
    reduce(cilk_spawn long_computation_1(args1));
    reduce(cilk_spawn long_computation_2(args2));
    cilk_sync;
    return x;
}
```
Simple Example: Native abort & Inlets

- Imagine you have two computations that should yield the same result, but one could take significantly longer to compute.
  - Speculatively execute both and abort when the first finishes.

```c
int long_computation_1(void* args);
int long_computation_2(void* args);

int first(void* args1, void* args2) {
    int x;
    inlet void reduce(int r) {
        x = r;
        abort;
    }
    reduce(cilk_spawn long_computation_1(args1));
    reduce(cilk_spawn long_computation_2(args2));
    cilk_sync;
    return x;
}
```

- No support for abort or inlet in cilk++.
Outline

1. A Recipe for Speculation
   • Porting the Example
     • Implementing abort

2. Evaluation
   • Performance
   • Complexity Porting Old Code

3. Conclusions
Handling inlets

- Inlets are local functions that merge the result of spawned computations.
  - Serve a similar purpose to reducers, but a little more general.
  - Get access to the parent function’s stack frame.

**Semantics**

- Inlets are locally serial, i.e. all of the inlets for a particular stack frame will run serially.
- The order of executing them is non-deterministic, implementation based on the amount of time that the computations take.
Handling inlets

```c
int long_computation_1(void* args);
int long_computation_2(void* args);

int first(void* args1, void* args2) {
    int x;
    inlet void reduce(int r) {
        x = r;
        abort;
    }
    reduce(cilk_spawn long_computation_1(args1));
    reduce(cilk_spawn long_computation_2(args2));
    cilk_sync;
    return x;
}
```
Handling inlets – via Translation

- Translate inlets into continuations.
  - Not a particularly painful translation.
  - It can be annoying to work with continuations in C code.
  - Ideally this would be a compilation step.
- Mostly mechanical translation preserves semantics
  - Depending on how they are used, it might be more efficient to use reducers.
cilk5 Code

```c
int f_1(void* args);
int f_2(void* args);

int first(void* args1, void* args2) {
    int x;
    inlet void reduce(int r) {
        x = r;
        abort;
    }
    reduce(cilk_spawn f_1(args1));
    reduce(cilk_spawn f_2(args2));
    cilk_sync;
    return x;
}
```
Translated \textit{cilk++} Code

\begin{verbatim}
struct InletEnv { cilk::mutex m; int x; };

int f_1(void* args, int(*cont)(int, InletEnv*), InletEnv* env);
int f_2(void* args, int(*cont)(int, InletEnv*), InletEnv* env);

int first_inlet(int result, InletEnv* env) {
  env->m.lock();  // Serial execution
  env->x = result;
  abort;
  env->m.unlock();  // Serial execution
}

int first(void* args1, void* args2) {
  InletEnv env;
  cilk_spawn f_1(args1, first_inlet, env);
  cilk_spawn f_2(args2, first_inlet, env);
  cilk_sync;
  return env.x;
}
\end{verbatim}
Translated \textit{cilk++} Code

```c
struct InletEnv { cilk::mutex m; int x; }

int f_1(void* args, int(*cont)(int, InletEnv*), InletEnv* env);
int f_2(void* args, int(*cont)(int, InletEnv*), InletEnv* env);

int first_inlet(int result, InletEnv* env) {
    env->m.lock(); // Serial execution
    env->x = result;
    abort;
    env->m.unlock(); // Serial execution
}

int first(void* args1, void* args2) {
    InletEnv env;
    cilk_spawn f_1(args1, first_inlet, env);
    cilk_spawn f_2(args2, first_inlet, env);
    cilk_sync;
    return env.x;
}
```
**Translated \textit{cilk++} Code**

```c++
struct InletEnv {
    cilk::mutex m;
    int x;
};

int f_1(void* args, int(*)(int, InletEnv*), InletEnv* env);
int f_2(void* args, int(*)(int, InletEnv*), InletEnv* env);

int first_inlet(int result, InletEnv* env) {
    env->m.lock();  // Serial execution
    env->x = result;
    abort;          // Still need to handle abort
    env->m.unlock();  // Serial execution
}

int first(void* args1, void* args2) {
    InletEnv env;
    cilk_spawn f_1(args1, first_inlet, env);
    cilk_spawn f_2(args2, first_inlet, env);
    cilk_sync;
    return env.x;
}
```

Ruben Perez & Gregory Malecha (MIT) Speculative Parallelism in Cilk++ May 11, 2010 17 / 33
Outline

1. A Recipe for Speculation
   - Porting the Example
   - Implementing abort

2. Evaluation
   - Performance
   - Complexity
   - Porting Old Code

3. Conclusions
We saw a notion of global abort in Lab 5.
- When you abort, you abort everything, completely done.
- This is not compositional.

Compositionality
Compositionality requires the ability to abort speculating computations. Need an abort hierarchy.
Hierarchical abort

- Abort any subtree of the computation without affecting the rest of the computations.
Hierarchical abort

Abort any subtree of the computation without affecting the rest of the computations.
Hierarchical abort

- Abort any subtree of the computation without affecting the rest of the computations.
User-Implemented abort

- Implement using polling...
User-Implemented abort

- Implement using polling...
- Two possible implementations:
  - Poll up toward the root.
User-Implemented abort

- Implement using polling...
- Two possible implementations:
  - Poll up toward the root.

Should I abort?  

```
  0 0
  |   |
X  0  0
  |   |
  0 0
```

Ruben Perez & Gregory Malecha (MIT)  Speculative Parallelism in Cilk++  May 11, 2010
User-Implemented abort

- Implement using polling...
- Two possible implementations:
  1. Poll up toward the root.

Should I abort?
User-Implemented abort

- Implement using polling...
- Two possible implementations:
  1. **Poll up** toward the root.
User-Implemented abort

- Implement using polling...
- Two possible implementations:
  1. Poll up toward the root.

Should I abort? Yes
User-Implemented abort

- Implement using polling...
- Two possible implementations:
  1. Poll up toward the root.
     - Also, lazily copy values downward.

```plaintext
Should I abort? 0 0
```

```
0
```

```
X
```

```
0
```

```
0 0
```

```
0 0
```

```
0 0
```

```
0 0
```
A Recipe for Speculation

Implement abort

User-Implemented abort

- Implement using polling...
- Two possible implementations:
  1. Poll up toward the root.
  2. Also, lazily copy values downward.

Should I abort?

Ruben Perez & Gregory Malecha (MIT)
Speculative Parallelism in Cilk++
May 11, 2010
User-Implemented abort

- Implement using polling...
- Two possible implementations:
  1. **Poll up** toward the root.
     - Also, lazily copy values downward.

```
    o
   / \
  O   O
 /   / \
X   O   O
 /   /   \
0   0   0
   /   /   \
  0   0   0
```

Should I abort?
User-Implemented abort

- Implement using polling...
- Two possible implementations:
  1. **Poll up** toward the root.
     - *Also, lazily copy values downward.*

```plaintext
Should I abort? 

```

```

```

```

```

```

```

```

```
**User-Implemented abort**

- Implement using polling...
- Two possible implementations:
  - **Poll up** toward the root.
    - *Also, lazily copy values downward.*

```
0

X

X  0

X  0

0

0  0

0  0
```

Should I abort? **Yes**
User-Implemented abort

- Implement using polling...
- Two possible implementations:
  1. **Poll up** toward the root.
     - *Also, lazily copy values downward.*
  2. **Abort down**, push the abort flag down the tree

![Diagram of a tree with nodes labeled 0 and X, showing the abort flag propagation.]
User-Implemented abort

- Implement using polling...
- Two possible implementations:
  1. Poll up toward the root.
     - Also, lazily copy values downward.
  2. Abort down, push the abort flag down the tree
User-Implemented abort

- Implement using polling...
- Two possible implementations:
  1. **Poll up** toward the root.
     - Also, lazily copy values downward.
  2. **Abort down**, push the abort flag down the tree
User-Implemented abort

- Implement using polling...
- Two possible implementations:
  1. **Poll up** toward the root.
     - *Also, lazily copy values downward.*
  2. **Abort down**, push the abort flag down the tree

```
                    X
                   /|
                  / |
                 X  0
                /   |
               /    |
              0      0
```

Abort!
User-Implemented abort

- Implement using polling...
- Two possible implementations:
  1. Poll up toward the root.
     - Also, lazily copy values downward.
  2. Abort down, push the abort flag down the tree
User-Implemented abort

- Implement using polling...
- Two possible implementations:
  1. **Poll up** toward the root.
  2. **Abort down**, push the abort flag down the tree.

```
Abort!
X
X X
X X

0
```

```
0
```

Also, *lazily copy values downward.*
High-level Performance

- **Poll-Up**
  - Polling is expensive (linear in depth of node)
  - `abort` is cheap (constant time, single memory access).

- **Abort-Down**
  - Polling is cheap (constant time, single memory access).
  - `abort` is expensive (linear in the size of the subtree).
Implementing `abort`

### High-level Performance

- **Poll-Up**
  - Polling is expensive (linear in depth of node)
  - `abort` is cheap (constant time, single memory access).
  - *Really simple implementation.*

- **Abort-Down**
  - Polling is cheap (constant time, single memory access).
  - `abort` is expensive (linear in the *size* of the subtree).
  - *Code is much more complex.*
  - *Some extra overhead, currently using a locking implementation.*
Outline

1. A Recipe for Speculation
   - Porting the Example
   - Implementing abort

2. Evaluation
   - Performance
   - Complexity Porting Old Code

3. Conclusions
Outline

1. A Recipe for Speculation
   - Porting the Example
   - Implementing abort

2. Evaluation
   - Performance
   - Complexity Porting Old Code

3. Conclusions
Different Flavors of Abort

- Compare the different abort techniques.
  - **Build** a tree of height 14 and branching factor of 4.
  - **Poll** at each leaf 100 times.
  - **Abort** the root.
  - **Poll** at each leaf.

Note that **abort** is serial.
Different Flavors of Abort

- Compare the different abort techniques.
  - **Build** a tree of height 20 and branching factor of 2.
  - **Poll** at each leaf 100 times.
  - **Abort** the root.
  - **Poll** at each leaf.

Note that **abort** is serial.
Polling Granularity Effects – Nodes Explored

- We’re running our ported pousse code (with some instrumentation).
- **Granularity N** doesn’t poll at the lowest $N$ levels of the tree.
- **Base** only checks abort in the loop that spawns the child computations.
Polling Granularity Effects – # of Polls

- We’re running our ported pousse code (with some instrumentation).
- **Granularity N** doesn’t poll at the lowest $N$ levels of the tree.
- **Base** only checks abort in the loop that spawns the child computations.

![Number of Polls](chart.png)

- **Granularity 3**
- **Granularity 4**
- **Granularity 5**
- **Base**
Polling Granularity Effects – # of Aborts

- We’re running our ported pousse code (with some instrumentation).
- **Granularity N** doesn’t poll at the lowest $N$ levels of the tree.
- **Base** only checks abort in the loop that spawns the child computations.

![Number of Aborts](image-url)
Outline

1. A Recipe for Speculation
   - Porting the Example
   - Implementing abort

2. Evaluation
   - Performance
   - Complexity Porting Old Code

3. Conclusions
Experience Porting Cilk Code

- We ported Cilk Pousse \(^1\) from cilk to cilk++.  

<table>
<thead>
<tr>
<th></th>
<th>Lines</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cilk Pousse</td>
<td>956</td>
</tr>
<tr>
<td>Cilk++ Pousse</td>
<td>1011</td>
</tr>
<tr>
<td>Increase</td>
<td>(\approx 5.07%)</td>
</tr>
</tbody>
</table>

- Pretty simple with the inlet translation.  
  - Most annoying part is adding the calls to poll.  
    - This can use some tuning.  
  - Code is a little more difficult to read, but not too bad.  

- The real problem is that this changes the interface.  
  - Need to pass around the abort object.  
  - If it is used with an inlet, need to add continuation.  
  - Whole-code transformation is bad.

\(^1\)people.csail.mit.edu/pousse/
Outline

1. A Recipe for Speculation
   - Porting the Example
   - Implementing abort

2. Evaluation
   - Performance
   - Complexity Porting Old Code

3. Conclusions
Speculations on Speculative Parallelism

- Speculative parallelism is definitely implementable as a library.
  - Native runtime support might be more efficient/nicer.
- Lack of some abstraction features makes using it at the high-level require interface changes.
- ...but, using it locally at the leaves does not leak into the rest of the code.