A Multicore ROBDD Builder

Nicolas Guillemot

Abstract

The goal of this project was to implement a multicore-optimized ROBDD builder using Intel Threading Building Blocks, based on a Cilk [1] fork/join design presented by Yuxiong He [2]. In this design, the two recursive evaluations of “if-then-else” are split into two tasks, allowing the two branches to execute in parallel on different cores. Additionally, the data-structures used while building the ROBDD (such as the unique table and the computed table) are modified to operate in a thread-safe and scalable way. Major performance gains are seen through parallel execution, although it seems the results are quite quirky, perhaps due to the way parallel execution patterns lead to a different access pattern for the ROBDD builder’s caches.

Introduction

For this project, a ROBDD builder was built that uses multi-core optimizations to try increasing the runtime performance. This report explains the design of this ROBDD builder.

First, the user interface is explained, which mainly consists of explaining the functions supported by a Lua-based domain-specific language used to input programs for the ROBDD. Several sample programs are shown, including basic Boolean functions, a ripple-carry adder, solver for the n-queens puzzle, and a graph coloring module.

Next, the overall architecture of the ROBDD builder is described, which explains at a high level how the inputs to the program are transformed into its outputs. This explains how the inputted Lua program is converted into a sequence of instructions for the ROBDD builder, and how the results of the ROBDD builder can be used to generate a graph used for visualization.

After that, more details are given on the implementation of the ROBDD builder, including concerns for thread-safety and work scheduling. This mainly elaborates on the data-structures used to support the parallel version of the ROBDD builder.

Performance results are shown for the 12-queens puzzle. These results show significant speedups on Intel Core i7-5960X and Intel Core i7-6650U CPUs. However, these results have some surprising quirks, which are discussed.

Finally, some conclusions are made from the performance results so far, which should help decide future directions for this project. One of the main aspects of these conclusions are considerations for a potential future GPU implementation.

The implementation of this project can be found here: https://github.com/nlguillemot/robdd

The project uses Visual Studio 2015. Opening the solution and running it should “just work”.

User Interface

To conveniently define new problems for the ROBDD builder, a domain-specific language was designed using the Lua language. What follows is a sample program:

```
title = 'test'
display = true

a = input.a
b = input.b
c = input.c
r = a*b + a*c + b*c

output.r = r
```

Figure 1: A simple ROBDD program specification

This program declares three variables (a,b,c), builds a Boolean expression from them, and labels this expression with the label “r” in the output. As you might guess, the + and * operators are overloaded to mean “OR” and “AND” respectively. Furthermore, setting “display” on line 2 tells the ROBDD builder to generate a visualization of the ROBDD using GraphViz dot[1], and the “title” is used to give the graph a title in the generated image. The following image is generated by this program:

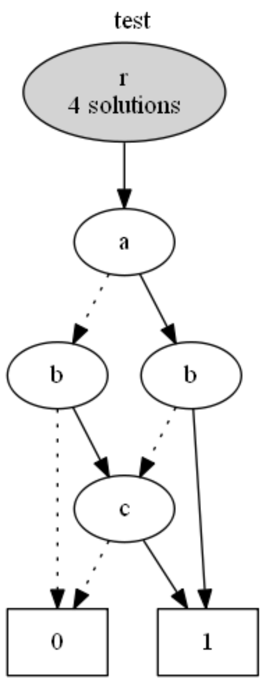


Figure 2: Output of the simple ROBDD program

[1] http://www.graphviz.org/

Multiple outputs are also supported. For example, consider the following extension to the simple program presented above:

```
title = 'test'
display = true

a = input.a
b = input.b
c = input.c
r1 = a*b + a*c + b*c
r2 = b*c

output.r1 = r1
output.r2 = r2
```

Figure 3: Program with multiple outputs

With this program, an ROBDD is generated that labels both outputs, and shares nodes between the two outputs. The output is shown below:

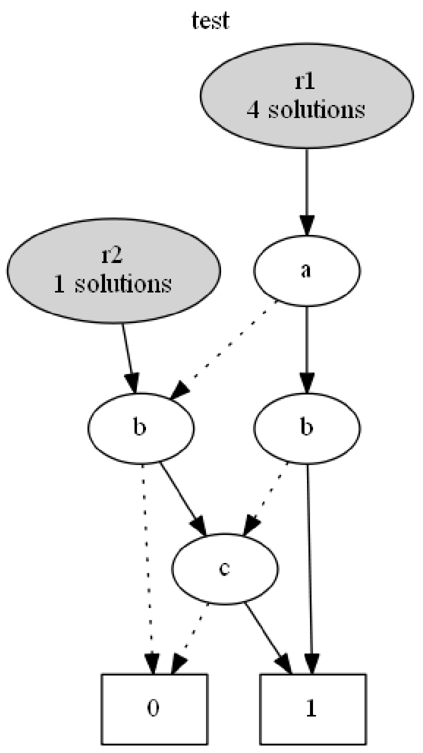


Figure 4: Result of the program with 2 outputs

From there, more interesting programs can be built. For example, here is the specification of a ripple-carry adder, with a configurable number of bits:

```
n = 2
display = true

title = tostring(n) .. '-bit ripple carry adder'

-- declare the inputs backwards to allow the circuit to reuse previous nodes
for i=n,1,-1 do
    _ = input['a' .. tostring(i - 1)]
    _ = input['b' .. tostring(i - 1)]
end

cin = input.cin

for i=1,n do
    a = input['a' .. tostring(i - 1)]
    b = input['b' .. tostring(i - 1)]

    cout = a * b + cin * (a ^ b)
    output['s' .. tostring(i - 1)] = a ^ b ^ cin

    cin = cout
end

output.cout = cin
```

Figure 5: ROBDD program specification for an n-bit ripple-carry adder

This program demonstrates many features of the scripting language interface. Namely:

- Input/output tables can be indexed by a string, to procedurally define inputs/outputs.
- Variable order is defined by their order of creation, for manual optimization.
- XOR is supported through Lua's pow operator.

Sample outputs for n=1 and n=2 are shown on the following pages of this document. Notice how the "circuits" for n=1 are reused in the computation of n=2.

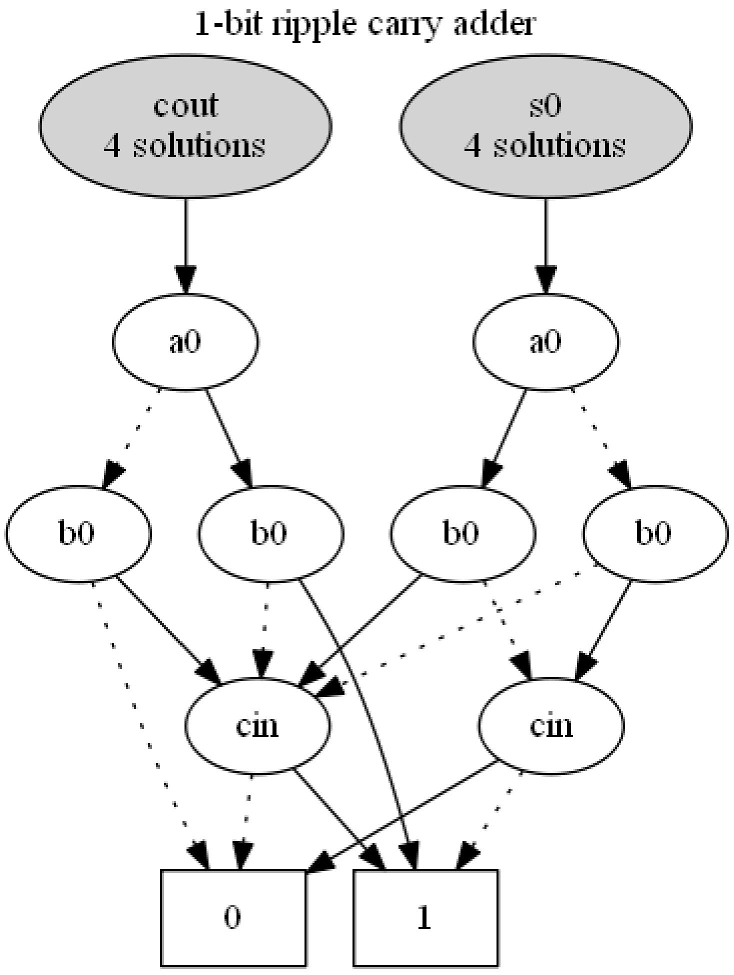


Figure 6: Sample output of the ripple-carry adder with n=1

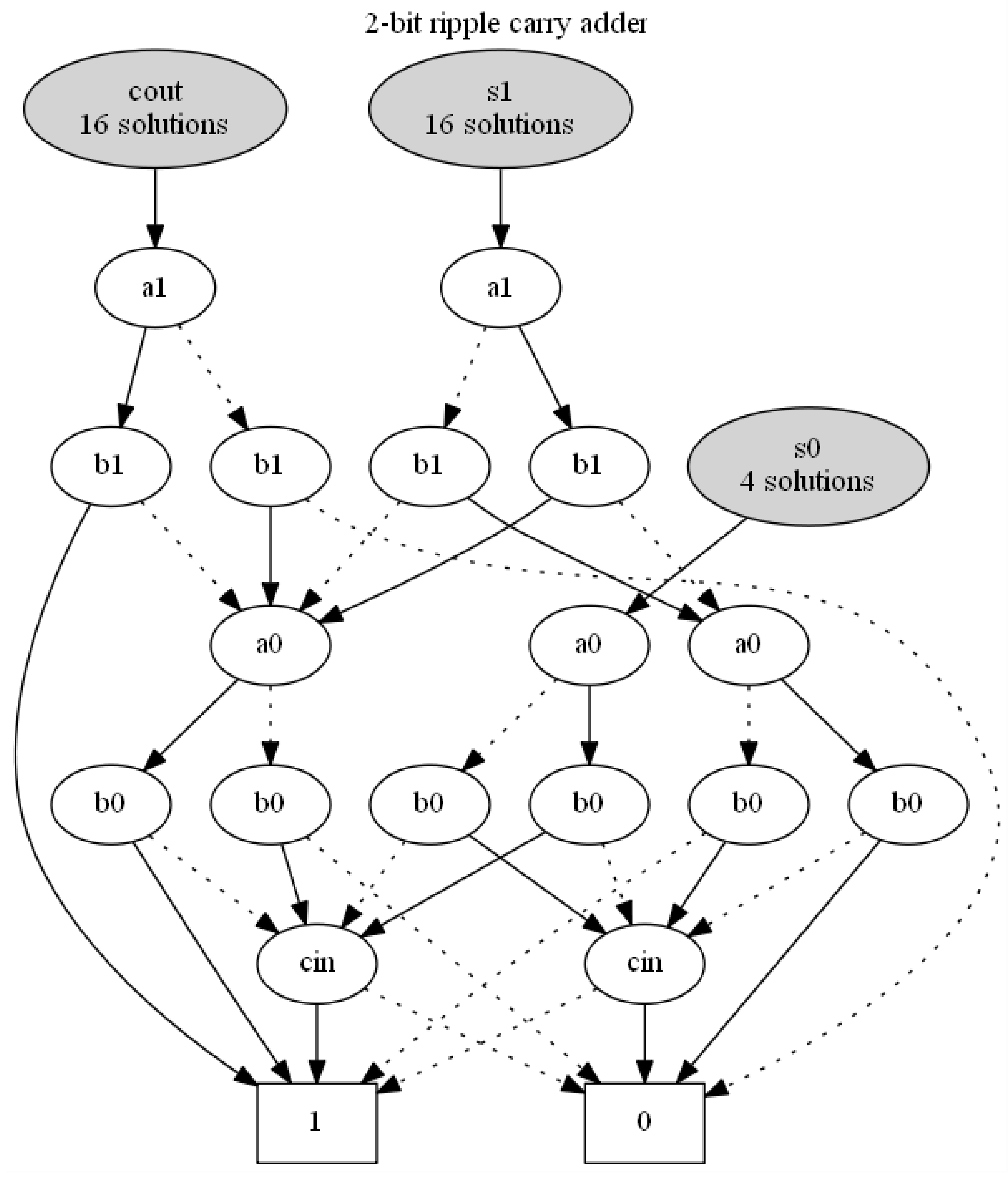


Figure 7: Sample output of the ripple-carry adder with n=2

An even more advanced program shown below solves the n-queens puzzle:

```
n = 12

display = n <= 6

title = tostring(n) .. '-queens puzzle’

function cellname(row, col)
    return 'r' .. tostring(row) .. 'c' .. tostring(col)
end

for row=1,n do for col=1,n do _ = input[cellname(row,col)] end end

board = true
for row=n,1,-1 do
    for col=n,1,-1 do
       T = false
       -- row
       for col2=col+1,n do T = T + input[cellname(row, col2)] end
       -- column
       for row2=row+1,n do T = T + input[cellname(row2, col)] end
       -- top left
       for col2=col-1,1,-1 do
           if row+(col-col2) > n then break end
           T = T + input[cellname(row+(col-col2), col2)]
       end
       -- top right
       for col2=col+1,n do
           if row+(col2-col) > n then break end
           T = T + input[cellname(row+(col2-col), col2)]
       end
       board = board * -(T * input[cellname(row, col)])
    end
    -- at least 1 queen
    T = false
    for col=n,1,-1 do T = T + input[cellname(row, col)] end
    board = board * T
end

output.board = board
```

Figure 8: Program to solve the n-queens puzzle

This program demonstrates 2 previously unmentioned features of the language:

- “true” and “false” as literals are supported, useful for writing loops.
- The unary minus operator is used to define NOT.

As a final example, modularity will be demonstrated. A “coloring.lua” module was created, which contains a function that builds a Boolean expression to define a graph coloring. This module currently only supports 3-coloring and 4-coloring.

Using this module, code was reused to solve three different graph coloring problems:

- 4-coloring of Japanese prefectures
- 4-coloring of the continental United States of America
- 3-coloring (and 4-coloring) of the Petersen graph

For example, the Petersen graph is solved as shown below:

```lua
local coloring = require 'coloring'

n = 3

title = tostring(n) .. '-colorings of the Petersen graph\'s vertices'

connections = {
     [1] = {2,5,7},
     [2] = {1,3,8},
     [3] = {2,4,9},
     [4] = {3,5,10},
     [5] = {1,4,6},
     [6] = {5,8,9},
     [7] = {1,9,10},
     [8] = {2,6,10},
     [9] = {3,6,7},
    [10] = {4,7,8}
}

output.coloring = coloring.color(connections, n)
```

Figure 9: Program computing the number of 3-colorings of the Petersen graph's vertices

Per this program, there exist 120 3-colorings and 12960 4-colorings of the Petersen graph, which is consistent with the results others have found online through mathematical methods[2].

[2] https://cameroncounts.wordpress.com/2012/04/12/counting-colourings-of-graphs/

For developers, the program also has some configurability at the C++ source code level:

- Define SHOW_INSTRS to print the generated bdd instructions (explained later.)
- Define SINGLETHREADED to run in a single thread (for testing purposes.)
- Define USE_APPLY_TASK to use a low-level TBB task construct instead of tbb::task_group.
- Define USE_TSX to make use of Intel Transactional Synchronization Extensions.
- Define BENCHMARK to output comma-separated timings with 1 to N threads.
- Define ITTPROFILE to enable Intel Instrumentation and Tracing Technology.
- robdd::unique_table::capacity defines the maximum number of unique table entries.
- robdd::computed_table::capacity defines the computed table's hash table size.

Overall Architecture

The program operates in three major phases:

1. The Lua program generates an Abstract Syntax Tree (AST) and a list of bdd instructions.
2. An instruction decoder reads each instruction, making calls to the robdd builder.
3. The ROBDD builder output is written to a dot graph file, and GraphViz dot is invoked on it.

Lua Program Execution

Before executing the Lua program, an "ast" Lua type is created, which overloads the operators as demonstrated previously in this document. Furthermore, a global "input" table is declared, used to allocate new AST node and ROBDD variable ID whenever a previously unseen input name indexes the table.

The result of indexing the input table and the result of any Boolean operators is a variable of type "ast", which stores the ID of the AST node that represents this expression. Although AST node IDs exist, the vertices and edges of the AST are not explicitly stored. Instead, the AST node IDs are used as the destination and source operands of BDD construction instructions that are created and appended to a list with every overloaded operator in Lua. These instructions are simple structs storing an opcode and operands, and their precedence is respected automatically by reusing Lua's own precedence of evaluation.

Note that a new AST node ID is generated with every operation, meaning the outputted instructions follow the design of Single Static Assignment (SSA) intermediate representation. SSA "phi" nodes are not supported, so representing control flow at the intermediate representation (IR) level is not possible. It is instead expected that the Lua program outputs a finite number of instructions.

At the end of execution of the Lua program, global variables are read, such as the list of outputs, the title of the dot graph, and whether to display the dot graph. For what it's worth, LuaJIT[3] is used as the implementation of Lua for this project.

[3] http://luajit.org/

ROBDD Instruction Decoding

Decoding the instructions generated by evaluating the Lua script is done with a simple linear fetch-decode-execute loop. As mentioned previously, control flow is not supported at the IR level, so fetching is a simple sequential scan of the list of instructions.

At present, there are five supported instructions:

- `dst_ast_id = NEW  var_id name`
- `dst_ast_id = AND  src1_ast_id  src2_ast_id`
- `dst_ast_id = OR   src1_ast_id  src2_ast_id`
- `dst_ast_id = XOR  src1_ast_id  src2_ast_id`
- `dst_ast_id = NOT  src_ast_id`

The NEW instruction invokes the ROBDD builder's `make_node()` function, to represent the new variable. Every other instruction invokes the ROBDD builder's `apply()` function, to incorporate the Boolean expression into the ROBDD.

The ROBDD builder works using BDD nodes, not AST nodes. Therefore, a lookup table is used to convert AST nodes into BDD nodes. This lookup table is used to convert the operands of instructions (AST node IDs) into operands for the ROBDD builder (BDD node IDs), and is implemented as a simple array that provides a 1-1 mapping (this is easy, since AST nodes increase linearly.)

While interpreting these instructions, the list of BDD node IDs that correspond to the AST nodes of the outputs are stored. These are used in the output of the graph and the counting of solutions.

ROBDD Graph Output

The instruction decoding pass outputs the list of ROBDD node IDs that correspond to the outputs of the Lua script. GraphViz dot nodes are created to represent the labels for these outputs, and the BDD nodes that are children of these outputs are outputted in a depth-first traversal using a stack. A previously created mapping of BDD variable IDs to string names (from the Lua input table) is used to give GraphViz nodes meaningful names.

ROBDD Builder Design

At a high-level, the ROBDD builder is designed in a relatively standard way:

- A unique table enforces the uniqueness of ROBDD nodes.
- A computed table caches results of evaluating ITE (for optimization purposes.)
- `make_node()` creates a new BDD node with a variable and lo/hi children
- `apply()` computes if-then-else recursively.

Of course, the devil is in the details.

Unique Table

The unique table implements an open addressed hash table used to guarantee the uniqueness of nodes in the ROBDD. It mainly supports a single operation: `insert(var, lo, hi)`. This function computes a hash from its inputs, looks for the node in the hash table, and creates a new node only if the node doesn't exist in the hash table.

Since this unique table is a shared resource between the threads building the ROBDD, it is necessary to enforce synchronized access to the unique table. To avoid expensive locking mechanisms, an interlocked Compare-And-Swap (CAS) operation is used to make insertions into the hash table.

When a node is not found in the hash table, a new node is to be created in the empty slot found at the end of the sequential scan of the hash table. To do this, a new node is created, its fields are filled, and it is conditionally inserted into the hash table using a CAS that succeeds only if the hash table location is still empty. This CAS can fail when a different node has been placed in this table location, or if the same node was placed in that location by a different thread.

When inserting a node fails, the insert operation resumes scanning the hash table from where it left, trying to find the next empty spot or the newly inserted node that is identical to itself.

One problem with this approach is that a failed insertion can leak an allocated node, technically requiring the node to be freed on exit of the function if it was allocated but turned out not to be needed (because another thread created the same node). Initially, a pool allocator was used, which supported this functionality. However, the added cost of pool allocation in memory bandwidth, and in the complexity of an interlocked implementation, made a plain linear allocator a favorable approach. Consequently, since freeing in arbitrary order is not possible for a linear allocator, failed insertions now simply leak the nodes they allocate. This is deemed a rare scenario, so hopefully not a big problem.

Computed Table

The computed table is a straightforward cache of the results of if-then-else. The fundamental problem to solve is that a cache entry should support N readers but only 1 writer. This was implemented by using one interlocked integer per cache entry. This integer represents the number of readers of each cache entry.

The cache is locked for writing by setting this integer to a number that exceeds the maximum number of readers using a CAS predicated on there being 0 current readers. When a reader wants to access the cache, it keeps trying to increment the integer until it receives a number less than the maximum number of allowed readers. These are 32-bit integers, so it's assumed that the cache entry will be accessed by readers before they collectively make the counter overflow.

The computed table supports an interesting optimization on very recent Intel processors, using the Transactional Synchronization Extensions (TSX) instruction set. This instruction set allows you to execute a block of code with no synchronization mechanisms whatsoever. If the processor finds that a data hazard happened during the transactional block, the CPU will go back in time to the start of the block and fail to enter the block. It sounds like magic, but it works. The magic trick is that computations are done entirely in cache. Therefore, if a conflict is found when the cache synchronizes, the cache is simply thrown away and the CPU's status registers are reset back to the start of the block.

There have been times during development where TSX blocks made a ~10% difference in performance. With such a lightweight readers/writer lock, it most likely does harm at this point. Regardless, it was an interesting experiment.

Apply

The main difference between the traditional implementation of apply() and this one is that its recursive calls are executed in parallel. This is done relatively elegantly using tbb::task_group, as follows:

```
tbb::task_group g;
node_handle lo, hi;

if (get_var(bdd1) == get_var(bdd2))
{
  g.run([&] { lo = apply(get_lo(bdd1), get_lo(bdd2), op, level + 1); });
  g.run_and_wait([&] { hi = apply(get_hi(bdd1), get_hi(bdd2), op, level + 1); });
  n = make_node(get_var(bdd1), lo, hi);
}
else if (get_var(bdd1) < get_var(bdd2))
{
  g.run([&] { lo = apply(get_lo(bdd1), bdd2, op, level + 1); });
  g.run_and_wait([&] { hi = apply(get_hi(bdd1), bdd2, op, level + 1); });
  n = make_node(get_var(bdd1), lo, hi);
}
else
{
  g.run([&] { lo = apply(bdd1, get_lo(bdd2), op, level + 1); });
  g.run_and_wait([&] { hi = apply(bdd1, get_hi(bdd2), op, level + 1); });
  n = make_node(get_var(bdd2), lo, hi);
}
```

Figure 10: fork/join parallel apply() using tbb::task_group

This code spawns two subtasks to run both recursive calls in parallel. The tasks are added to a work-stealing scheduler, which prevents oversubscription and allows dynamic load balancing.

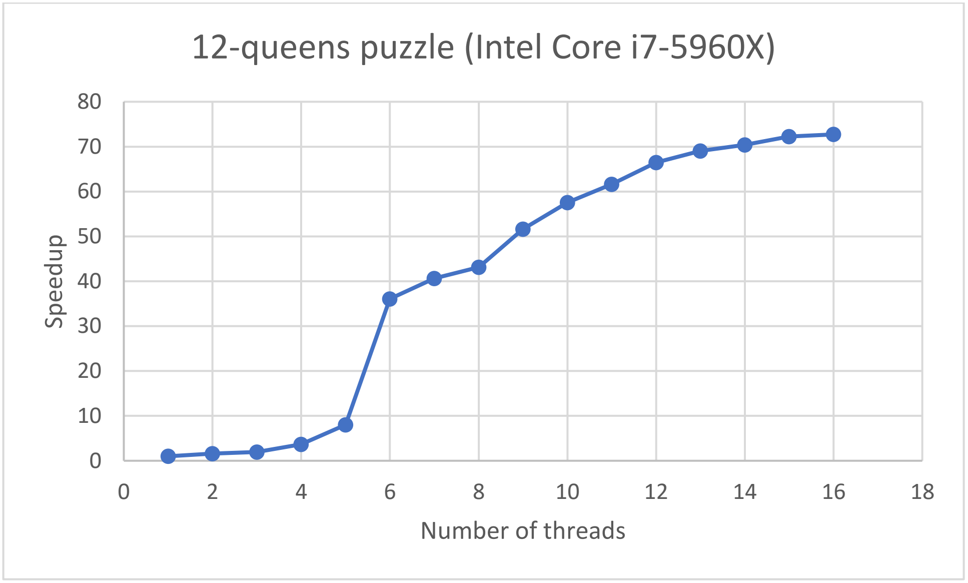
To avoid generating excessively small tasks, a limit on the number of levels of parallel recursion is set, using the "level" variable seen in the code above.

tbb::task_group is a relatively high-level task programming API that favors convenience over detailed control, so I made an alternate implementation of apply(), that uses TBB continuation tasks and task recycling. In theory, this should have performance benefits by reducing the number of task allocations, but the difference is not very noticeable.

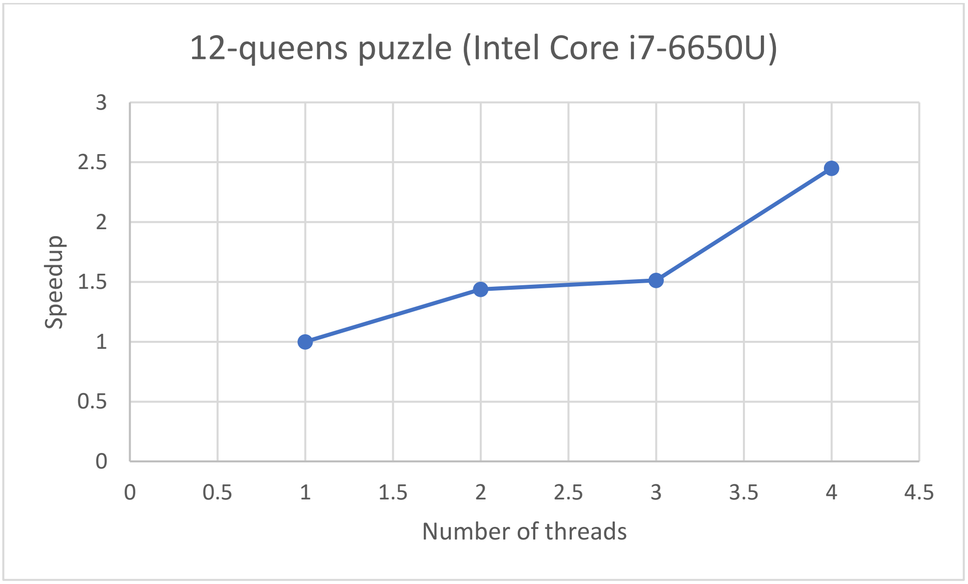
One of the major problems with the implementation is overhead from TBB's task scheduler. TBB is designed for relatively heavyweight tasks, but it's hard to predict when a recursive call to apply() will have a long execution time. With better predictions, work could be scheduled better.

Results & Conclusions

The 12-queens puzzle yields the following timings on an Intel Core i7-5960X (8 cores, 16 threads):



The 12-queens puzzle yields the following timings on an Intel Core i7-6650U (2 cores, 4 threads):



These results for i7-5960X are a little suspicious. For example, how is it possible that 16 threads give a speedup of over 70x from a single-threaded version? The sudden bump in speed from 5 to 6 threads is also strange. My guess is that the concurrent streams of execution happen to land on a much more efficient access pattern for the unique table or the computed table. If that's the case, then maybe it would pay off to think about more intelligent ways to schedule work, or ways to reorder variables, which would probably give great benefits even to a single-threaded implementation. The results on i7-6650U look more as expected.

Unfortunately, the other sample problems I've prepared finish too quickly for threading to have any significant impact. The exception is the 4-coloring of the USA, which I've never actually seen finish, though I'm guessing that either I have a bug in my data entry or I happened to pick an extremely pessimistic variable order in my data entry. The 4-coloring of Japan executes in a time like what was shown in the course lecture slides, though I get a different answer. It's unclear to me if it's due to a bug in my coloring code, or if our definitions of the connectivity of Japanese prefectures are different. My graph coloring code gives correct answers for the Petersen graphs, so my graph coloring code is not entirely wrong…

Thinking about a GPU implementation, it's uncertain to me how beneficial that would be. With a good choice of variable order, the execution is already so fast. When it comes to parallelizing this code, we might be better off implementing parallelism at a higher level, like building multiple ROBDDs for entirely different functions in parallel in multiple processes. Alternatively, we might need extremely large problems.

One of the biggest problems with a GPU implementation will probably be the overhead of transferring the input data to the GPU and reading back the output. Furthermore, I suspect that some custom scheduling of GPU cores will need to be in place, since the interpreter runs serially while the interpretation of each task will spawn additional jobs through fork/join parallelism.

To reduce memory transfer overhead, and to allow code tightly synchronized with a serial CPU command interpreter, I think an integrated GPU might be best suited for the job.

References

[1] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall and Y. Zhou, "Cilk: An Efficient Multithreaded Runtime System," *Journal of Parallel and Distributed Computing,* vol. 37, no. 1, pp. 55-69, 1999.

[2] Y. He, "Multicore-enabling a Binary Decision Diagram algorithm," 29 May 2009. [Online]. Available: https://software.intel.com/en-us/articles/multicore-enabling-a-binary-decision-diagram-algorithm. [Accessed 27 February 2017].