Speculative Parallelism in Intel Cilk Plus

by

Ruben Perez

B.S. Computer Science, Massachusetts Institute of Technology, 2011

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2012

Signature of Author .... [illegible]
Department of Electrical Engineering and Computer Science
May 21, 2012

Certified by .... [illegible]
Professor Charles Leiserson, Thesis Supervisor
May 21, 2012

Accepted by .... [illegible]
Prof. Dennis M. Freeman Chairman, Masters of Engineering
Thesis Committee

Speculative Parallelism in Intel Cilk Plus

by

Ruben Perez

Submitted to the Department of Electrical Engineering and Computer Science
on May 21, 2012, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

Certain algorithms can be effectively parallelized at the cost of performing some redundant work. One example is searching an unordered tree graph for a particular node. Each subtree can be searched in parallel by a separate thread. Once a single thread is successful, however, the work of the others is unneeded and should be ended. This type of computation is known as speculative parallelism. Typically, an abort command is provided in the programming language to provide this functionality, but some languages do not. This thesis shows how support for the abort command can be provided as a user-level library. A parallel version of the alpha beta search algorithm demonstrates its effectivenesss.

Thesis Supervisor:
Charles E. Leiserson, Professor

Acknowledgments

Many thanks to my family. Their unconditional support means more than they know.

I would like to thank my advisor, Professor Charles Leiserson of MIT CSAIL. His support and encouragement led me to explore areas I would not have considered at all.

A special thanks to Don Dailey. His help in making a test framework for this project was invaluable.

Thanks to the Supertech Research Group. Their feedback was extremely helpful, not only for improving my work, but also my speaking and presentation skills.

A big thanks to my friends. Thanks to Erek Speed for encouraging me to enter the M.Eng program. Thansk to Tom Lieber for reviewing my thesis draft on the first days of his vacation. Thanks to Dan Logsdon and Joey Whempner for their humor and encouragement.

One last thanks to the students of Fall 2011 MIT course 6.172, *Performance Engineering Of Software Systems*. Their use of my work provided essential feedback on both design and quality of code.

Contents

List of Figures

1 Introduction and Motivation

Let us assume we have two ways of computing a solution to a problem. We have no idea which method is quicker, but know that neither method benefits from having multiple processors. Since we have two available processors and no other work to perform, we can have the answer computed in parallel by having one processor compute it using the first method, and the second processor with the second method. If one finishes sooner, we terminate the other's work and move on to other work. This type of parallelism is known as *speculative parallelism.* Given that we have more resources that we can use, we speculate that some work may be useful, although potentially redundant, and perform it anyway. If and when its results are found to be no longer needed, it is halted so that resources dedicated to that computation can be allocated to more useful tasks. Many languages offering support for threads also offer an abort command primitive to terminate a thread. One particular language that does not is Intel Cilk Plus, a set of parallel programming language extensions to C++. I have implemented an easy way to express this type of parallelism at the user level for Intel Cilk Plus. To demonstrate the use of this library in a compelling fashion, I have also implemented a parallel version of the alpha beta search algorithm.

The remainder of this section is broken down as follows. Section 1.1 begins with a brief overview of the Cilk programming language and its history. Speculative parallelism has been implemented in older versions of Cilk, so Section 1.2 will go over the mechanism by which it is expressed.

1.1 Basics of Cilk

Cilk is designed to be a set of extensions to the C/C++ languages whose goal is to express parallelism by having the programmer simply identify what code can safely execute in parallel, while leaving to the runtime system the details of how to distribute that work to processors. The first version, MIT Cilk, was developed at the MIT Laboratory for Computer Science [5], as an extension to the C programming language. Cilk Arts, Inc. built upon this work and created Cilk++, which supported C++ while adding several useful features such as a scalability analyzer and parallel for-loops. Recently, Cilk Arts, Inc was acquired by Intel Corporation, which combined Cilk's parallelism philosophy with their own work in parallelism into Intel Cilk Plus.

```
int x = 0;
cilk_spawn some_func();
other_func();
cilk_sync;
x = 1;
```

Figure 1: Cilk_spawn and cilk_sync

```
cilk_for(int i = 0; i < 10; i++) {
  some_array[i] += 1;
}
```

Figure 2: Cilk_for

Intel Cilk Plus is notable for having three main keywords for expressing parallelism, which we will describe here.

cilk_spawn Indicates the function can be performed in parallel. The runtime system may schedule this work to be performed in an idle processor. In Figure 1 both `some_func()` and `other_func()` can execute in parallel. Given two processors, execution would begin with one processor initializing `x`. The `cilk_spawn` keyword indicates that `some_func()` can be executed in parallel, and it is added to a deque of work(this will be further discussed in Section 2.1). Execution continues on to `other_func()`. The second idle processor then takes execution of `some_func()` from the deque and executes it concurrently with `other_func()`.

cilk_sync Acts a barrier. Execution halts at this instruction until all threads spawned in the function have returned. In the previous example, the variable `x` is not set until both `some_func()` and `other_func()` have returned.

cilk_for A parallel version of a standard for loop. Iterations of the for-loop body are performed in parallel. In Figure 2, such a loop is used to increment the elements of an array in parallel. There is an implicit `cilk_sync` at the end of the for loop, so execution will not continue past the loop until all iterations of the loop body have completed.

With just these three simple keywords, a surprisingly large number of useful parallel programs can be written, but additional tools are needed for speculative computation.

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

int first(void* args) {
  int x;

  inlet void reduce(int result) {
    x = r;
    abort;
  }

  reduce( spawn f_1(args));
  reduce( spawn f_2(args));
  sync;

  return x;
}
```

Figure 3: Using Inlets and Abort in MIT Cilk-5 for Speculative Computation

1.2 Speculative Parallelism Semantics

MIT Cilk-5 implemented speculative computation with two additional linguistic mechanisms, the *abort* command and *inlet* functions. The `abort` command, much like the name implies, will attempt to terminate spawned computations.

Inlets, which are denoted by the *inlet* keyword, are functions used to process the result of a spawned computation. In many respects they are similar to a nested function. The first argument of an inlet is a spawned computation, but the inlet is invoked only when the spawned function completes. Execution does not wait for that spawned computation to complete, however. Instead, execution resumes to the next instruction. Furthermore, inlets are guaranteed to run serially. Even if two spawned computations return at the same time, only one invocation of the inlet function runs at a time. This makes reasoning about thread safety relatively easy, as there is no need to worry about the inlet simulaneously updating variables. With these two tools, speculative parallelism can be expressed as shown in Figure 3. Note that in MIT Cilk-5, the `cilk_spawn` command is replaced by `spawn` and `cilk_sync` is replaced by `sync`. Their functionality is the same.

Here we have two functions `f_1()` and `f_2()` which represent two methods of calculating some integer value. We do not know which is faster, but since two processors are available, both are run in parallel using `spawn`. The inlet function reduce is used to set the variable `x` with the result. It is important to note that while the `reduce` inlet is not prefaced with `spawn`, execution continues past those statements to the `sync` instruction. Execution pauses at the `sync` instruction until either `f_1` or `f_2` completes, at which point the `reduce` inlet is invoked with the result of the computation. Because the inlet function is nested within the first function, the `x` variable is within scope and accessible, and it is set to the value of the result. Once the value of `x` is set, the `abort` command is used to terminate the execution of the other function. When both threads have completed, execution continues past the `sync` statement, and `x` is returned.

We stated at the beginning of this section that the `abort` command will attempt to terminate spawned computations. Due to its implementation, there are few guarantees about if and when they are ended. In this example, even after `f_1` sets the value of `x` and signals an abort, `f_2` may finish computation and also set the value of `x`. Should this be undesired behavior, it is a simple matter to add code to the inlet to prevent it. If the computation is aborted, we can think of the thread executing it as effectively killed; no value is returned from the function.

Aborts and inlets provide an intuitive framework for expressing speculative parallelism. The pattern of using the inlet, which processes results of computations, to abort currently running computations is an easy one to conceptualize, but perhaps not to implement.

These two useful mechanisms, `abort` and inlets, were removed from later versions of Cilk. As MIT Cilk-5 evolved into Cilk++ and Intel Cilk Plus, support for C++ was added. C++ does not allow for nested functions, making an implementation of inlets difficult. As we will see in Section 2.2, implementing abort in an object-oriented language is also difficult.

1.3 Contributions and Thesis Outline

This thesis provides the following contributions

- An Abort library for speculative parallelism in Intel Cilk Plus
- An implementation of parallel alpha-beta search using the Abort library. This implementation will be included with this thesis on DSpace@MIT, and maintained at `http://github.com/rubenmp8`

The remainder of the thesis is broken up as follows. Section 2 discusses prior implementations of speculative parallelism and describes a polling approach for an implementation. Section 3 explains precisely how the Abort object and inlets are implemented in Intel Cilk Plus. Section 4 covers the alpha-beta search algorithm and how it pertains to the problem of speculative parallelism. Section 5 discusses the implementation details. There are a few algorithm specific issues relating to speculative parallelism, but they illustrate potential issues to watch out for when implementing other algorithms. Sections 6 and 7 discuss the possible effects of varying polling granularity and measured results respectively. Lastly, Section 8 provides a discussion of future work and closing thoughts.

2 Prior Work

This section will discuss possible implementations for implementing speculative parallelism. MIT Cilk 5 provided effective language mechanisms for speculative parallelism, whose implementation will be examined in Section 2.1. Cilk was also ported to the object oriented language Java as JCilk [3]. Implementation of abort and inlets in this language will be examined in Section 2.2. Previous work with Malecha [8] explored polling to implement abort in Cilk++, which is discussed in 2.3.

2.1 Speculative Parallelism Implementation in MIT Cilk-5

Before getting into the finer points of abort's implementation in MIT Cilk-5, we must take a look at Cilk's runtime scheduling algorithm. As stated in Section 1.1 that a `cilk_spawn` indicates that work can be performed in parallel, but that it may not necessarily be done in parallel. Cilk uses a "work-stealing" algorithm [5] which can be briefly summarized as follows. The runtime creates a set of worker threads, ideally one per processor. Each worker owns a deque of procedures to execute. The worker adds and removes spawned function frames to and from only the head of its deque, essentially treating it as a call stack for functions that can be executed in parallel. If a worker's deque is empty, it attempts to "steal" work from the tail of another worker's deque by removing the frame and executing it.

In MIT Cilk-5, an abort causes the runtime to visit the deques of workers operating on spawned computations and set an exception variable associated with the deque. This variable is always checked upon a pop operation from the deque, causing an immediate return if set. Because of the fact that an abort can occur only at a pop from the deque, however, it is quite possible that execution continues for a long time before the abort actually executes. The designers of MIT Cilk-5 considered this eventuality to be unlikely, and accepted this limitation since a solution was judged to be too costly in performance.

Inlets are similar to nested functions, and are implemented by adding a mutex to ensure serial execution. Although nested functions are not part of the ANSI C standard, some compilers, such as the GNU C compiler upon with Cilk is based [4], do support them. By having a mutex lock and unlock before and after the inlet's execution, we can ensure the inlet is run by only one thread at a time.

2.2 Implementation in JCilk

JCilk is an implementation of Cilk in the Java programming language [3]. It provides support for the abort command and inlets by using the exception mechanisms of Java. Spawned computations are contained within try blocks. When aborts occurs, an exception is thrown. The exception handler, the catch block, can then act as an inlet by processing results, or more importantly performing cleanup or deallocation of resources.

This approach of using exceptions is not well suited for C++. Asynchronous exception handling is also not precisely specified by the C++ standard, and it can also hinder performance optimizations. Furthermore, the presence of object destructors executing when an exception occurs (potentially anytime, even during an object constructors or destructors) makes reasoning about safety difficult. Consequently, using exceptions the way JCilk does for speculative parallelism support was judged not feasible for later versions of Cilk.

2.3 Polling Approach for Abort

Because of the lack of runtime support for signaling to a spawned computation that it should terminate, we instead take the approach that spawned computations should periodically poll to determine if they should terminate. For the above example, or a general tree search, spawned computations could simply poll a shared boolean flag in the following manner. The flag is initally set to false. Spawned computations periodically poll this flag during computation. If the flag is set, they return early, otherwise they continue working. Once any spawned computation has successfully completed computation, the flag is set.

In some tree searches such as alpha-beta search, however, the user may need to abort only the search of a specific subtree. Furthermore, the search of a tree node may recursively search subtrees or spawn additional work. We can model this spawning of computations as a hierarchial relationship, where aborting a particular computation should also abort any computations it spawned recursively.

Abort Using the Invocation Tree Model

To conceptualize this relationship, we model function calls as an *invocation tree*. In an invocation tree, a node represents a function call, and its children are functions it has spawned. Figure 4 shows an invocation tree for the speculative parallelism example in Figure 3. To make the example slightly more interesting, we assume that function `f_2()` now also spawns two functions itself, `A()` and `B()`.

From this invocation tree we can determine precisely under which conditions any particular spawned computation should abort. An execution of `B()` needs to abort if and only if either `first()` or `f_2()` executes an abort instruction. Function `f_1()` needs to abort if and only if `first()` executes an abort. Generally, we can see that a spawned computation needs to abort if any node along the path to the root has executed an abort instruction.

In previous work with Malecha [9], I explored the viability of implementing abort as a user-level library in Cilk++ by using this polling approach. Each spawned computation keeps an Abort object, containing a pointer to a parent Abort object and a boolean flag indicating aborted status.

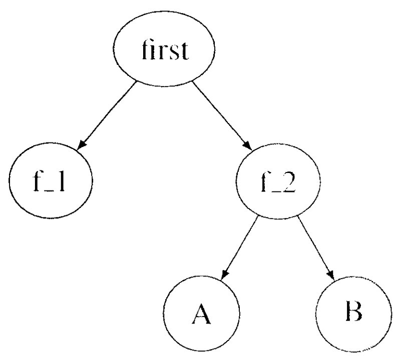


Figure 4: Invocation Tree for Speculative Parallelism Example

The tree formed by the pointers of the Abort objects directly mirrors the invocation tree.

With the invocation tree directly available, several approaches for implementing abort are possible. One approach to is to have a spawned computation poll status by checking if any node along the path to the root has signaled an abort by setting its aborted status boolean, an approach I call the *poll-up* approach. If so, then the spawned computation should return immediately. If a spawned computation wishes to signal an abort, it sets only its own flag and relies on spawned computations to check it. This protocol makes executing the abort a constant time operation, since only a single boolean is set. Checking whether early termination is needed, however, takes time proportional to the height of the invocation tree.

Alternatively, abort could set the boolean flag of its descendents, so that polling need check only a single boolean. I call this approach the *push-down* approach. In this push-down approach, abort takes time proportional to the size of the subtree, but polling only constant time. The push-down approach requires a great deal of care in ensuring that children have not terminated normally when an abort is signaled. Otherwise, there is a potential race condition where an attempt to update a node of the invocation tree occurs at the same time its memory is being deallocated after completing its work normally. Ultimately, Malecha and I judged the cost of synchronizing as too high.

In both the poll-up and push-down approaches, user code must periodically poll status. If their status is determined as aborted, the user code can safely return early at its convenience. This provides an opportunity for deallocating any resources, and performing any other necessary cleanup before invoking the inlet.

Malecha and I simulate inlets as normal functions whose arguments were references to variables in their "parent" frame, essentially acting as closures. To ensure serialization, we used a mutex to prevent two threads from executing that code at the same time. While functional, this approach is somewhat unwieldly. When changes to which variables the inlet references are made, it requires changing all inlet function signatures.

Semantic Differences with Polling

There are a few semantic differences between MIT Cilk-5's abort and the poll-up approach for abort. In the poll-up approach, all spawned computations in the poll-up approach have a return value. In MIT Cilk-5, an aborted computation is essentially terminated and has no return value. The inlet is invoked only with the results of a completed computation. With the poll-up approach, aborted comptuations will still return a value, since they "terminate" by returning early. It is up to the inlet function to determine which return values are the useless result of an aborted computation and which are valid results.

More importantly, user code must periodically poll status and return early if aborted. This is both an advantage and a disadvantage. It is beneficial because the user can guarantee the aborted functions return relatively soon after an abort by polling frequently. The downside is that the user must use judgement on when and how often to poll. Polling is a potentially costly operation, taking time proportional to the height of the invocation tree. If polling is too frequent and there is no abort, we waste time polling that could be better spent performing our desired computation. If we poll too infrequently, then we potentially waste time executing an irrelevant computation. We call the parameter controlling the frequency of polling the *polling granularity*.

I have implemented both abort and inlets in the latest version of Cilk, Intel Cilk Plus. While a straightforward port of the poll-up approach is quite simple, the fact that Intel's compiler for Intel Cilk Plus, *icc*, supports some features of the C++11 standard allows for improvements to the design. I also explored the effect of varying polling granularity on performance in the alpha-beta search algorithm.

3 The Speculative Parallelism Library

This section will explain how the poll-up approach to abort is implemented in the `Abort` class, and how inlets are implemented using the lambda function feature of the C++11 standard. I will also show how these two mechanisms are used to implement speculative parallelism.

3.1 Abort Implementation

The Abort class, shown in Figure 5, is straightforward. It is essentially just a wrapper for a pointer to a parent `Abort` and a boolean for status.

The default constructor is used to create the root of an invocation tree, an `Abort` object with no parent. The copy constructor is used not to create a duplicate `Abort` object, but to create one that is the child of the argument `Abort` by appropriately setting the parent pointer.

The user can check whether a computation should be ended by using the `isAborted` function. In order to express polling granularity easily to the user, we take the approach that a user should have many `isAborted` calls in their code, but only some of those `isAborted` calls actually perform a check up the invocation tree. All other calls increment a counter and return false to indicate a not aborted status. Every *i*th call (where *i* is the polling granularity) to `isAborted` performs this potentially expensive check by recursively calling `isAbortedNoInc`, which always checks the parent's status regardless of the counter. Thus, it checks all `Abort` objects on the path to the root in the invocation tree. The function returns true if any of those flags are set, and false otherwise.

The user can modify the `pollingGranularity` variable in order to tune polling frequency for their particular needs. Initially, it is set to a compile time constant, `P_GRAN`. Section 6 discusses tuning `pollingGranularity` in more detail.

3.2 Inlet Implementation

The inlet function cannot simply be replaced with a call to a normal function because the inlet has access to the surrounding function's scope. In the example shown in Figure 3 , the inlet function `reduce` can modify the variable `x` in the first function's scope. It behaves exactly as a nested

```
class Abort {
public:
        Abort() : aborted(false), parent(NULL) {
            pollGranularity = P_GRAN;
            count = 0;
        }

        explicit Abort(Abort *p) : aborted(false), parent(p) {
            pollGranularity = P_GRAN;
            count = pollGranularity;
        }

        int isAborted() {
            //allows for polling every i'th call
            if(count >= pollGranularity - 1 ) {
                count = 0;
                return aborted || (parent && this->parent->isAbortedNoInc());
            }
            else {
                count++;
                return 0;
            }
        }

        //always poll, regardless of polling granularity
        int isAbortedNoInc() const {
            return aborted || (parent && this->parent->isAbortedNoInc());
        }

        void abort() {
            aborted = true;
        }

private:
        Abort *parent;
        bool aborted;
        int pollGranularity;
        int count;

};
```

Figure 5: The Abort Class

function, which some C compilers support. Unfortunately, nested functions are not supported in C++, and Intel Cilk Plus does not use them.

My previous work with Malecha in Cilk++ worked around this limitation by defining an inlet environment structure that contained references to any variables that the inlet function needed to access. The inlet function takes both the computed result and this environment structure as arguments. In order to ensure this inlet function is invoked, the user function takes both the environment structure and a function pointer to the inlet as arguments, and returns not the result, but rather the result of a call to the inlet function with the environment struct and computed value as arguments.

As one might imagine, this requires significant changes to any existing user code. The user must change their function signatures to take these additional arguments, as well as defining and setting up these inlet structures.

One of the new features of the C++11 standard, lambda functions, provides an excellent substitute. A lambda function allows the user to define a function, and the compiler automatically creates a function object for it. This function object can then be passed as an argument to a function or used in any way that an object can used. What makes lambda functions particularly powerful is that they can reference any variable defined in the same scope at the point of the lambda function's definition, either by value or reference [6], a process known as capturing variables. An informal spec for a lambda function is shown in Figure 6.

```
[capture_mode] (args) -> return_type {function body}
```

Figure 6: Lambda Function Specification in C++11

If capture mode is set to &, all variables defined at the time the lambda function itself is defined are captured by reference. A capture mode of = captures all variables by value. It is important to note that lambda functions are not truly functions, they are objects of a type generated by the compiler. Thus, they must be assigned to a variable of the compiler-defined type. By using the *auto* keyword, another feature of the C++11 standard, the lambda function can be stored in a variable whose type is determined by the type with which it is initialized.

```
 1 int f_1(void* args, Abort*);
 2 int f_2(void* args, Abort*);
 3
 4 int first(void* args) {
 5   int x;
 6   Abort abort;
 7   tbb:mutex m;
 8   auto reduce = [&](int r)->void {
 9     m.lock();
10     x = r;
11     abort.abort();
12     m.unlock();
13   }
14   cilk_spawn [&]{reduce( f_1(args, &abort));}();
15   cilk_spawn [&]{reduce( f_2(args, &abort));}();
16   cilk_sync;
17
18   return x;
19 }
```

Figure 7: Speculative Computation with Lambdas and Abort

3.3 Speculative Parallelism Using Abort and Lambda Functions

The example in Figure 3 with MIT Cilk-5 abort and inlets can now be translated into a version with lambda functions and Abort, as shown in Figure 7.

To provide the guarantee that the lambda function is run serially, a mutex is used. Again we spawn both computations, and when one returns, the lambda function is invoked. It would seem that the syntax in line 14 and 15 should be

```
1 cilk_spawn reduce( f_1(args, &abort));
```

This is not correct because this would indicate that `f_1` is executed in the parent thread before executing `reduce` as a spawned computation. Thus, an enclosing lambda function is used to ensure both the function and computation of its arguments execute as a single spawned computation.

Because the lambda functions capture mode is set to capture all variables by reference, any variable defined before it can be modified directly in the function. Now the code exactly mimics the behavior of a nested function in that we can access the enclosing scope. The mutex is locked, and the value of `x` is set. Lastly, the `abort` function of the `Abort` object is called before unlocking the mutex and returning. Though not shown, `f_1` and `f_2` must periodically poll the `Abort` object to determine if they should return early. Since the inlet is invoked even for aborted computations, it is also necessary to determine an invalid return value.

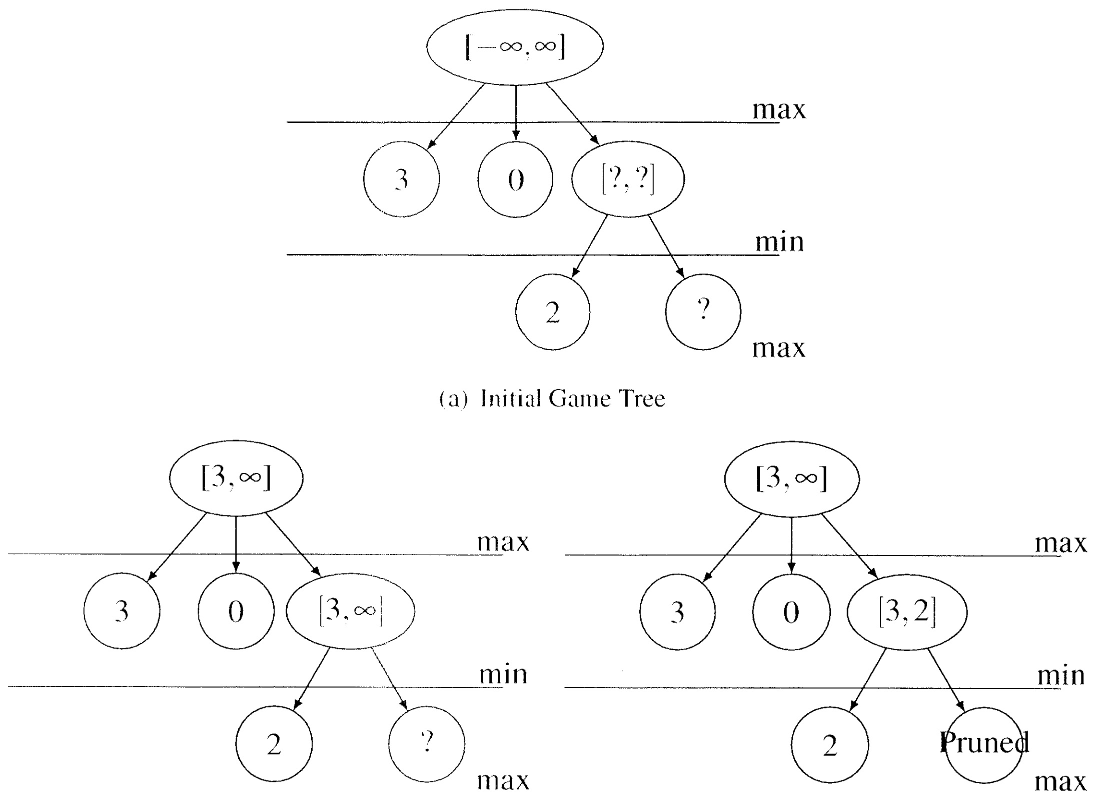
4 Alpha-Beta Search Algorithm

Before explaining how to use these tools to implement alpha-beta search, I will first go over the algorithm itself. I will then explain how alpha-beta search can be parallelized using speculative parallelism.

4.1 Alpha-Beta Overview

Two player adversarial games, such as chess or checkers, can be modeled as a game tree. In game trees, a node represents a possible game state. Child nodes represent game states that are possible transitions from the parent node's game state, making edges the possible moves. Alpha-beta search is a search algorithm used to search these trees in order to select the most beneficial move[7]. Each node is assigned a score based on how beneficial it is to the player using a static evaluator function. One player attempts to win by maximizing this score, and is known as the maximizing player. The other player wins by minimizing this score, and is known as the minimizing player.

Two bounds, alpha and beta, are kept at each node. The alpha value represents the minimum score the maximizing player is guaranteed to get at the end of the game if both players play optimally. A way to think of this is that from the root node, there exists a leaf in the tree with the score of alpha that would be reached if both players play optimally. Beta represents the worst score the opponent can force the maximizing player to get. From a node, there exists a leaf in the subtree that would be reached in optimal play yielding the beta score. If alpha is greater than or equal to beta at a node, this move is one that is invalid, because one player is playing suboptimally. The opponent allowed the player to reach a state with a high score, even though a move that yielded a lower score was available, the move that the beta value represents. This is explained in more detail in Section 4.2 with an example. This observation allows us to avoid searching parts of the game tree, a technique known as alpha-beta pruning. If searching subtrees yields such a condition, we can stop searching



(a) Initial Game Tree

(b) Search Rightmost Subtree

(c) Discovering a Prune

Figure 8: Alpha Beta Search Yielding a Prune

this part of the game tree, since it represents a part that would never be entered if both players are playing optimally.

4.2 Alpha-Beta Pruning

Let us walk through an example of an alpha beta prune, shown in Figure 8. In this game tree, the maximizing player is the first to move. The root is initialized with alpha and beta values of $-\infty$ and ∞ respectively since there are no bounds on the score yet. From this game state, the root node, the player has three possible moves. The leftmost two leaves have been statically evaluated to have scores of 3 and 0. Thus, before searching the right subtree, the player knows he is guaranteed a score of at least 3 by taking the leftmost move, causing the alpha value of the root to be updated to 3.

We begin the search of the rightmost subtree in Figure 8(b). The alpha and beta values are initialized to that of the parent state, 3 and ∞. At this node, the opposing player is seeking to minimize the score. The left leaf of this node returns a score of 2 which causes the beta value, the worst score the opponent can force on the player, to be 2.

Now we see why alpha can never be greater than or equal to beta. If the maximizing player chose the right subtree as his move, the opponent could immediately force him to a score of at most two. Depending on the scores of the rest of the subtree, he may be able to force him to an even lower score. Since the maximizing player already had a better move available–the game state yielding 3 at the leftmost child of the root–this move is strictly worse. Thus, we can skip the search of the rest of the right subtree, since there is no possibility of finding a better move than the one represented by alpha. The intuition is that once a move is determined to be a bad one, it is unnecessary to determine exactly how bad it is.

4.3 Parallelization of Alpha Beta Search

For the purposes of exploring the effectiveness of the abort mechanism, only a simple parallelization of the algorithm is needed. Much work has gone into effectively parallelizing alpha beta search, but for testing purposes the algorithm is parallelized by searching every subtree in parallel. Strictly speaking, a parallel implementation may perform more total work than a serial implementation, since a serial implementation may prune subtrees that the parallel version speculatively searches. In the example in Figure 8, if all three subtrees are searched in parallel, some time may be spent searching the pruned right subtree, since the search may begin before the alpha and beta values are updated to indicate a prune should occur.

This speculative nature of subtree searches is, however, what makes alpha-beta search such a compelling example for demonstrating Abort's effectiveness. When a prune condition occurs, I use the Abort mechanism described in Section 3 to end searches of affected subtrees. Because of the technique of searching every subtree in parallel, the invocation tree mirrors the game tree. This correspondence allows us to use the well-defined characteristics of the game tree, the known height and branching factor, to analyze performance.

5 Alpha-Beta Search Implementation

I will now discuss the some of the specifics of the alpha beta search implementation. A simplified version of the alpha-beta search function executing on a subtree is shown in Figure 9. Because the search of the root returns the best move instead of a score, different functions are used to search the root and subtrees that differ primarily in their returned value. Several optimization details are omitted for the sake of brevity.

5.1 Use of Abort and Lambdas in search()

The `search` function shown on line 1 of Figure 9 takes as arguments an `ABState` object representing the parent state called `prev`, the `ABState` object of currently searched game state, the depth to which to search, and a pointer to the `Abort` object of the parent state. This `Abort` object is used to construct a child `Abort` object for this search on line 3. By using the proper constructor, `localAbort`'s parent is set to be argument `Abort` object.

The inlet is the lambda function `search_catch`, defined on line 9. It is used to process the results of a search of a subtree. It takes as arguments the resulting score from searching the subtree, and the index of the searched move from the list of possible moves. The function uses these values to update the best known move and aborts if necessary. To ensure that no simultaneous updates of any of the variables occurs, a mutex is locked before the function executes and released when it completes.

After the inlet definition, the alpha and beta values for the `cur` game state are initialized. A poll is performed on lines 33 and 42 to verify this search is still needed. If it is not, a value of 0 is returned, which the inlet treats as an invalid score (a game state is unlikely to evaluate to 0). A vector of possible transition states is generated, and a `cilk_for` loop is used to search them in parallel. There is a subtle point to using a `cilk_for` loop to perform these searches. This code is easily the longest running part of the function. If an abort were to occur while this loop is running, we would ideally break out of the loop. This early termination is not possible in Cilk (that itself would be speculative parallelism!). Instead we rely on the fact that the searches return relatively quickly, as they are aborted. While we always iterate over the entire list of moves, the searches end

```
int search(ABState *prev, ABState *cur, int depth, Abort* abort ) {

    Abort localAbort = Abort(abort);
    tbb::mutex m;
    int local_best_move = INF;
    int bestscore = -INF;
    std::vector<ABState> cur_moves;

    auto search_catch = [&] (int ret_sc, int ret_mv ) {
      m.lock();
      ret_sc = -ret_sc;

      if (!(ret_sc > bestscore) || ret_sc == 0){
        m.unlock();
        return;
      }

      if (ret_sc > bestscore) {
        bestscore = ret_sc;
        local_best_move = ret_mv;
        if (ret_sc >= cur->beta) {
          localAbort.abort();
        }

        if (ret_sc > cur->alpha) cur->alpha = ret_sc;

      }
      m.unlock();
      return;
    };

    if (localAbort->isAborted() ) return 0;

    if (depth <= 0) return cur->evaluate();

    cur->getPossibleStates(cur_moves);
    cur->alpha = -prev->beta;
    cur->beta = -prev->alpha;

    if (localAbort->isAborted()) return 0;

    cilk_for(int stateInd = 0; stateInd < cur_moves.size(); stateInd++ ) {
        ABState* next_state = &next_moves[stateInd];
      search_catch( search(next, next_state, depth-1, &localAbort), stateInd);
    }
    return bestscore;
}
```

Figure 9: Simplified Version of Alpha Beta Search

quickly after the functions poll once.

5.2 Performance Considerations using Parallel For Loops

Alternatively, we could use a standard for-loop, and use `cilk_spawn` within it. This has the advantage of allowing polling within the for loop itself, and allowing us to break out of it if an abort is signalled. Also, the list of moves would generally be sorted with better moves at the beginning, as evaluated by some quick heurestic (In chess, moves that capture a bishop with a pawn are probably better than moves that capture nothing). If an ordered list of moves is searched sequentially, this strategy would make alpha-beta pruning occur more frequently, since better moves correspond to higher scores.

This problem is a sort of weak dependency between parallel tasks. While there is no absolute requirement for a particular task to execute before another for correctness reasons, there is a definite performance benefit to doing so. Cilk's philososphy is, however, to let the programmer define where parallelism may occur and have the runtime handle scheduling details. A `cilk_for` loop does not execute the iterations of the body in order for performance reasons. The scheduler breaks the loop's range into manageable contiguous chunks, such as 1 through 5 and 6 through 10. The loop bodies corresponding to those indices are delegated to available processors, which prevents the scheduler from creating the overhead of a spawn for every iteration of the loop while still using all available processors. However, this means that the iterations of the loop corresponding to `stateInd` being 10 and 15 may execute before the iterations for `stateInd` being 2, depending on how that range is divided.

A for-loop of `cilk_spawns` throws away those performance benefits but now the iterations of the loop will be added to work queues in order. The algorithmic benefit alone is almost certainly be worth the cost of additional spawns, and we also get the potential of properly polling and breaking out of the loop if needed. Unfortunately, at the time of writing, this approach currently causes segmentation faults for unknown reasons.

6 Methods

This section will discuss the hypothetical costs of using the poll-up approach in alpha-beta search. By providing an estimate of the worst case costs, we will see that a cost proportional to the height of the invocation tree is acceptable. We will also discuss the tradeoffs in varying polling granularity, and how they will be measured.

6.1 Performance Analysis with Alpha-Beta Search

The final implementation of Abort uses the poll-up approach, which checks whether the current computation should be aborted by checking the `isAborted` boolean of the parent `Abort` object, recursively up to the root of the tree. Thus, the cost of performing this `isAborted` check is proportional to the height of the invocation tree, or equivalently the height of the game tree. The game tree of most interesting games has a large branching factor: there are many moves possible from a given position. Thus, to look one more move ahead, which would require generating the tree for depth $d+1$ it would cost b^{d+1} where b is the branching factor.

The simplest parallelization of the algorithm searches all subtrees in parallel, which makes the game tree mirror the spawned computation tree since every game state is matched by a call to the search function. In most game scenarios a limit is imposed on the time allotted per turn. Given that time limit and the exponential growth rate of game tree size, the generated game tree tends to be relatively shallow, but wide. Chess, for example, has an average branching factor of about 35 [1], meaning that on average a node in the game tree has 35 children. Deep Blue, the high performance chess playing system which defeated world chess champion Gary Kasparov in 1997, could only perform a search to depth 12 in three minutes without pruning [2]. These data points indicates that even a performance cost that is $O(h)$ in game tree height is still practical. With performance rivaling Deep Blue and the poll-up Abort implementation, a poll requires accessing and comparing about a dozen boolean values.

6.2 Polling Granularity Variations

Perhaps the most immediate concern that comes to mind with the poll-up approach described is the simple fact that any time spent polling is time that is not spent doing useful work, regardless of how cheap each poll is. If we poll too often, then we return quickly after an abort has been signaled, at the cost of wasting time when aborts do not occur. If we poll too little, then we waste time performing work after an abort has been signalled. For a library whose goal is ease of use, this polling granularity must at least be easily exposed for user adjustment, or better yet automatically adjusted. As described in Section 3, the user should have frequent isAborted polling calls, since only some of them actually execute. The user can then adjust the polling granularity parameter to balance these tradeoffs.

Another question is where should polling calls be performed. It would seem ideal that they are performed every n cycles (where n is some tuning parameter), but this is difficult to accomplish from a user-level library. Generally, there should be no long running computations where a poll is not performed periodically. Otherwise the parent thread might signal an abort, but because the child is stuck in a long-running computation, it does not return until it is finished and a poll is performed, completely negating the point of the abort mechanism.

There are two points where it makes sense to perform polls, assuming a polling granularity of 1, i.e always polling. One is right before a spawned function performs any significant work, and another is before a significant amount of work is about to be spawned. Intuitively, these points make sense. We would not want to incur the overhead of spawning if an abort has been signalled. Since we cannot break out of a `cilk_for` loop, it makes sense that a poll before starting may save a considerable amount of work. If we use the `cilk_for` loop approach, the most effective way of "breaking" out of the loop when an abort occurs is to have all subtrees terminate quickly by polling before they start any significant computation.

7 Results

To evaluate the effectiveness of varying polling granularity, I used the alpha-beta search library described in Section 5 to implement a game engine for a game called Khet [8]. This game is somewhat similar to chess, but the precise rules of the game are irrelevant. Khet is characterized by a high branching factor (on the order of 50 [10]), which allows for high parallelism. I implemented several different versions of the engine that differ only in their polling granularity. I have the engines perform searches from several interesting game positions, both with a fixed time limit and to a particular depth.

In fixed depth searches, the time taken to search and number of nodes searched are the measures of efficiency. A lower time to search to a given depth *d* in the game tree is always better, since this means it takes less time to look *d* moves into the future. To ensure that pruning is eliminating parts of the game tree instead of exhaustively searching the entire game tree, I also measure the number of nodes searched.

7.1 Fixed Time Results

In fixed time searches, the average depth per search and the number of nodes searched show how efficient the engine is. A high average depth with a low number of nodes searched indicates that pruning effectively stops the search of parts of the game tree.

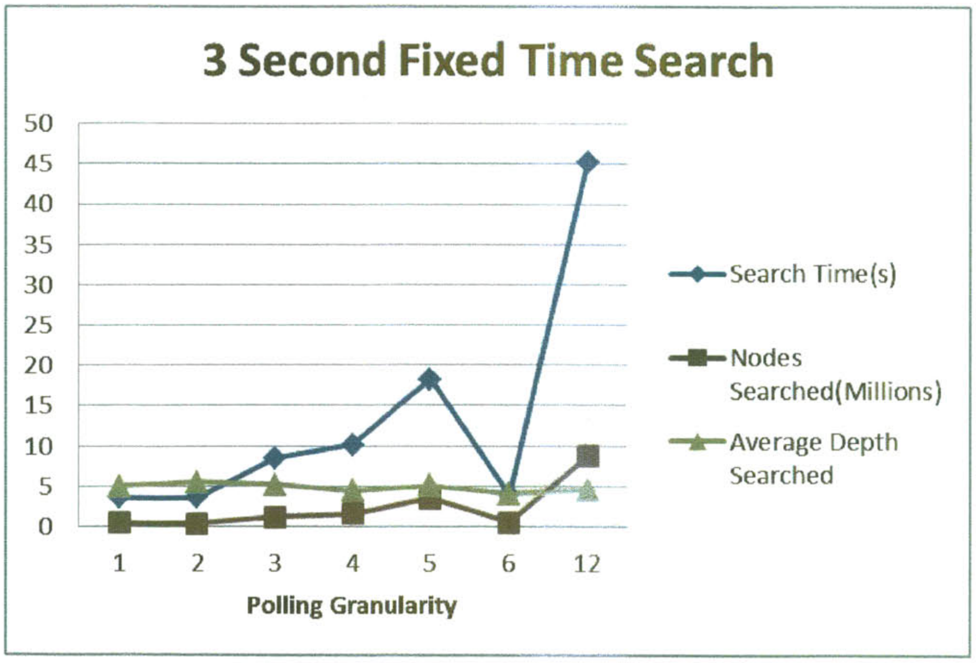


Figure 10: 20 Second Fixed Time Search

The first set of tests were short 3 second searches, whose results are shown in Figure 10. We see that as polling granularity increases, average depth remains relatively constant, while the number of nodes searched increases. This indicates that pruning does not occur at higher polling granularities because while an abort was signaled, a successful poll up the invocation tree to detect it does not occur quickly.

Another important consideration is the actual time taken per search. The engines are given the time allotted per search as a parameter, but they use the Abort class for aborting all searches. A timer thread creates an Abort object, which is used by all searches as the root of the invocation tree. Once time expires, the timer thread calls `abort` on the Abort object, which ends all running searches. At polling granularities past 2, the actual time per search greatly exceeds 3, indicating that at higher polling granularities, the spawned searches do not quickly detect a signaled abort.

There is an anomaly at polling granularity 6 caused by lack of data points. About 25 searches per polling granularity were performed. In all searches, the search process is halted by process termination after 90 seconds, with no statistics recorded. For polling granularity 6 and 12, this happened frequently. Only one search succeeded for polling granularity 6, which did succeed with a relatively short search time, but this is an outlier in the data.

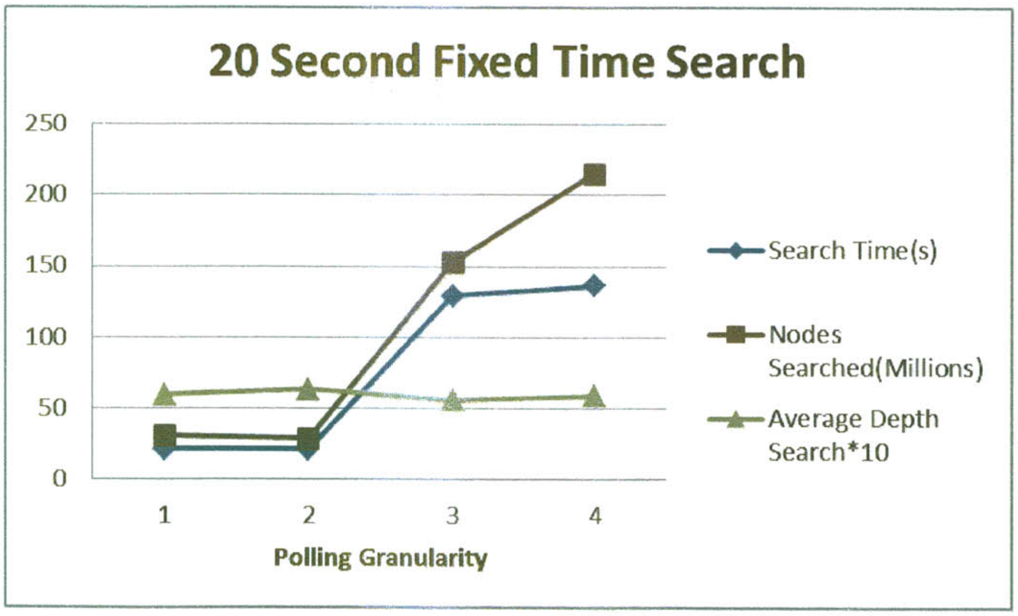


Figure 11: 20 Second Fixed Time Search

For longer searches of 20 seconds, shown in Figure 11, we see results consistent with short time searches. Again, average depth per search remains relatively constant at all polling granularities, but both nodes searched and time per search increase dramatically at polling granularities higher than 2. Only at polling granularities 1 and 2 do searches terminate in close to twenty seconds, again indicating that high polling granularities are not effective. Only results for polling granularities 1 through 4 are shown because at higher polling granularities, process timeout always occurs.

7.2 Depth Based Results

This set of tests vary polling granularity while executing searches to a particular depth. By measuring average time per search, a sense of relative efficiency can be determined. A lower average time per search is better, since that means the engine can look *n* moves ahead quickly. A lower number of nodes searched indicates that pruning effectively cuts down the search of the tree. Thus, a low number of nodes searched implies that polling quickly detected signaled aborts.

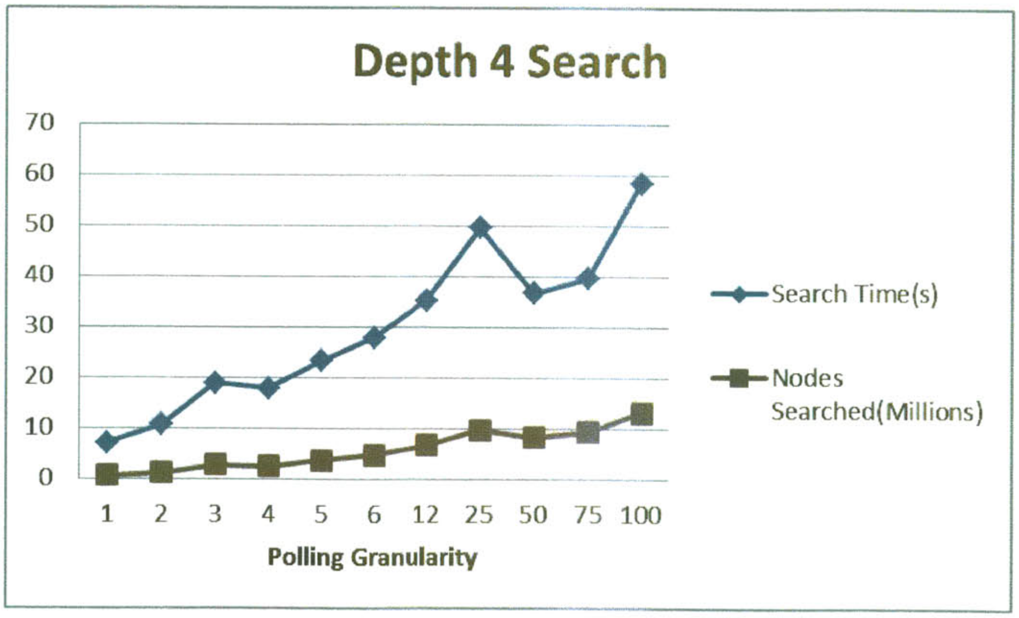


Figure 12: Time to Search to Depth 4

The results for searches to depth 4 are shown in Figure 12. We see a general trend of average time required and nodes searched increasing as polling granularity increases. In fixed time search tests, searches with a higher polling granularity take longer to end. Given that the Abort class was used to end the searches, it is clear that polls do not occur at high polling granularities. Thus, in fixed

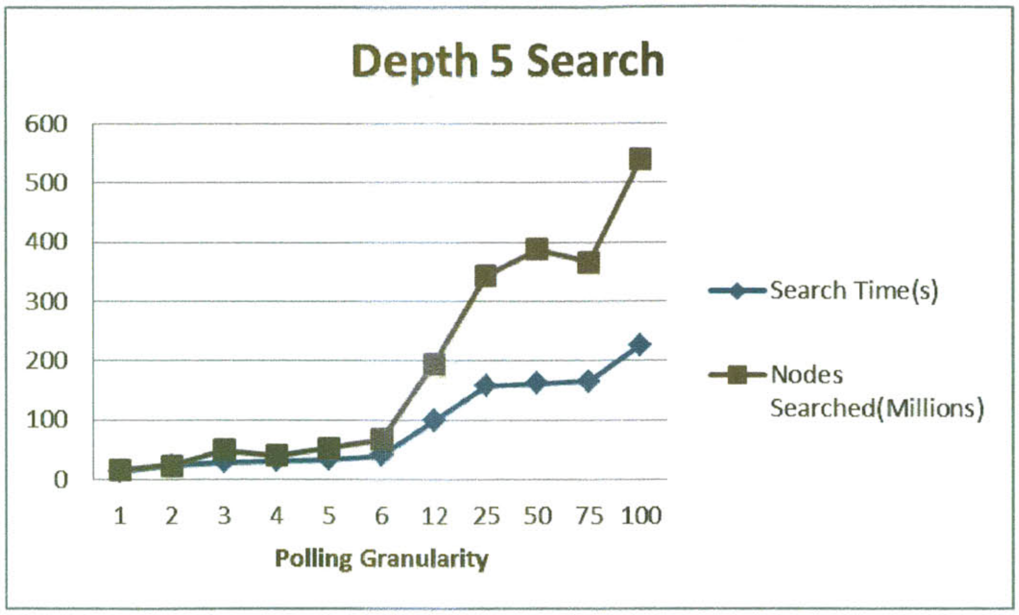


Figure 13: Time to Search to Depth 5

depth searches, the number of nodes explored should grow as polling granularity increases, since the lack of polling will cause an exhaustive search of the entire game tree. Indeed, this pattern is visible in searches to depths 5 and 6 as well, shown in Figure 13 and Figure 14 respectively.

7.3 Trends and Thoughts

These results indicate that a frequent polling level is more effective. This makes a sense when one considers that since invocation trees of alpha beta search are shallow, polls are relatively cheap when compared to the amount of work that may be saved. All of these tests show that a polling granularity of one is the most effective. Thus, inserting more `isAborted` calls in the code may yield even better results. Doing this, however, requires either polling within the for-loop itself, or polling in code other than the alpha-beta search code (during move generation or game state evaluation, for example).

One interesting last note is that this Abort library was used in the final project of a course at Massachussetts Institute of Technology called Performance Engineering Of Software Systems. The goal of the project was to optimize a Khet engine that used a serial implementation of alpha beta search with few optimizations. Students explored the challenges of effectively parallelizing the

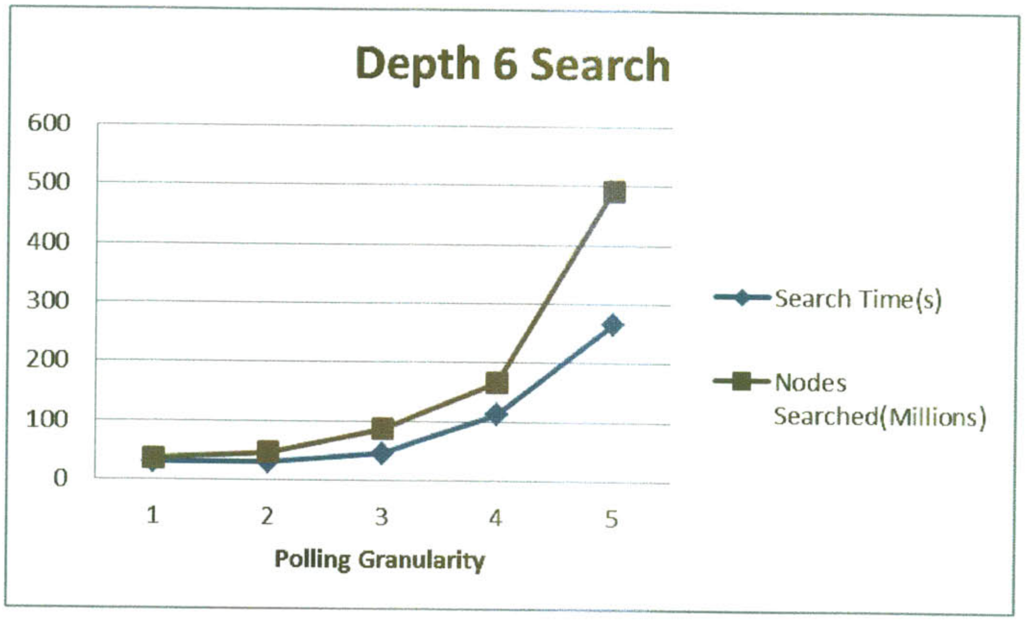


Figure 14: Time to Search to Depth 6

search algorithm, improving the data structures, and implementing heurestical improvements such as hash tables and ignoring moves that were not promising. In all cases, the most improvement in player strength as measured by the ELO rating system came from areas not related to parallelism. As their heurestics for reducing the effective branching factor improved, the benefit from parallelization decreased. This would indicate while there may be some performance increases from adjusting the polling granularity, the best use of time for improving performance comes from other areas. Verifying if adjusting polling granularity on other speculatively parallel algorithms has a more pronounced effect is an area of future research.

8 Conclusion and Future Work

We conclude with some thoughts for future work and closing remarks.

The framework presented here for expressing speculative parallelism is no way Cilk specific. While a direct comparison of this alpha beta search implementation to one in another language would not be fair simply because of the language differences, it would be relatively simply to use this Abort library in another parallel language. The most obvious example would be to port this library to C, and compare against the orignal MIT Cilk-5 implementation. This would allow for a direct comparison between the polling approach and a runtime abort.

Another area of interest would be to explore other use cases for speculative parallelism. A very common one is heuristic search such as alpha beta search, which is what the Abort library was designed in mind with. Since the invocation tree depth is seldom high in this application, the polling approach works well here, but that may not be the case in other applications. Other algiorithms such as Huffman coding, lexical analysis and JPEG decoding can be speculatively parallelized and may show different performance profiles because of different rates of abort, and height of invocation tree.

Lastly, the polling granularity idea has more potential than explored here. The analysis performed here was based only on statically setting the polling granularity. An potential area of interest would be experimentally adjusting the polling granularity. For example, polling when high in the invocation tree is very cheap, but could potentially save a lot of work. However, assuming that an abort is equally likely anywhere in the tree (a rather large assumption), the likelihood of the poll detching an abort would be lower. It may also be possible to adjust polling granularity at runtime based on frequency of previous aborts.

It is not clear if or when Intel will implement abort and inlets for Intel Cilk Plus. The work here shows that speculative parallelism can be implemented with ease at the user level, providing a viable option for now. While we cannot make any strong performance guarantees, the polling granularity parameter can be adjusted based on the application. Regardless of the specific performance numbers, this implementation of speculative parallelism is a useful tool that will allow implementation of algorithms that are not easily parallelizable in other ways.

References

[1] Louis Victor Allis, *Searching for solutions in games and artificial intelligence*, 1994.

[2] Murray Campbell, A. Joseph Hoane Jr., and Feng Hsiung Hsu, *Deep Blue*, ARTIFICIAL INTELLIGENCE **134** (2002), 57–83.

[3] John S. Danaher, I-Ting Angelina Lee, and Charles E. Leiserson, *Programming with exceptions in JCilk*, Science of Computer Programming (SCP) **63** (2006), no. 2, 147–171.

[4] Free Software Foundation, Inc. *Extentions to the C Language Family*

[5] Matteo Frigo, Charles E. Leiserson, and Keith H. Randall, *The implementation of the cilk-5 multithreaded language*, SIGPLAN Not. **33** (1998), no. 5, 212–223.

[6] Intel Corporation, *Intel X++ Compiler XE 12.0 User and Reference Guides*, November 2011.

[7] Donald E. Knuth and Ronald W. Moore, *An analysis of alpha-beta pruning*, Artif. Intell. **6** (1975), no. 4, 293–326.

[8] The Laser Game: Khet 2.0, 2012. Online; http://www.khet.com.

[9] Gregory Malecha and Ruben Perez, *Speculative parallelism in Cilk++*, Tech. report, Massachusetts Institute of Technology, 2010.

[10] J. A. M. Nijssen and J. W. H. M. Uiterwijk, *Using Intelligent Search Techniques to Play the Game Khet*, Proceedings of the 21st Benelux Conference on Artificial Intelligence (T. Calders, K. Tuyls, and M. Pechenizkiy, eds.), 2009, pp. 185–192.