Lab 5: Backtracking Search

In this lab, you will implement a parallel backtracking search to solve games of peg solitaire. The write-up for this lab is due on Wednesday, April 7 at 5:00 P.M.

1 Peg Solitaire

Peg solitaire is a classic one-player game which involves removing pegs from a board with holes. Peg solitaire is played on a board which is an $n$-by-$m$ grid of squares, with each square being either valid or invalid. A valid square, or hole, may contain a peg. A hole is filled if it contains a peg, and it is empty otherwise. A configuration is a board with some subset of its holes filled. Thus, a board with $H$ holes has $2^H$ possible configurations.

In peg solitaire, a player transforms one board configuration into another via jumps. A jump involves 3 (vertically or horizontally) consecutive holes, with the first square identified as the starting square for the jump. More precisely, for a given board, one can represent a jump as a triple $(i,j,dir)$, where $i$ and $j$ are the row and column on the board of the jump’s starting hole (indexed from 0), and $dir$ is the direction of the jump (either NORTH, EAST, SOUTH, or WEST). A jump on a board is legal for a particular configuration if the jump involves 3 consecutive holes with the first and second holes filled and the third hole empty. A legal jump transforms the configuration by removing the two pegs from the first and second holes, and filling the third hole with a peg. For example, Figure 1(b) admits 3 legal jumps starting from square (2,2): (2,2,NORTH), (2,2,EAST), and (2,2.WEST). The jump (2,2,SOUTH) is never legal for any configuration of this board, however, because (4,3) is out of range.

The goal of peg solitaire is to find a sequence of legal jumps which transforms a starting configuration into a final configuration. In many games, the desired final configuration contains only a single peg. For example, in Figure 1, there exists a sequence of legal jumps which takes configuration (b) to configuration (f); in this case, (c), (d), and (e) represent the intermediate configurations for this sequence.

![Figure 1](image-url)

In this lab, you will write a program to solve peg solitaire: given a start and a final configuration, find a sequence of legal jumps connecting these configurations, or report if no such sequence exists. Since the problem of solving an arbitrary peg solitaire board is known to be NP-complete [3], a provably efficient polynomial-time algorithm for solving peg solitaire is unlikely to exist. Instead, in this lab you will implement a backtracking search algorithm, combined with some heuristics for pruning the search space.
2 Backtracking Search

One way to solve peg solitaire is to use a backtracking search algorithm, as shown in Figure 2. This algorithm searches for a solution from a given start configuration by making a legal jump \( J \) from start, after which it recursively searches for a solution from the new configuration. Since every jump reduces the number of pegs (filled holes) on the board by one, in the base case, the algorithm stops recursing when both start and final have the same number of pegs.

For this lab, we have provided code for a simple serial solver for peg solitaire. You can check out code using the following command:

```
git clone /afs/csail.mit.edu/proj/courses/6.884/spring10/labs/lab5/ lab5
```

This program takes as input the starting and final board configurations from a text file, and searches for a solution using an algorithm similar to the one in Figure 2.

(a) Construct some test inputs and benchmark the existing solver. For a fixed-size board, experiment and see if you can find the configurations which take the longest time to solve. For a
worst-case configuration, how large of an input board can the solver finish searching within 2 minutes?

The backtracking algorithm in Figure 2 is correct, but can be inefficient because it does not make any attempt to prune the search space while looking for a solution. Pruning the search space is particularly useful for peg solitaire problems which have no solution; a simple solver may need to check many configurations to verify that no solution exists. For example, the code we provide does not solve the peg solitaire board shown in Figure 3 in a reasonable amount of time.

```
--XX--       --oo--
--XX--       --oo--
XXXXXX       ooo000
XXoXXX       oooXoo
XXXXXX       ooo000
--XX--       --oo--
--XX--       --oo--
```

Figure 3: A peg solitaire problem on a 7-by-6 grid. This problem is solvable, but a simple solver without any heuristics takes a long time to find the solution.

Kiyomi and Matsui in [1] describe a heuristic for pruning the search space based on the number of times a particular jump can occur the solution sequence. Given particular start and final configurations, their heuristic is to compute an upper bound \( R(J) \) on the number of times the jump \( J = (i, j, \text{dir}) \) can appear in the solution. It turns out that this upper bound provides a useful way to prune the search space, since one need not search for solutions after a jump \( J \) occurs more than \( R(J) \) times. To compute this upper bound \( R(J) \), one can use a (relatively small) integer linear program. (See [1] for more details.)

(b) We have extended the simple peg solver to compute the upper bounds \( R(J) \) and use this quantity to prune the search space. Verify that this solver compiles and reports that the board shown in Figure 3 has a solution.

```
cagnode1:~$: make peg
cagnode1:~$: ./peg_tests/long_test1.in
```

To compute \( R(J) \), this solver uses the SYMPHONY library for mixed integer linear programming [2].

3 Optimizing the Solver

Your goal in this lab is to implement a peg solitaire solver which finds solutions as quickly as possible. Since different portions of the search space can be searched independently, conceptually, there is ample parallelism in a backtracking search that you can potentially exploit.

As written, however, the backtracking search in Figure 2 is serial, since it updates the start configuration and the current solution path in place when testing possible jumps \( J \). Thus, in a parallel implementation, one may need to make copies of the board state to avoid race conditions and expose the potential parallelism. If
one makes only a few copies, the parallelism of the algorithm may be limited and it may be difficult for the scheduler to efficiently load-balance the computation. On the other hand, repeatedly copying the board state introduces additional overhead as compared to the serial algorithm. One challenge for your implementation will be to find an appropriate trade-off between these options.

(c) Implement your own parallel peg solitaire solver which does not use Kiyomi and Matsui’s heuristic. You may wish to optimize the provided implementation to use a more space-efficient representation of board configurations.

How large of an input board can your parallel solver finish searching within 2 minutes? What is the parallel speedup of your implementation?

(d) Modify your parallel peg solitaire solver to use Kiyomi and Matsui’s heuristic to prune the search space. How large of an input board can the solver search within 2 minutes, assuming that the values of $R(J)$ for the problem have already been precomputed? What if one includes the time spent computing $R(J)$? What is the parallel speedup of your algorithm over the fastest serial solver you can implement?

Extend your solver by optimizing your implementation and finding solutions as quickly as possible. Some potential optimizations include:

- Accelerating the search by storing intermediate configurations in a hash table, and searching both forward from the start configuration and backwards from the final configuration.
- Speeding up the search by implementing other heuristics for pruning the search space. What is the impact of your heuristics on how easy it is to parallelize the search algorithm?
- Lazily computing the upper bounds $R(J)$ as needed, instead of relying on a precomputation step.

Be wary of unconsciously turning this assignment into your term project, however. You should spend only about 12 hours on this lab.

If you are interested in extending this lab into a term project, some ideas include:

- Parallelizing an integer linear solver such as SYMPHONY. In general, solvers rely on underlying branch-and-bound search algorithms which can themselves be parallelized.
- Parallelize another application which relies on a backtracking search (e.g., a SAT solver).
- Parallelizing a minmax searching algorithm for a game such as chess.

References


Lab5 Writeup — Parallel Backtracking Search

1 Summary of what we did

In lab5, we parallelize the backtracking search for the peg game. In the serial version, only a single board configuration and a single path log is used. In the parallel version, a single copy of these variables is no longer sufficient. Instead, each worker needs to keep track of its own local copy. Upon a steal, the worker makes a copy of the board configuration at the root, and replay the path log from the root down to the point where the steal occurs, in order to get the correct value for the board configuration. We thought that a more compact board representation may help the performance, since the more compact the board is, the faster it is to make a copy. Hence, we rewrote the design for the board configuration to use a more compact representation. In Section 2, we describe how we parallelize the search algorithm in more details. In Section 3, we describe our design of the compact board representation, including some additional design that we thought would improve the performance, but didn’t have time to finish implementing. Finally, in Section 4, we evaluate our design of the parallel algorithm and the board configuration.

2 How we parallelize the backtracking search

In the serial version, the search function recursively calls itself, upon making a move, to explore the next level of search tree with the new move. Only a single board configuration is needed throughout the computation: the code first makes a move $m$ on the board variable, $\text{board}$, logs the move it made (i.e., storing $m$ into the $\text{path}$ variable), recursively calls itself with the updated $\text{board}$ and $\text{path}$. The call returns whether a solution is found down the path with move $m$. If no solution is found, the code then undoes the move $m$, makes the next move, say $m'$, and calls itself to explore the next branch of the search tree associated with $m'$. One can think of the serial search algorithm as exploring the tree in depth-first order.

To parallel this search algorithm, we speculatively spawn off searches down the different branches. A “global” variable is used as a flag: once a solution is found in some branch, the flag is set to true; each worker periodically polls the flag (in this case, we simply check the flag at every spawn), and returns from the search if the flag is ever set to true.

In the parallel version, unlike the serial version, a single board configuration is no longer sufficient, because each of the workers exploring different branches of the search tree in parallel needs to have its own local copy of the board, reflecting moves associated with the current branch it is exploring. We solves this problem by creating an array of $P$ boards at the root. Whenever a worker resumes a stolen function (this is done via checking the $\text{current_worker_id}$ before and after a spawn), the worker copies over the value of the board configuration at the root of the search tree and replays the $\text{path}$ log associated with the stolen frame on the board. Doing so restore the board configuration to the appropriate value, reflecting all the moves made from the root down to the stolen frame. The thief then resumes the stolen frame with its own copy of the board configuration. Since no worker ever needs to look at another worker’s board configuration, having $P$ copies of the board is sufficient.

For the $\text{path}$ log variable on the other hand, upon a successful steal, the thief creates a new $\text{path}$ log, copies over the prefix of the $\text{path}$ associated with the stolen frame, and resumes the stolen frame with the new $\text{path}$. The worker has access to the correct $\text{path}$ log because the pointer to the $\text{path}$ and the length of
the path associated with the stolen frame are stored as local variables in the frame. In addition, if the victim is still updating the path, it is only updating the tail of the path, so the thief can safely read the prefix of the path without causing a race. The thief needs to allocated a new path, however, because upon a successful steal, the paths taken by the victim and the thief start to diverge.

Note that the strategy of keeping only $P$ copies of the path log (like what we do with the board) is not sufficient, because a thief resuming a stolen frame needs to look at the path log associated with the stolen frame. If only $P$ copies of the path is kept, the path changes upon a successful steal. If a thief is not able to resume the stolen work and copy over the old path before the victim performs its own steal, the path associated with the stolen frame may no longer be valid.

If we had a splitter implementation, the path could be declared as a splitter. Unfortunately, a splitter implementation does not exist in Cilk++ at the moment, so we have to manually simulate this behavior at the user code level. One thing we need to be careful about is when and how to recycle the memory, because we are creating a new copy of the path upon every successful steal, which must be recycled at some point. If we had a hook to the runtime system, a way to do it is to deallocate the victim’s copy of the path associated with the stolen frame when it returns from the spawn and realizes that the frame has been stolen, but we don’t really have such a runtime hook.

What we ended up doing is to have a dynamic array local to the frame (i.e. a local variable), which stores pointers to the newly allocated path due to a successful steal of this particular frame. The array is dynamic; when it is needed, it is initialized to size `current_worker_count`, then its size is doubled every time the size limit exceeds. This local array is then deallocated once the function instance is complete, because we know for sure that all the path logs can be safely deallocated at that point.

We considered implementing the path using a reducer, but we didn’t make it work before running out of time. This strategy involves declaring path variable as a reducer. Ideally, we like to initialize the newly created path with the prefix of the old path variable stored in the stolen frame, but we are not quite sure how to get the identity function of the reducer to do that. On the other hand, the reduce function doesn’t need to do much, and the runtime should just recycle the memory allocated for the additional path created upon successful steals, which would save us some trouble.

There are two versions of backtracking search: peg and peg_simple, where peg uses a heuristic to prune the search space, and peg_simple simply walks the entire search tree until a solution is found, or until the entire tree is explored (upon which point, it declares no valid solution). What we described so far is how we parallelized peg_simple. Parallelizing peg is very similar, except that it has additional move_table variable used to prune the search space. For the move_table variable, again, we simply allocate $P$ copies, one for each worker, and replay the log from the root upon a successful steal.

### 3 How we implemented the board

In order to improve the performance of our peg solver, we implemented a more compact representation of the peg board. The original implementation used an array of integers, where an integer represented each position on the board and could take on one of three values: invalid, empty, or filled. We replaced this representation with a more compact representation where each location on the board was represented using two bits. We then improved our design iteratively to take advantage of this new board design. Finally we determined how to use this design to take advantage of word-level parallelism, although time did not permit us to implement this final version.

The first improvement to the peg board involved representing each position using 2 bits instead of an entire integer. Because each board position could be either empty, filled (with a peg), or invalid, we needed
2 bits to represent all three possible states. We chose to represent an “invalid” position with 11, an “empty” position with 00, and a “filled” position with 01.

With this representation we realized that we could represent most large board sizes using a single 64-bit integer or 32-bit integer for each row. If a maximum board size we could feasibly search in 2 minutes was roughly $10 \times 10$, we could represent each row of this board in only 20 bits, so a 32-bit integer would be sufficient to represent an entire row. With this change we could represent the board using an array of $n$ 32-bit integers or an array of $m$ 32-bit integers, instead of an array of $n \times m$ integers.

We also noticed that we could represent positions and jumps in a more compact way. The original board representation used a struct storing an integer for the position and an integer for the direction of the jump. Because our boards were relatively small, however, we could use only 16 bits for the position, and use the additional space in a 32-bit integer to store 2 bits for the direction.

The appearance of a valid jump and the action of performing a valid jump could be represented more compactly in our new board representation. If the board were stored in a row-major fashion, we noticed that a west jump would have a bit pattern in some row of 000101, while an east jump would have the bit pattern 010100. These two patterns also applied for north and south jumps respectively in the column-major representation of the board. This meant that we could check the legality of a jump by $\text{xor}$-ing the appropriate bit pattern shifted by the appropriate number of bits in the correct board representation, masking off the rest of the bits in the row, and checking if this result equaled zero. To support efficient jump legality checking for all jumps, we decided to store and maintain both representations of the board, believing that the overhead of two compact representations would not be too significant.

Similarly, performing a valid jump could be represented as a simple bit operation on the correct board representation. In particular, performing a jump amounted to $\text{xor}$-ing the appropriate 6-bits in some row or column with the pattern 010101. For example, to perform a west jump at a valid location in the row-major board representation, we could shift the pattern 010101 to the correct location, $\text{xor}$ the row contents with this pattern, and save the result as the new row. This trick only worked with a board representation that matched the particular jump, however. To perform a jump in the other representation involved $\text{xor}$-ing the pattern 01 with the appropriate position in three different entries in the array. We implemented both the valid jump check and this operation for performing a jump in our peg solver, and found that we got notable speedup over the simple board improvement.

Finally we realized that we could more efficiently scan the board for all legal jumps, using word-level parallelism on the CPU. To see this, consider searching for all valid west jumps in some row $r$ of the board, represented in row-major order. We could perform all of these checks by doing the following. Let $b = 000101$ be the bit pattern for a west jump. First create the 32-bit word $w = b00b00b00b00$, which contains the bit pattern for a west jump 8 times in a row, each of which is padded by two 0’s. We $\text{xor}$ $r$ with this word and shift the result right by 2 bits to get $00x_100x_200x_300x_4$, where $x_i$ is the result of $\text{xor}$-ing $b$ with the corresponding triple of positions in $r$. If we add this result to 00111110011111011111011111011111, any $x_i$ that contains a 1 will result in a 1 in the adjacent 0 to its left, i.e. 00...01...0dots00.... If we mask out all the bits in this result except for the bits at positions $7 + 8j$ for $0 \leq j \leq 3$, we can compact these unmasked interesting bits into a nibble using a multiply and a shift. Note that each 0 in this nibble corresponds to some position in $r$ with a valid west jump. Since there are only $2^4 = 16$ possible values for this nibble, we can use this nibble to index into a lookup table of arrays of position results to get all of the positions in which a valid west move appears. We can then repeat these steps after shifting $w$ right by 1 bit, as long as we adjust the intermediate constants by the same shift and increment each position in the resultant array by the corresponding offset.

After 4 iterations of this procedure, we will have accumulated all of the positions in which a valid west
jump may exist. Because each iteration of this procedure requires a constant number of bit operations, we can process a row of the board for one kind of jump in constant time using word-level parallelism. We can do the same thing for the east, north, and south jump directions, assuming we maintain the appropriate board representations. The total time to compute all of these valid jumps on the board is therefore $O(n + m)$.

Furthermore we could parallelize the evaluation of each row and column to get a work-efficient computation with span $O(\lg(n + m))$, although the overheads of doing so could be too expensive. Unfortunately we were unable to try this word-level parallel evaluation of the legal jumps on the board due to time constraints.

4 Evaluation

While we parallelized both version similarly, we focus our experiments on peg_simple, because with the pruning enabled, it is difficult to come up with test cases where it runs for a long period of time, and therefore more difficult to gauge how much parallelizing the search helps.

To evaluate the parallelization, we created few additional test cases besides the ones provided, because the provided test cases are either too short for evaluation purpose (it runs so quickly that the cilkview outputs 0 seconds), or too long that they didn’t finish running in few minutes, so we got impatient and cut them off.

We have two test cases that have valid solutions, and two test cases that have no valid solution. We realize that, for the test cases with valid solutions, they ran for about the same amount of time, no matter how many processors we run them on, even though cilkview reported plenty of parallelism (shown in Figures 1 and 2). We speculate that, this is because there is only a single solution, and the solution is always found by the first worker (i.e., it’s in the first branch down the search tree). Thus, having more workers doesn’t necessarily allow the solution to be found faster; instead, all other additional workers are only doing busy work, exploring down the branches that won’t bear fruit. We confirm this speculation by reversing the order of the moves to explore, and as we speculated, the search didn’t finish within few minutes (and again, we cut it off because we are impatient). On the other hand, running test cases that have no valid solution indeed shows speedup (shown in Figures 3 and 4).
Figure 3: The cilkview output on parallel \texttt{peg\_simple} running with -\texttt{trials} all 8 and input of a peg configuration with no solution.

Figure 4: The cilkview output on parallel \texttt{peg\_simple} running with -\texttt{trials} all 8 and input of a peg configuration with no solution.

<table>
<thead>
<tr>
<th>tests</th>
<th>(T_{\text{serial}})</th>
<th>(T_1)</th>
<th>(T_2)</th>
<th>(T_4)</th>
<th>(T_8)</th>
</tr>
</thead>
<tbody>
<tr>
<td>valid 1</td>
<td>1.66</td>
<td>2.00</td>
<td>2.05</td>
<td>2.03</td>
<td>2.03</td>
</tr>
<tr>
<td>valid 2</td>
<td>6.77</td>
<td>8.15</td>
<td>9.36</td>
<td>8.14</td>
<td>8.15</td>
</tr>
<tr>
<td>invalid 1</td>
<td>41.90</td>
<td>56.87</td>
<td>30.19</td>
<td>15.43</td>
<td>7.85</td>
</tr>
<tr>
<td>invalid 2</td>
<td>18.07</td>
<td>23.73</td>
<td>12.11</td>
<td>6.35</td>
<td>3.26</td>
</tr>
</tbody>
</table>

Figure 5: The running times of the serial \texttt{peg\_simple} versus parallel \texttt{peg\_simple} running on one, two, four, and eight processors. All running times are shown in seconds, averaged over five runs.

<table>
<thead>
<tr>
<th>tests</th>
<th>(T_{\text{serial}})</th>
<th>(T_{\text{compact board}})</th>
<th>(T_{\text{simple board}})</th>
</tr>
</thead>
<tbody>
<tr>
<td>valid 1</td>
<td>2.03</td>
<td>2.03</td>
<td>1.40</td>
</tr>
<tr>
<td>valid 2</td>
<td>8.15</td>
<td>8.15</td>
<td>6.04</td>
</tr>
<tr>
<td>invalid 1</td>
<td>7.85</td>
<td>7.85</td>
<td>6.21</td>
</tr>
<tr>
<td>invalid 2</td>
<td>3.26</td>
<td>3.26</td>
<td>2.53</td>
</tr>
</tbody>
</table>

Figure 6: The running times of the parallel \texttt{peg\_simple} running on eight processors, using the compact peg board versus using the simple peg board. All running times are shown in seconds, averaged over five runs.

We are also curious to know, how our parallel version perform compared to the serial version (both uses the compact board representation). Not surprisingly, the serial version runs slightly faster compared to the parallel version running on single processor. With test cases that have valid solutions, no speedup was observed. With test cases that have no valid solution, near perfect linear speedup was observed compared to \(T_1\). The result is shown in Figure 5.

Finally, to see how much effort of making a compact board really paid off, we compare the running time of the parallel version using the compact peg board versus using the simple (provided) peg board. It turns out that, it didn’t make too big of difference, and using the compact board is actually slower (shown in Figure 6; ⚠️). We are not sure why. We wonder if we actually had implemented the board scanning with word level parallelism, if the compact board representation would win out. We ran out of time unfortunately.
1 Introduction

For this lab, I explored ways to optimize the runtime of the backtracking search, which was used to find solutions to various peg solitaire board configurations. My first step was to parallelize the backtracking search by all allowing jump possibilities for a given iteration to be explored in parallel. Next, I attempted to extend my parallelized algorithm to terminate more quickly by adding a global 'found' indicator, in an attempt to reduce the amount of work done by the algorithm. Finally, I added the Kiyomo search heuristic to my parallel algorithm to improve the runtime further.

2 Parallelizing Backtracking Search

To parallelize the backtracking search, I modified the algorithm so that for each iteration, all the possible jumps for all the positions on a given board were explored simultaneously. In order to do this, I parallelized both the loop for iterating through all the possible positions on the board, as well as the loop for iterating through all the possible jump directions for each position. By doing so, I significantly improved the runtime of the algorithm and achieved substantial parallelism.

Where the original serial implementation of the algorithm was performing $4 \cdot n \cdot m$ checks for the next move serially (since it would have to check 4 possible jump directions for each position in the n-by-m grid), my algorithm performed all of these checks at the same time. However, to prevent races my algorithm had to make copies of the board for each valid possible next move, as well as copies of the solution path. Although this increases the amount of work that must be done by the algorithm overall, in practice this copying step is only performed for a small fraction of the $4nm$ checked potential moves. The worst possible case for this algorithm to run on would be a situation in which, for each step, there were many ($\approx \Theta(4nm)$) valid jumps that could be made for that particular board configuration.

As can be seen from Columns 1 and 2 in Table 1, which show the runtimes for the original serial implementation and the new parallel implementation when run serially, the addition of the extra copying increases the total amount of work that the second algorithm has to do. Thus, the runtimes in the second column are slower than the ones shown in the first column.

To improve the runtime of my parallelized algorithm, I coarsened it to only perform all jump checks in parallel for the first several iterations. Using the number of jumps left (i.e. (end number of pegs - current number of pegs) ) as the coarsening threshold, I obtained the best runtimes for a coarsening value of 6. Graph 1 shows the speedup of the parallelized algorithm for this coarsening value. For 8 workers, my algorithm achieves a speedup of $\approx 4.5$. Below is the cilkview output for running my parallelized algorithm on test7, which is the longest no solution test in the run_tests test suite.
Column 3 in Table 1 shows the runtimes of the parallel algorithm for Tests 1-7, which seem to be noticeably faster only for the tests in which there is no solution. This makes sense because for cases where there is a solution, a serial algorithm will stop as soon as it finds the solution, which often does not require looking through the entire search space. The parallel algorithm, on the other hand, is not aware of which parallel spawned instances have found a solution and thus will look through all of them no matter how quickly the solution is found. But for test cases in which both algorithms must look through the entire search space, the parallel algorithm will outperform the serial one.

Statistics for Simple

1) Parallelism Profile
   - Work: 111254731673 instructions
   - Span: 81209 instructions
   - Burdened span: 818308 instructions
   - Parallelism: 1369980.32
   - Burdened parallelism: 135957.04
   - Number of spawns/syncs: 279930123
   - Average instructions / strand: 132
   - Strands along span: 78
   - Average instructions / strand on span: 1041
   - Total number of atomic instructions: 8
   - Frame count: 1609025875

2) Speedup Estimate
   - 2 procs: 2.00 - 2.00
   - 4 procs: 4.00 - 4.00
   - 8 procs: 8.00 - 8.00
   - 16 procs: 16.00 - 16.00
   - 32 procs: 31.99 - 32.00

3 Adding A Global 'Found' Variable

For my extension, I attempted to further optimize my parallel implementation by adding a global 'found' variable, which could be checked by all spawned instances of the search to determine whether they should keep going or whether they can terminate early. Any instance of the algorithm that found valid path to the solution, would set the global 'found' variable to true, and thus inform all other instances of the algorithm that they need not keep recursing to continue looking for a valid path. Although this kind of shared variable would create race conditions, they would be benign, since the shared variable would only ever be set to 'true'. My hope was that by doing this, I would be creating a way to prune the search space by preventing unnecessary additional overhead from spawns of iterations after the solution is found.

As can be seen from Columns 3 and 4 of Table 1, which show the runtimes for the (A) original parallelized search and (B) the parallelized search with the global found variable added, this modification to the algorithm does achieve faster runtimes for tests in which there is a solution (Tests
Adding Kiyomi Heuristic to the Parallelized Search

In terms of implementation, adding the Kiyomi heuristic to my parallelized backtracking search required additional copying work to be done, so that each spawned instance could store the current number of moves for each type of jump. Although this increased the amount of work done for each step of the algorithm, the addition of the heuristic significantly decreased the overall work by reducing the size of the searched subspace.

Adding the Kiyomi Heuristic significantly improved the runtime of the algorithm. Columns 5 and 6 in Table 1 show the runtimes for Tests 1-7 of the serial and parallelized versions of the backtracking search algorithm with the Kiyomi heuristic used a way to prune the search space. The first value in Column 5 represents the pre-computing times and the second value shows the runtimes of the rest of the algorithm. These values clearly indicate that the R(J) heuristic is very effective in pruning the search space, and is especially effective at lowering the runtimes for boards with no solution. Both for the parallelized and un-parallelized versions of the algorithm, the Kiyomo heuristic dramatically lowers the runtime. However, the cost of the pre-computing is demonstrably high, especially since it is a serial bottleneck for this algorithm and lowers the parallelism of the overall algorithm.
<table>
<thead>
<tr>
<th>Test</th>
<th>Original</th>
<th>Serial</th>
<th>Parallelized</th>
<th>Parallelized A</th>
<th>Parallelized B</th>
<th>Serial With R(J)</th>
<th>Parallel With R(J)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0.001</td>
<td>0.042</td>
<td>0.027369 + 0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>0.094</td>
<td>0.112</td>
<td>0.172</td>
<td>0.001</td>
<td>0.637891 + 0.195</td>
<td>0.005</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>0.105</td>
<td>0.122</td>
<td>0.235</td>
<td>0.011</td>
<td>0.683297 + 0.109</td>
<td>0.003</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>0.02</td>
<td>0.014</td>
<td>0.014</td>
<td>0.042</td>
<td>0.292368 + 0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>0.056</td>
<td>0.036</td>
<td>0.101</td>
<td>0.013</td>
<td>0.225339 + 0.118</td>
<td>0.003</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>2.085</td>
<td>2.292</td>
<td>1.306</td>
<td>1.608</td>
<td>0.345477 + 0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>7</td>
<td>12.819</td>
<td>13.084</td>
<td>8.049</td>
<td>9.01</td>
<td>0.033266 + 0</td>
<td>0.025</td>
<td></td>
</tr>
</tbody>
</table>

Table 1: Runtimes for Tests 1-7

5 Further Work

With more time, I would have liked to expand my analysis data set and have run the the algorithm on larger size test cases. Unfortunately, due to some cagnode and X windows errors and the resulting time constraints, my exploration of the effectiveness of this algorithm was limited to smaller test cases, which are not necessarily as effective at demonstrating the scope of the algorithm. I would also have liked to explore more the difference in runtimes for larger test cases with and without solutions, and the heuristics and strategies that are most effective in yielding shorter runtimes for each of these two scenarios.
Lab5: A failure to implement an interesting extension

For this lab I experimented with optimizing the peg solitaire search algorithm by avoiding unnecessary cilk_spawn, tuning the base case, and searching both forwards and backwards simultaneously. I ran all my experiments on problems that took the original peg solitaire algorithm with pruning between 60 seconds and 120 seconds to search. Figure 1 shows the basic search algorithm without optimizations.

**Unnecessary cilk_spawns:** My initial implementation spawned SearchJump without checking if the jump was legal according to the heuristic move table or the current board configuration. For long searches cilkview reported that this algorithm had parallelism in the tens and hundreds, but a burdened parallelism less than 1. The performance of this algorithm was typically several times slower than the serial version. Checking if a jump was legal before spawning SearchJump reduced the parallelism reported by cilkview for long searches to 1, but the burdened parallelism was within a factor of two of the parallelism.

**Understanding base case tuning:** Tuning the base case based on the path length improved performance by about 20%. Initially I thought the cost of copying data structures so spawns to SearchJump could execute in parallel would dominate the cost of cilk_spawn, however this was not the case. The performance of running with a single worker that copied the data structures before each cilk_spawn was only about 5% slower than the performance of running with a single worker that never copied the data structures. Therefore the performance improvement from tuning the base case was the result of decreasing the burden of cilk_spawn statements.

I found that a base case path length (BASE_LENGTH in Figure 1) of 5 worked well for many searches.

**Forward and backward search:** I implemented an algorithm that simultaneously searched forwards from the initial configuration and backward from the final configuration. I found that searching backwards often outperforms searching forwards, sometimes by a as much as 100x. One possible reason for this is that for a board with a final configuration of one peg the starting branching factor of the backward search will generally be smaller than the starting branching factor of the forward search. This might result in fewer total paths to search.

**Miscellaneous:** I like the backtracking search problem and would be interested in investigating applications besides peg solitaire. I think a more interesting application might be one where the intermediate data structures, such as the board configuration in peg solitaire, are large, so copying them would dominate performance.
// Make a copy of startBoard, path, and moveTable, then search for a
// solution starting with a jump from pos in direction.
void searchJump(direction, pos, startBoard, stopBoard,
    path, pathLength, moveTable, sol);

// Same as searchJump, except don't copy startBoard, path, or
// moveTable
void searchJumpNoCopy(direction, pos, startBoard, stopBoard,
    path, pathLength, moveTable, sol);

void find_path(startBoard, stopBoard, path, pathLength, moveTable, sol)
{
    if (path_length <= 0) {
        if (*startBoard == *stopBoard)
            sol->pathIs(path); // found a solution
            return;
    } else if (path_length <= BASE_LENGTH) {
        for (Pos p = 0; p < start->getN() * start->getM(); p++) {
            searchJumpNoCopy(NORTH, p, startBoard, stopBoard, path,
                pathLength, moveTable, sol);
                if (sol->valid())
                    return;

            searchJumpNoCopy(EAST, p, startBoard, stopBoard, path,
                pathLength, moveTable, sol);
                if (sol->valid())
                    return;

            searchJumpNoCopy(SOUTH, p, startBoard, stopBoard, path,
                pathLength, moveTable, sol);
                if (sol->valid())
                    return;

            searchJumpNoCopy(WEST, p, startBoard, stopBoard, path,
                pathLength, moveTable, sol);
                if (sol->valid())
                    return;
        }
    } else {
        for (Pos p = 0; p < start->getN() * start->getM(); p++) {
            cilk_spawn searchJump(NORTH, p, startBoard, stopBoard, path,
                pathLength, moveTable, sol);
                if (sol->valid())
                    return;

            cilk Spawn searchJump(EAST, p, startBoard, stopBoard, path,
                pathLength, moveTable, sol);
                if (sol->valid())
                    return;

            cilk Spawn searchJump(SOUTH, p, startBoard, stopBoard, path,
                pathLength, moveTable, sol);
                if (sol->valid())
                    return;

            cilk Spawn searchJump(WEST, p, startBoard, stopBoard, path,
                pathLength, moveTable, sol);
                if (sol->valid())
                    return;
        }
    }
}

Figure 1: The pseudo code for peg solitaire backtracking search. BASE_LENGTH is the tuning factor for the base case. For the unoptimized algorithm BASE_LENGTH is 0 and for the optimized algorithm it is 5.
For this lab, implementing backtracking search, due to difficulties debugging I hit the 12-hour limit well before being finished. That said, I really wanted to finish this, so I kept going and after probably 20 hours had a parallelized backtracking search that works pretty quickly, cutting some of the test cases down to finding solutions in $1/1000$ their original runtime. This isn’t to say, either, that this lab was particularly difficult, I just had trouble with debugging which I will explain later.

1 Peg Configurations

I first examined the peg problems using the provided serial algorithm to determine which configurations took the greatest amount of time to solve. Obviously, the configurations with the greatest running times are those which are impossible to solve, as the algorithm must test every possible series of jumps. Looking only at solvable configurations, those with the largest path would seem to take more time as there are more possible jumps the algorithm runs through before terminating any given series of jumps. This is also why the heuristic for putting bounds on the number of a particular jump is useful - it cuts down on the total number of jumps tried by the algorithm. So, to maximize time spent, starting configurations with many pegs and a stop configuration with only one should be used.

Time taken by the algorithm also increases with the number of jumps available to any given peg. This can be increased, at least in the starting configuration, by having an open space along the perimeter of the pegs. Each of these spaces added provides at least one extra jump to be explored.

The provided `longtest1` example, also shown in the lab assignment, is a good example of a configuration with many pegs. I created a configuration taking more than two minutes to run based off this by adding more spaces for jumps - a single column of holes added to the right of the existing configuration was enough to make the configuration take much longer to solve.

2 Optimizations

In this section, I look at the two optimizations - path bounding and parallelizing - which I looked into.

2.1 Path Bounding

Rather than have the serial algorithm test all possible jumps, it is possible to put an upper bound on the number of times any given jump could be taken in a solution path. Kiyomi and Matsui’s heuristic was provided in an implementation similar to the previous serial solution. Using this heuristic, even including the time taken for precomputation, the time used by the serial algorithm was cut by $1/3$ for the `longtest1` example and the modified form with an extra column of holes.

2.2 Parallelizing

Parallelizing this algorithm was interesting because, as the lab points out, we must balance smart division of workload with the cost of constantly copying the board object for each subdivision. Cilk has seemed to perform very well with workload balancing so I was inclined to divide the work as much as possible, perhaps spawning a new thread at every recursion, so everything would get done. However, I’ve also seen previously how copying data structures is very expensive. So I decided to go with a coarse-grained approach, doing all of the divisions right at the start and none over the course of the rest of the algorithm.

To do this, I modified the `find_path` function to, instead of simply calling a helper, start a new thread for the configuration after every possible move in the start configuration. This was done with a `cilk_for` loop and
creating a new copy of the board configuration in each iteration of the loop so that there is no race between the threads. Doing this proved to be more difficult than I expected due to some C++ quirks - allocating space in an array of board configurations did not seem to work (while simply creating a new board with a given configuration within the loops was just fine). After much debugging I realized that the board which I thought was being created was not actually being made, and had a few holes from the very beginning missing. Once this was determined I was able to give up on having an array and just built a new board variable.

One downside to my approach is that the size of work units is very large. This means that the running time could be as long as the deepest path past the first jump. Even if a solution path is found by one thread, the whole program must wait for all others to finish. To get around this, I created a global variable to indicate that a path was found and act as a basic inter-process communications mechanism. Using this and checking for the ‘found’ bit to be set at each recursion allows the program to finish as soon as any thread finds a solution path.

Another downside is the general coarseness of my algorithm’s work units. My approach was done entirely to avoid excessive copying and, in fact, puts all of what little copying work there is into the very start of the algorithm. This coarseness also means, though, that only one thread may be running in a worst-case setup. In fact, one could easily produce a start configuration with only one valid move, and my parallel algorithm would perform exactly the same as a serial one.

2.3 Results

It is difficult to quantize the speedup of my parallel algorithm as it depends entirely on the start and stop configuration being used. The provided longtest1 example ran in approximately \( \frac{1}{1000} \) its serial time when using my parallel version. However, there is no way that this is indicative of my speedup. Rather, a solution to that problem happened to be at the start of one thread’s assigned work. My speed-up and efficiency, then, may be better-measured based on the data.

The speed-up of my program is linear but only up to the minimum of a) the number of processors available and b) the number of legal moves available from the start configuration. Still, a linear speedup is very good. How is this practical? It goes with the model I use - I merely split the work up into the same individual runs which the serial algorithm goes through in order. Because this is a basic serial algorithm, there is no information shared or strategies changed after seeing some portion of the data, so running each portion on its own does not lose any efficiency. I also suffer almost no efficiency loss because the only extra steps are copying the start configuration some number of times equal to the number of valid start moves and a small boolean check at each recursion to see if some other portion of the algorithm has already returned.
Backtracking Search

Summary

In this lab, I consider parallelizing a backtracking search to solve a simple peg solitaire game. The general algorithm and approach is generally applicable to a variety of problems, including SAT solving, depth-first-search, graph colorability, etc. The primary algorithm is simple, but not entirely efficient due to the need to copy data. I consider several variants of the implementation and analyze their performance characteristics.

- Section 1 describes the serial version of the problem.
- Section 2 describes the simple parallel implementation and two attempts to reduce extraneous copies.
- Section 3 describes a modification to the board structure to decrease its memory footprint in an attempt to reduce the burden of copies.

1 The Serial Implementations

The serial solver implements a simple depth-first search of the board graph. Given a board with moves left to make, the algorithm makes each legal move and recurses on the resulting board. If a solution is found, it returns a flag stating that a solution was found and further recursion is stopped.

The algorithm works reasonably well but can be improved, especially for unsolvable boards, by adding a pruning heuristic. The heuristic pre-computes the maximum number of jumps that can be made from a given location using linear programming. Using this, before recursing on a particular jump, the search checks to ensure that the jump can lead to a solution, if not, the recursion is skipped.

Figure 2 gives the performance of the serial algorithms solving the problem given in Figure 1.

```
--XX--  --XXXX--  --oXo--  --XX--
--XX--  --XXXX--  --XXX--  -oXX--
XXoXXX  XXXXXoXXX  oXXXXXo  XXXXXX
XXXXX  XXXXXXX  oXXXXXo  XXXXX
--XX--  --XXX--  --XXX--  -oXX--
--XX--  --XXX--  --oXo--  --XX--
```

Problem #1  Problem #2  Problem #3  Problem #4 (unsolvable)

Figure 1: The problems used for benchmarking. (X = peg, o = hole, = invalid)

<table>
<thead>
<tr>
<th>Variant</th>
<th>Problem #1</th>
<th>Problem #2</th>
<th>Problem #3</th>
</tr>
</thead>
<tbody>
<tr>
<td>simple</td>
<td>29.588</td>
<td>19.509</td>
<td>13.023</td>
</tr>
<tr>
<td>heuristic</td>
<td>12.857</td>
<td>12.429</td>
<td>9.794</td>
</tr>
</tbody>
</table>

Figure 2: Serial performance (running on an 8 core cagnode machine).

The timings show a considerable improvement from using the heuristic. The improvement is even more apparent on even larger boards and especially boards that are not solvable. For example, Problem #4 isn’t solvable but the simple

---

1Example:http://www.woodentoys-uk.co.uk/Peggy/wooden-games10.html
variant analyzes more than 25 billion moves in order to conclude this (even then, I wasn’t able to run it to completion). On the other hand, the constraints that the heuristic solver analyzes prove immediately that it isn’t possible and so the search part of the process takes no time at all (solving for the constraints takes 3.96132s).

## 2 Parallelizing and the Copy Problem

The potential parallelism of the algorithm is considerable since all of the branches of the search tree can be searched in parallel. In order to exploit this parallelism, it is necessary to give each worker its own copy of the board. In addition to this, I add a global abort flag that is set to `true` as soon as a solution is found and causes other workers to return immediately.\(^2\) I compare three variants for addressing this:

- **pure** naïvely copies the board for every recursive call to the solver (`par_solve_pure`).
- **purepre** constructs all of the child boards *in serial* and then recursively calls the solver on them in parallel (`par_solve_purepre`).
- **batch-\(n\)** batches \(n\) candidate boards together and runs them serially (`par_solve_batch`).

The **purepre** variant aims to divide the work more evenly amongst the spawned workers since, in most cases, legal jumps occur in groups. This causes an uneven amount of work to be passed into the children. The **batch-\(n\)** variant further refines this idea by avoiding creating a new board for each jump and only copies the board once for each batch. This forces the batches to be run serially, but avoids some of the copy overhead. Figure 3 gives the performance of these variants.

<table>
<thead>
<tr>
<th>Variant</th>
<th>Problem #1</th>
<th>Problem #2</th>
<th>Problem #3</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Time (s)</td>
<td>Speedup</td>
<td>Time (s)</td>
</tr>
<tr>
<td>pure</td>
<td>0.003</td>
<td>9940x</td>
<td>0.641</td>
</tr>
<tr>
<td>purepre</td>
<td>0.001</td>
<td>29821x</td>
<td>0.138</td>
</tr>
</tbody>
</table>

Figure 3: Parallel performance on the problems from Figure 1 (running on an 8 core cagnode machine).

The speedups seem unnaturally high and vary considerably, but are reproducible. This is probably due to the serial implementation getting stuck on an incorrect early move and being forced to search a large tree before it backtracks enough to make a move that has a solution. This suggests that the performance gained by parallelism is unlikely to be additive when combined with the constraints heuristic.

My implementation of the batch algorithm seems to have a memory leak so results are not reported, but from looking at the **purepre** variant I hypothesize that the **batch-\(n\)** variant would not be very different. Many boards, especially at the very beginning, have very few jumps which end up fitting within any reasonable grain size. In fact, `top` often reported low processor utilization (25-50% of the 8 cores\(^3\)) when setting the grain size much higher than 2.

Running the code under `cilkview` reports perfect linear speedup for both the **pure** and **purepre** variants. This isn’t surprising because, as previously mentioned, the branches of the tree have no interdependencies.

Looking at the times for different worker counts (given in Figure 4) gives an interesting picture of the variability of the search algorithm. I run the code on different numbers of processors from 1 to 8. The timings suggest a plateau-like pattern where running on few processors takes a considerable amount of time. As more workers are added, the time fluctuates a fair amount, but then drops off at some point and doesn’t change much after that. I conjecture that this is due to finding a point where the algorithm “gets lucky” which occurs at a later jump in the sequence. The drastic decrease occurs when a spawned worker attempts this point early.

The other interesting thing is that doubling the amount of workers from 4 to 8 actually increases the time to solve the problem. The amount isn’t very large, but it is possible that this is due to the overhead of needing to abort the remaining workers before returning.

---

\(^2\) This is implemented by polling so some extraneous work will be done before the abort takes effect.

\(^3\) `top` was reporting 200-400%.
3 Compressing the Representation

I wanted to see if a more space-efficient representation of the board would mitigate the copy costs that are incurred by the parallel implementation. Since there are 3 possible board states, a basic compression would require 2 bits per cell, this is a 4x compression over the SimpleBoard implementation. I can improve this compression by noting that no board operations in the search affect whether a cell is invalid. Using this, I factor out a “prototype” representation that encodes the validity of cells in a single bit and pair this with a peg representation (called mask in the implementation) which encodes the locations of pegs using 1 bit per cell. Using this style (called BitBoard in the implementation), speculating a move on a board only requires a copy of 1-bit per cell.

Based on the hypothesis that the parallel performance loss is due to copying, it would seem that this compression would provide a considerable performance improvement, however, this doesn’t seem to be the case. The performance numbers are given in Figure 5

<table>
<thead>
<tr>
<th>Problem</th>
<th>Time in Seconds on # of Workers</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
</tr>
<tr>
<td>#1</td>
<td>108.72</td>
</tr>
<tr>
<td>#2</td>
<td>70.65</td>
</tr>
<tr>
<td>#3</td>
<td>46.867</td>
</tr>
</tbody>
</table>

Figure 4: Time required for solving the benchmark problems on different numbers of workers.

The results suggest that that the benefit of board compression is marginal. The problem is that, while copies are cheaper, the cost to access the data structure at the bit level is considerably greater. I measure the overhead of the bit operations by running the serial implementation with the SimpleBoard and the BitBoard, the results are given in Figure 6. Note that given these overheads since we are roughly breaking even even when running the parallel version, the speedup of the compression to copies is approximately 1.34x. This is considerably smaller than the 8x that the theoretical analysis, but can partially be contributed to the fact that the boards are padded to int sized blocks so for a 7x6 board the representation pays an additional 22 bits of overhead which is roughly 33% of the memory.

<table>
<thead>
<tr>
<th>Problem</th>
<th>Uncompressed Time (s)</th>
<th>Compressed Time (s)</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>#1</td>
<td>0.003</td>
<td>0.003</td>
<td>1.00x</td>
</tr>
<tr>
<td>#2</td>
<td>0.585</td>
<td>0.551</td>
<td>1.06x</td>
</tr>
<tr>
<td>#3</td>
<td>30.086</td>
<td>26.235</td>
<td>1.15x</td>
</tr>
</tbody>
</table>

Figure 5: comparison of simple and compressed board on the problems from Figure 1 (running on an 8 core cagnode machine).

<table>
<thead>
<tr>
<th>Problem</th>
<th>Uncompressed Time (s)</th>
<th>Compressed Time (s)</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>#1</td>
<td>29.605</td>
<td>38.883</td>
<td>0.761x</td>
</tr>
<tr>
<td>#2</td>
<td>20.392</td>
<td>25.055</td>
<td>0.813x</td>
</tr>
<tr>
<td>#3</td>
<td>12.048</td>
<td>18.085</td>
<td>0.666x</td>
</tr>
</tbody>
</table>

Figure 6: The overhead of the more complex board operations of the BitBoard representation for the problems from.
Prune Juice: Searching, Backtracking, and Compact Representations

1 Searching in Parallel

I implemented four different parallel backtracking search algorithms to solve the peg solitaire problem. The three main ideas analyzed are: representation compactness, pruning, and guided search. While it is the case that search provides ample parallelism, getting better performance requires thought. I implemented two different compact representations of a peg solitaire board (algorithms `peg`\textit{compact} and `peg`\textit{parse}). Furthermore, I implemented two heuristics: one which pruned board configurations which can be easily check unsolvable, and another which simply prioritized the ordering of the search. Table 1 shows the performance numbers for the different approaches. The representation was by far the most effective way of increasing performance tested (with the `peg`\textit{compact} algorithm being the fastest).

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Test Input Time (sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0</td>
</tr>
<tr>
<td><code>peg</code>\textit{compact}</td>
<td>0</td>
</tr>
<tr>
<td><code>peg</code>\textit{parse}</td>
<td>0</td>
</tr>
<tr>
<td><code>peg</code>\textit{parse}+prune</td>
<td>0</td>
</tr>
<tr>
<td><code>peg</code>\textit{serial}</td>
<td>0</td>
</tr>
</tbody>
</table>

Figure 1: Timing results for algorithms on different test inputs

2 Compact Board Representations

The original `peg`\textit{serial} algorithm represents a board configuration as an \( n \times m \) array of integers. Copying this array (something which must be done when parallelizing the search) is costly. I tested two different compact representations for a board: `peg`\textit{compact} and `peg`\textit{parse}.

The `peg`\textit{parse} algorithm represents a board as two sets of position values: one stores positions of invalid locations, the other stores positions of pegs. This means that the number of positions stored per board is just the sum of the number of pegs and the number of invalid locations. The number of invalid locations does no change within the search, but the number of pegs can only decrease, so boards deeper down the search tree take use less space in memory. Of course, having such a representation makes access slower. We can see from the timing information that the `peg`\textit{parse} algorithm is faster that the original `peg`\textit{serial}, so any speed-downs because of access were made up by the gains because of parallelism.

An even more compact board representation was tested in the `peg`\textit{compact} algorithm. Here, note that we only need two bits of information for every location of the board (EMPTY, INVALID, or FILLED). I implemented a bitwise board which stored an array of integers and packed the board position states into these integers. For all of the boards tested only two integers are required to represent the state of the board. Accessing the elements of this compact bit representation is not that slow (just a few extra bitwise operations) and copying boards is extremely fast. This is the best optimization strategy tested: having a small, fast, compact board representation provided the best speedup.

3 Guided Search

I analyzed several heuristics to prioritize the order in which the search was conducted. The most promising was to give priority to moves which kept pegs together since separated pegs are difficult to utilize. I also tried prioritizing those moves which were from the edge of a set of pegs (or not). None of these heuristics offered any real performance
benefit. This is particularly the case since several of the test input cases are unsolvable problems, in which case such guided search heuristics still have to search the entire space and you get a performance hit for calculating the heuristic in the first place.

4 Pruning

I developed a pruning heuristic for the peg solitaire game and implemented the pruning in the peg\textsubscript{parse}+\textsubscript{prune} algorithm. The heuristic quickly checks whether a board configuration is unsolvable by trying to find two consecutive empty rows or columns that separate the connected components of the pegs. The rules of the game dictate that a single peg (or connected peg set) cannot somehow affect the state of the board two rows (or cols) away unless there are other pegs in those rows (or cols). This is because pegs essentially use each other to ‘hop’ and if there are no pegs that one cannot build a hopping sequence to bridge the two sections. As such, my heuristic checks for a slightly weaker condition which is: does there exist two empty rows (or columns) with pegs at either side. If these empty rows (or cols) exist, then the board is not solvable from that configuration so we can stop the search and backtrack. Table 2 shows the number of boards (the path lengths) pruned with this heuristic above and beyond those pruned by the Kiyomi-Matsui heuristic. The path lengths pruned is the number of path lengths (depth) we did not search because a board was pruned. While the heuristics successfully reduced the search space, computing the heuristics seems to outweigh the pruning value given that the peg\textsubscript{parse}+\textsubscript{prune} algorithm performed slower than the peg\textsubscript{parse}. My intuition is that a highly optimized heuristic checker (which this implementation is not) will show performance improvements in larger test cases than those studied.

<table>
<thead>
<tr>
<th>Test</th>
<th>Boards Pruned</th>
<th>Path Length Pruned</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>117</td>
<td>4964</td>
</tr>
<tr>
<td>3</td>
<td>855</td>
<td>3982</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>558</td>
<td>2415</td>
</tr>
<tr>
<td>6</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>7</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Figure 2: Pruned Boards (and Path Lengths) Beyond those of Kiyomi-Matsui
1 Approach

My approach for this lab was to build a data structure that would store the relevant state, and copy it on demand when threads compete for the data. My original plan was to do this without any knowledge of the active processor. Essentially, each function would first try to grab a lock on the data. If it was unable to grab the lock, then it would copy the data and start its own computation.

I had trouble getting this design working, however, but perhaps it would be possible to solve the problems I ran into. One problem is that if you are unable to grab the lock, then there is a data race in copying the data. Another problem is that the calling function holds the lock (and must in order to keep its data consistent), so the spawned functions will always copy the data. This wasn’t the on-demand behavior that I wanted.

Instead, I implemented a “Cilk++-aware” data structure. The data structure provides acquire and release functions by reference counting. If someone holds a reference to the data structure, then it is “owned” by a particular worker thread. This thread is tracked using cilk::current_worker_id(). When a thread calls acquire, if it is not the owner (has a different worker id), then it makes a copy of the state and continues.

Because a worker thread may make changes to the state while it is running, it is possible that the “stealing” worker would get an inconsistent copy of the state. This is solved by the commit method. This method copies the state that changes so that it can be referred to by a stealing thread. One parameter to the function is the version of the state that it should use when it copies. For backtracking search, this is simply the depth of the recursion, so there is very little state that needs to be saved (at most the depth of the search tree, which is logarithmic on the total work).

I also planned originally on implementing an abort operation that would propagate an error code to through the call graph. I ran into some problems getting this working, and because it wasn’t necessary for this problem I gave it up. Instead, I use a global flag that stores if a solution has been found and what the solution path is.
At a higher level, I parallelized the search by essentially using a \texttt{cilk\_for} at each level of the recursion. As my results show, this provides ample parallelism. In theory, I could also parallelize the spawning of each level of the recursion, but this ended up introducing some bugs I didn’t have time to track down. As a detail, because Cilk++ imposes restrictions on the form of a \texttt{for}-loop (no breaks allowed), I had to manually write the divide-and-conquer logic.

2 Results

I gathered results for the following input files: \texttt{test6.in}, \texttt{test7.in}, \texttt{test8.in}, and \texttt{long\_test1.in}. These were chosen because they cover a large range of problem sizes and are the largest problems. Table 1 shows that my algorithm exposes ample parallelism.

<table>
<thead>
<tr>
<th>Test</th>
<th>Parallelism</th>
</tr>
</thead>
<tbody>
<tr>
<td>test6.in</td>
<td>71930</td>
</tr>
<tr>
<td>test7.in</td>
<td>244127</td>
</tr>
<tr>
<td>test8.in</td>
<td>308923</td>
</tr>
<tr>
<td>long_test1.in</td>
<td>587942</td>
</tr>
</tbody>
</table>

Table 1: Parallelism.

In some cases, there was very unusual behavior. For example, \texttt{test8.in} takes roughly 100 seconds to run on a single thread. But with 2 or more threads, it completed in roughly 0.4 seconds.\footnote{The checker confirms that it finds a valid solution.} This was even more extreme for \texttt{long\_test1.in}. This test took 0.002 seconds to complete with more than 3 workers, but 270 seconds with fewer. This behavior can easily be explained by the properties of the problem – if any solution is found, then the algorithm can terminate. By dividing the problem essentially randomly among the threads, some divisions will get “lucky” very quickly and find a good solution. This behavior depends in large part on how uniformly solutions are distributed across the space of all solutions.

I’ve included a few graphs from cilkview for these tests in Figure 1. As you can see, inputs \texttt{test8.in} and \texttt{long\_test1.in} look strange. But the other results show good scaling, albeit with some noise due to the same reason. Also note that the number of worker threads goes up to 16, because hyperthreading is enabled on the machine I was using. Surprisingly, scaling is still quite good (although not as good and less consistent) on the hyperthreaded cores.

I show in Table 2 the relative performance of serial and parallel versions. These numbers show the overhead of parallelizing the algorithm and the speed-up that can be achieved. I only show these for \texttt{test6.in} and \texttt{test7.in}, because the other results are somewhat irrelevant. This table shows excellent speed-up with parallel threads, approximately 10× across 8 cores. I suspect the superlinear results are due to the same effect discussed above. However, the overhead

\footnote{The checker confirms that it finds a valid solution.}
introduced in parallelizing the code is pretty unacceptable. (This is probably largely why it scales so well.)

If I had more time, I would focus on tracing down where the overhead is coming from. It is possible that the overhead is fundamental in my approach of tracking state to enable on-demand copies, in which case there is little I could do to correct it. But I suspect it is a simple matter of optimizing data references through the structure. On the bright side, even with a horrible parallel implementation, I manage to beat the performance of serial code.

<table>
<thead>
<tr>
<th>Serial</th>
<th>test6.in</th>
<th>test7.in</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.597 sec</td>
<td>8.278 sec</td>
<td></td>
</tr>
<tr>
<td>Parallel w/ 1 thread</td>
<td>5.048 sec</td>
<td>32.025 sec</td>
</tr>
<tr>
<td>Parallel w/ 8 threads</td>
<td>0.693 sec</td>
<td>3.298 sec</td>
</tr>
</tbody>
</table>

Table 2: Performance comparison.

I also did not have time to implement the upper bound heuristic with my algorithm. This would be fairly simple to do using my data structure, as the
“current move table” is just another piece of state that must be copied to each thread. I suspect it would have the same problems as the current implementation. I have also already spent an unnecessarily large amount of time on this, as indicated by my title, although that’s probably my own fault. My approach was probably over-designed from the start, and I should have tried a simpler approach to begin with.

One thing that would have made my life much easier is if Cilk++ provided some kind of callback to set up local storage for the thread on a spawn. (Maybe this exists?) This would have enabled copying to be done without the awkward tracking of worker threads. I found myself sometimes frustrated and wanting to implement my own parallel framework in pthreads just to get this feature. The (apparent) lack of a debugger with full symbol information also depressed me.\(^2\)

\(^2\)I ended up converting and compiling the code as plain C++ in order to use \texttt{gdb}.
1 Introduction

This lab explores using backtracking search to find solutions to the game of peg solitaire. Peg solitaire is a game with \( n \) rows and \( m \) columns of holes, where each hole can be invalid, filled, or empty. Jumps consists of taking a peg, and moving it 2 spaces north south east or west with the destination hole being empty, and the intermediate node containing a peg. Thus, a jump consists of a start hole, and a direction. The intermediate hole’s peg is removed, making the hole empty. The goal is to transform one board configuration into another via valid jumps. Solutions can be found using a depth first backtracking search, a serial version of which might look like this:

However, since the branches of the search tree are independent, we can search in parallel.

2 Parallel Backtracking Search Data Structures

The first issue in parallelizing the algorithm is how to keep track of the path we are working with. In a serial implementation, an array is sufficient, by simply keeping a counter to what step in the path we are on, and decrementing to backtrack. This clearly will not work in a parallel algorithm because of race conditions. I chose to resolve this by having a struct called PathNode which includes a Jump, as defined earlier, and a pointer to the PathNode representing the previous Jump. It also includes a boolean, sol, to indicate whether this PathNode is part of a valid solution. The method markSolved sets sol to 1, and calls the method on the previous jump. This allows us to mark an entire list of PathNodes as being part of a solution by calling the method on the final PathNode.

### Figure 1: Serial algorithm for backtracking search

```plaintext
struct SimpleJump

FINDSOLUTION(start,final,path)
if start.numPegs = final.numPegs
    return (start = final)
else
    for each jump J in \{0,n\} \{0,m\} (NORTH,EAST,SOUTH,WEST)
        if J is a legal jump for start
            start.makeMove(J)
            path.push(J)
            found = FINDSOLUTION(start,final,path)
            if found
                return TRUE
            else
                start.makeReverseMove(J)
                path.pop()
return FALSE
```

Figure 1: Serial algorithm for backtracking search
Another issue is that the start board is modified as we search. This data race is initially solved by simply copying the board, despite the possible high overhead of copying.

### 3 Algorithm Modifications

Initially, we took the route with the highest parallelism, simply substituting a cilk for loop and making the necessary data structure changes described earlier.

```plaintext
FINDSOLUTION(start, final, previousJump)
if start.numPegs = final.numPegs
    found_solution = previousJump
    found_solution.markSolved()
    return previousJump
else
    cilk_for each jump J
        if(!found_solution)
            if J is a legal jump for start
                newboard = start
                newboard.makeMove(J)
                curJump = new PathNode(J, prevJump)
                found = FINDSOLUTION(start, final, curJump)
                if sol_found && curJump.sol == 0
                    curJump.deletePath
                else
                    if(curJump->sol == 0)
                        delete curJump
                    return NULL
            return NULL
```

A global variable, found solution is kept to ensure all spawned threads know when to stop executing. When a solution is found, it is set to the PathNode of the last jump in the sequence, and markSolved is called on it to ensure every PathNode in the sequence is marked. In the for loop, we would normal return upon finding a solution. However, returning in a cilk for loop does not make sense, nor can we break out of it. What we really need is an abort command of some sort that would notify all sibling threads, all threads spawned by a parent to stop executing. This is somewhat complicated, but the problem allows the easier solution of simply halting ALL threads, since once a solution is found, we are completely finished. Thus the body of the for loop is surround in an check for a found solution. This causes threads to simply skip through the parallel for loop upon a found solution, which essentially is an abort. An interesting challenge with the data structure chosen for paths was memory management. Since each explored node in the graph creates the PathNode data structure, it is important to make sure those nodes get deallocated. From the serial program statistics upwards for 40 million nodes were explored even for the relatively small test cases, which makes for a great
deal of memory being used without proper deallocation. The strategy I chose was to have a destructor that would delete all ancestors that are not part of a solution. deletePath, and a destructor that would just delete that node. Since a found solution will update all ancestors and set the global, I can check those conditions to decide which destructor to use. This worked well in a serial implementation, as valgrind determined there was virtually no leaked memory, but in a parallel implementation, I hit a roadblock of what occurs when two children, which are leaves, both call the destructor of the parent simultaneously.

4 Analysis

The version as described so far has very good parallelism, each potential starting location is considered in parallel. On an \( n \) by \( m \) board with \( s \) pegs on the start and \( f \) pegs in the final configuration, we can expect span to be the length of the path of moves, which is \( s-f \) since each move removes a peg. An upperbound on the work is simply the number of ways to arrange the pegs on the board, which could be estimated as \( \binom{n+m}{s} \). Thus parallelism is \( O\left(\frac{(n+m)!}{(s-f)!s!(n+m-s)}\right) \). Parallelism is quite high but because of each step of the search creating a PathNode, and copying the board, this overhead could reduce the speedup provided by our high parallelism. The overhead of pathNodes probably can’t be helped, as we must consider the potential paths independently, and returning values is not possible within the for loop. As for the copying of the board, the overhead can be reduced by using a more compact board representation, such as a bitstring for the board positions. Since each holes status can be represented with 2 bits, we can potentially reduce the board size to \( n \times m \times 2 \) bits rather than the the \( 8 \times 4 \times n \times m \) bits with the current integer enumeration data structure. Although this could significantly improve the overhead of copying, I prioritized balancing the tradeoff between parallelism and copy overhead since I predicted far more parallelism than would be needed for our cagnodes (greater than 10x processor count).

5 Results

Initial results were extremely promising as a serial version that did not deallocate the PathNode structures still performed comparably to the original code, an average of 3x slowdown over the original code. Due to the memory leak, it was not possible to run the entire test case suite at once. Initial attempts at resolving the leak by using the different destructors caused inconsistent performance, as crashes resulting from double frees occurred on some test cases. This the analysis was done with no memory deallocation, making it race free, but with the knowledge that a remarkably large memory leak occurs while a solution is developed. In the 3 cases analyzed, Parallelism is indeed quite high. Test cases 5 and 6 were 5 by 5 boards, and 7 was a 5 by 6 board, all with single peg solutions.

<table>
<thead>
<tr>
<th>Test case</th>
<th>Parallelism</th>
</tr>
</thead>
<tbody>
<tr>
<td>Test case 5</td>
<td>54.51</td>
</tr>
<tr>
<td>Test case 6</td>
<td>2342.4</td>
</tr>
<tr>
<td>Test case 7</td>
<td>13296.83</td>
</tr>
</tbody>
</table>
Trial results for 'Test 5'

Trial results for 'Test case 6'
One really interesting observation was that in some cases, such as test case 5 and long test1, superlinear speedup is observed. Long test1, a 7 by 6 board, is solved in .003 seconds with a parallel algorithm, but in 29 seconds by the serial. Considering the machines have but 8 cores and because of Amdahl’s law, I found this curious. Because of the parallel nature of the search, it is possible that occasionally, we can find a solution that is relatively high in the search tree but a sibling of the the branch the serial version would search first. This almost suggests that a parallel algorithm would be more effective than a serial one, even on a single core machine, but the opposite case would also be possible, where the solution is relatively shallow in the first branch and the graph has a wide branching factor. This seems like the comparison between BFS and DFS, in some cases one is better than the other.

6 Further Work

Probably the single most effective improvement will be to implement a hashtable and search from the end configuration and beginning configuration simultaneously. This will reduce the work by a square root factor, and the span by half. More important is to rethink the nature of the data structures used to keep track of visited nodes in the dynamically generated graph. A planned fix is to have PathNodes keep pointers to their parent and to their children, though this would make the data structure much larger. Another potentially easier fix is to have a global vector which will contain a solution, and when a solution is found, to lock it and walk back up the tree to add the needed jumps to the vector. This would allow stack allocation of the nodes as well as avoid memory issues. The contention for the lock should predictably be low, since solutions will be found infrequently.
1 Overview

In this lab I parallelized the peg solitaire solver with and without the Kiyomi and Matsui heuristic. I implemented a more space-efficient representation for board configurations, which only uses 2 bits to store square status and additionally improves board scanning. I also added hashing to store intermediate board configurations and avoid re-computations.

1.1 Benchmarks

There are several things to consider when constructing the input boards: the dimensions of the board, the number of filled/invalid holes, and whether there is a solution path from the starting configuration to the final configuration. If there is no solution for a given start and end configuration, then the solver will have to check all the possible paths before it can finish. Experimenting with the layout of the boards, we can observe that the solver will take longer for any board size if the start configuration is entirely filled with pegs except for one empty hole towards the center and the stop configuration has only one peg. Adding additional empty holes or invalid squares, reduces computation time significantly. I will refer to a configuration as \((x - \text{empty}, y - \text{inv})\), meaning there are \(x\) empty holes and \(y\) invalid squares on the board.

2 Parallel Solver

Given a board configuration, our initial solver scans through all the squares and attempts to make all possible jumps from each square. We can parallelize the solver by scanning the squares in parallel (using a \textit{cilk for} over the board). Since making a jump will change the board, we need to make sure that each worker operates on a \textit{copy} of the board to avoid races. We also need to ensure that if a path is found, all the workers terminate their computations. In order to accomplish this, I introduced a global variable \textit{global\_found} that is updated when a solution is found. I used a lock for this variable to ensure that only one thread reports finding a solution and constructs the jump sequence. Reading the variable does not require locking since any races here are benign and so this lock should not create significant overhead (only the threads that found a solution around the same time will compete for the lock).

In order to avoid copying in each iteration of the \textit{cilk for} loop, we can define a grain size, \(GS\), and divide the board into \((\text{total}\#\text{of squares})/GS\) sections over which we can scan in parallel. Each worker then needs to make the copy of the board once before scanning the squares in its subsection.

Table 1 shows the results of the parallel solver (\textit{PSolver}) using the initially provided board representation (\textit{SimpleBoard}), with and without \(GS\) (I found that \(GS = (\text{total}\#\text{of squares})/2\) works
well). Note: the code without GS uses the `cilk_grainsize` parameter for the `cilk_for` loop in order to reduce cilk overhead, this parameter is also set to `(total # of squares)/2.

Table 1: Solver Running Times (s), w/out Heuristics (X = over 2 minutes)

<table>
<thead>
<tr>
<th>Board Size</th>
<th>Sequential</th>
<th>PSolver(Ser)</th>
<th>PSolver(Par)</th>
<th>PSolver+GS(Ser)</th>
<th>PSolver+GS(Par)</th>
</tr>
</thead>
<tbody>
<tr>
<td>4x4 (1-empty, 0-inv)</td>
<td>3.819</td>
<td>5.764</td>
<td>0.79</td>
<td>3.686</td>
<td>0.773</td>
</tr>
<tr>
<td>4x5 (2-empty, 1-inv)</td>
<td>164.047</td>
<td>165.558</td>
<td>25.254</td>
<td>165.38</td>
<td>27.682</td>
</tr>
<tr>
<td>5x5 (2-empty, 4-inv)</td>
<td>103.925</td>
<td>101.468</td>
<td>16.863</td>
<td>101.878</td>
<td>15.739</td>
</tr>
</tbody>
</table>

Without heuristics, the solver handles the worst case configurations on boards up to size 4x4. However, when the size is increased to 4x5, the computation does not finish in a reasonable time. In order to reduce computation time, I increased the number of empty/invalid holes in the starting configuration. The 4x5 and 5x5 boards in Table 1 finish in about 2 minutes using the original sequential solver. The times for the parallel solver code run serially are close to the sequential code time, and therefore, the board copying does not create too much overhead for these board sizes. The parallel solver with and without GS take approximately the same time. Cilkview reports parallelism of 256.38 (burdened = 77.00) with GS and 1469.73 (burdened = 284.06) without GS (Figure 1-2 show both cilkview plots for the 5x5 board).

3 Solver Optimizations

3.1 Space Efficient Board Configuration and Improved Board Scanning

The provided representation of board configurations (SimpleBoard) stores the status of each square of the board (invalid/filled/empty) as an integer value, forming an array of integers of size N*M (N = number of rows, M = number of columns). In order to conserve space and speed-up board copying, I replaced this array with two arrays of only N integers, where each integer stores information about all the squares in a given row - each square only uses 1 bit of information. The first array stores
the valid/invalid status of each square and the second array stores the filled/empty status. Only the second array needs to be copied in the parallel solver computations, since square validity never changes. Note: this representation assumes that boards with both N and M greater than 32 will take too long to compute and therefore one dimension can fit into a 32-bit integer (I ran the solver with heuristics on a 32x32 board and the computation did not finish in 20 minutes).

This representation also lets us traverse the board more efficiently. Instead of trying to make a move from each square of the board (regardless of its status), we can now only consider the filled squares of the board. For each row value of the filled/empty status array, we can continuously get the least significant set bit, attempt a move from the position corresponding to that column, and remove the bit after we are done with the position - until all the originally set bits have been checked. I only parallelized the loop over the rows of the board (the least significant bit computations are done by the same worker).

3.2 Hashing

We would like to stop computing along a branch if we got to a board configuration, which we have already seen along a different path. In order to store what configurations we already saw, we can use a hash table. I implemented a hash table with a lock for each table entry. Before recursing along a branch, I try to add the new board configuration into the table, this add will succeed if there was no previous entry for this configuration in the table; otherwise, it will fail and this branch will be stopped (we need to lock the entry during this try-add operation). In case of a collision, I keep the most recent board in the table. The size of the table was set to 1024 * 1024 * 16. A hash value was computed from the N integer row values of the board.

3.3 Optimized Results

<table>
<thead>
<tr>
<th>Board Size</th>
<th>Opt(Ser)</th>
<th>Opt(Par)</th>
<th>Opt+Hash(Ser)</th>
<th>Opt+Hash(Par)</th>
<th>Opt+Heur(Precomp, Ser, Par)</th>
</tr>
</thead>
<tbody>
<tr>
<td>4x4 (1-empty, 0-inv)</td>
<td>2.121</td>
<td>0.597</td>
<td>1</td>
<td>0.93</td>
<td>4.12, 0, 0</td>
</tr>
<tr>
<td>4x5 (2-empty, 1-inv)</td>
<td>85.122</td>
<td>20.22</td>
<td>0.886</td>
<td>1.224</td>
<td>0.169, 0, 0</td>
</tr>
<tr>
<td>5x5 (2-empty, 4-inv)</td>
<td>58.412</td>
<td>11.272</td>
<td>1.73</td>
<td>1.036</td>
<td>0.033, 0, 0</td>
</tr>
<tr>
<td>4x5 (1-empty, 0-inv)</td>
<td>X</td>
<td>X</td>
<td>0.964</td>
<td>0.903</td>
<td>14.101, 0, 0</td>
</tr>
<tr>
<td>5x5 (1-empty, 0-inv)</td>
<td>X</td>
<td>X</td>
<td>9.244</td>
<td>2.768</td>
<td>X, ?, ?</td>
</tr>
<tr>
<td>5x6 (2-empty, 4-inv)</td>
<td>X</td>
<td>X</td>
<td>6.432</td>
<td>2.392</td>
<td>18.045, 0, 0</td>
</tr>
<tr>
<td>6x6 (8-empty, 3-inv)</td>
<td>X</td>
<td>X</td>
<td>36.947</td>
<td>12.875</td>
<td>90.269, 0, 0</td>
</tr>
</tbody>
</table>

Table 2 contains performance numbers for the optimized board. Comparing the first 3 rows and 2 columns of Table 2 with Table 1, we can notice that the optimized board (w/out hashing and heuristics) reduced the serial running time twice. It also reduced the parallel running times. Using the optimized board, however, the solver still could not solve the 4x5 worst case board. Adding hashing significantly improved solver performance. The solver was able to compute the worst case 4x5 and 5x5 boards fast. Although the solver with heuristics (ignoring precomputation time) ran faster comparing to the hashing for smaller boards, its precomputation step took far too long for the worst case 5x5 board (did not finish after 5 min). To present a few more performance numbers
for comparison, I increased the number of empty/invalid holes on larger boards. We can notice that the serial and parallel solver with hashing seem to take the same amount of time on smaller boards, this is because the time is dominated by the initialization of the hash table.

If we include the heuristic precomputation time, the hashing solver runs fastest. The speedup on the largest worst case board it can compute (5x5) is $9.244/2.768 = 3.34$. 


1 Parallelizing Backtracking

1.1 Copy Requirements

The first requirement for parallelizing a backtracking search is to properly deal with changes in the board configuration. Since each branch in the search adds a possible jump to the board, it changes the configuration. If this happens in parallel, the search will at best not be exhaustive and at worst reach invalid states.

To solve this problem, any branch which can be executed in parallel needs to copy the board configuration. This operation is expensive, over tripling the time of visiting a branch. A more memory efficient approach could use a hash map or a tree to store only those positions which are filled. Then, positions which are invalid could be stored in a global structure. This would reduce the cost of copy operations and allow deeper spawns. I’m not entirely convinced that this will be extremely beneficial, though if I had more time I would have implemented this.

1.2 Abort Statements

Since backtracking search uses a depth first search strategy to, when one branch finds a solution, the whole search should terminate. Because multiple threads could be running, this is not accomplished when the bottom thread returns. Therefore, an abort variable needs to be used. In my implementation, a boolean pointer is passed to each recursive call in the search. Before each child checks a new configuration, it checks the boolean and returns when the boolean flips. This variable does not need a lock, though a write-race occurs, since it is okay if more than one thread begins to return since the parent branches use locks to prevent multiple child threads from returning a solution. This of course, causes a determinancy race – so that the number of attempted jumps varies with each run. This effect will be noticeable when I examine the parallel speedup of the heuristic algorithm.

2 Parallelizing of the Simple Strategy

Because, the simple strategy performs such an exhaustive search, especially on impossible boards, the parallelized version exhibits parallelized speedups that cilkview would expect (Figure 1.)
The results of parallelizing the jump-heuristic algorithm are interesting. Using the technique described in Section 1, I had the runtimes in Figure 3 on the given `longtest1` and the problem in Figure 2, which I refer to as the `english` problem.
<table>
<thead>
<tr>
<th>Worker Count</th>
<th>English runtime (s)</th>
<th>Longtest1 runtime</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>7.446</td>
<td>9.508</td>
</tr>
<tr>
<td>2</td>
<td>0.019</td>
<td>9.044</td>
</tr>
<tr>
<td>3</td>
<td>0.02</td>
<td>9.232</td>
</tr>
<tr>
<td>4</td>
<td>0.019</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>0.02</td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td>0.02</td>
<td>0.001</td>
</tr>
<tr>
<td>7</td>
<td>0.018</td>
<td>0.001</td>
</tr>
<tr>
<td>8</td>
<td>0.018</td>
<td>0.001</td>
</tr>
</tbody>
</table>

Figure 3: Runtime for parallel heuristic solver as worker count increases.

As you can see, the speedup is immense - on 8 cores, the solution was found orders of magnitude faster. Cilkview gives a parallelism of 775,566 for longtest1 (Cilkview output in Figure 4) and 116.5 for english, which means there is a great deal of parallelism. But why is there so much speedup and why does the speedup occur so abruptly?

Because threads are spawned at the top level, the search actually operates as a kind of hybrid between breadth-first and depth-first search. At a certain number of workers, one of the branches which contains a fast solution is explored long before the other branches finish. If the problem required a more exhaustive search (for example a minimum length solution) then the speedup would behave more like what is predicted by cilkview.
4 Additional Heuristics for Parallel Approach

4.1 Fillable / Emptyable Square

Using the upper bound for jumps, I can perform some additional pruning. If a square in a configuration is empty, and filled in the final configuration, then one of four possible jumps must occur at some point in the future. A similar constraint holds for squares that are full in a configuration and empty in the final configuration. I use this to prune configuration by checking each square that changes during a jump (this tests 3 squares with at most 16 possible jumps.) This operation is fast, however, in my tests, did not perform much actual pruning.

4.2 Quick and Dirty Validity Testing

By far the best “heuristic” I added was to prevent testing jumps which started from squares without pegs. This massively improved parallelism, since cilk would not be overrun with spawns on loop iterations which would quickly end.

long_test1.in
w/o heuristic: 107105655 jumps tested. Runtime: .98 s
w/ heuristic: 7831 jumps tested. Runtime: .002 s

english.in
w/ heuristic: 22612357 jumps tested. Runtime: .138 s
w/ heuristic: 2993694 jumps tested. Runtime: .022 s

5 Further Work

I wish that I had time to implement a hash table to store previously visited configurations. On top of that I would have liked to implement reflection and rotation.
Lab 5 Writeup

Introduction

In Lab 5, I explored backtrack search within the context of peg solitaire. By combining several optimizations over different iterations, I came up with an efficient solver for peg solitaire that had lots of parallelism and little overhead. I also experimented with Kiyomi and Matsui’s integer linear programming heuristic and the idea of lazily computing the heuristic when needed. My writeup is broken up into the following sections:

- Section 1 discusses a naive approach to parallelizing backtrack search.
- Section 2 shows how we can coarsen to trade parallelism for performance.
- Section 3 outlines a technique to avoid searching the whole space.
- Section 4 explains why mounting your lab while working in eclipse is dangerous!
- Section 5 reveals mixed results with respect to lazy computation vs pre-computation of Kiyomi and Matsui’s heuristics.

1 Simple Parallel

First, I implemented a \textit{parallel\_naive}, a naive parallel backtrack search for peg solitaire. In \textit{parallel\_naive}, I used two \texttt{cilk\_for} loops to explore each new valid move in parallel on each level of the recursion. To maintain correctness, a copy of the board and path must be made each time a valid new move is explored. Otherwise, multiple workers may mutate the same board / path at the same time, causing a race condition.

\textit{parallel\_naive} has two issues:

1. Parallelizing down to the leaves of the search space, while generating a huge amount of parallelism, introduces lots of overhead, because the board and path must be copied for each recursive call. This suggests that coarsening may be extremely useful, since we can sacrifice parallelism for efficiency at lower levels of the recursion.

2. Returning or breaking out of \texttt{cilk\_for} loops is not allowed in \texttt{cilk++}, so \textit{parallel\_naive} can not quit early once it has found a solution. This means, \textit{parallel\_naive} must visit the entire search space. For boards with no solution, this doesn’t matter, but for boards with many solutions or solutions that occur earlier in the search space, \textit{parallel\_naive} will not perform optimally.

To isolate the first issue, I tested \textit{parallel\_naive} on boards where the whole search space must be visited anyways: boards with no solution. Because \textit{parallel\_naive} runs slowly, it was only practical to test on small boards, so test7.in was chosen as a benchmark. The results of running test7.in on an 8-core cagnode machine are shown in Table 2.

\begin{table}[h]
\centering
\begin{tabular}{|c|c|c|c|}
\hline
\text{test7.in} & \text{T}_s & \text{T}_1 & \text{T}_8 \\
\hline
 & 6.69 s & 27.66 s & 4.66 s \\
\hline
\end{tabular}
\caption{Time taken for test7.in}
\end{table}

\text{T}_s was the time to run the original serial program, \text{T}_1 was the time to run \textit{parallel\_naive} on one core, and \text{T}_8 was the time to run \textit{parallel\_naive} on 8 cores. On one core, \textit{parallel\_naive} ran much slower ($\text{T}_1/\text{T}_s = 4.13$) than the original serial program, confirming the overhead issues of \textit{parallel\_naive}. Encouragingly, \textit{parallel\_naive} ran slightly faster than the original serial program on 8 cores ($\text{T}_8/\text{T}_s = 1.43$).

Cilk\_view reported work of 142310089388 instructions, span of 46358 instructions, resulting in huge amounts of parallelism: 3069806.49. The scalability graph is shown in Figure 1.
Figure 1: Scalability graph for parallel_naive on a cagnode machine with 8 cores.

2 Coarsening the search

Since the parallelism of parallel_naive was extremely high (3069806.49), I implemented parallel_coarsen, a coarsened version of parallel_naive. In parallel_coarsen, I parallelized the upper $k$ levels of the search space to generate ample parallelism, and serially searched the remaining levels to reduce overhead.

To see the improvements from parallel_naive to parallel_coarsen, I ran parallel_coarsen on test7.in on a cagnode machine. $k = 4$ produced the best balance between overhead and parallelism on 8 cores (cilkview reported the parallelism to be 64.94). The cilkview scalability information is graphed in Figure 2.

Figure 2: Scalability graph for parallel_coarsen on a cagnode machine with 8 cores.
parallel_coarsen on 8 cores was roughly 8 times faster than the original serial program ($T_S = .764$ s vs $T = 6.69$ s), which shows that the overhead due to duplicating grids and paths at higher levels in the search space is negligible compared to the speedups from parallelism. However, since parallel_coarsen cannot quit early, I found that on many boards with solutions, it ran slower than the original serial version. In fact, on a relatively large board with many solutions, such as test8.in, parallel_coarsen did not complete in any reasonable time: it was still searching the entire space after 30 minutes, as opposed to the original serial version, which found a solution in 17.73 s.

3 Quit Early

To be of any use on boards with a large search space and many solutions, parallel_coarsen needed a mechanism to quit early once a solution was found. To facilitate this, I added a global boolean, globalFound, and initialized it to false. If any worker found a solution, it immediately set globalFound to true. Each level of the recursive parallel search checked globalFound before searching further, so once the first solution was found, each worker searched at most one level deeper, before stopping.

globalFound introduces a race condition as multiple workers may try to access and set the boolean at once. However, since it is only set to true and only once a path is found, behavior is still correct if multiple workers overwrite it. Therefore, the race condition is benign.

After implementing the globalFound variable, most of the boards ran much faster. On a cagnode machine, running with 8 cores, test8.in found a solution in an astonishing .042s. While test8.in was an extreme case, resulting from one of the workers discovering a path abnormally fast, this optimization speeds up the expected run time of boards with solutions considerably. Assuming even distribution of solutions in the search space, a parallelized backtrack search with globalFound would be expected to run $s$ times faster than one without globalFound, where $s$ is the number of solutions. Since most boards with solutions have tons of solutions due to symmetry (especially as the board size grows), implementing globalFound is very useful step in creating a fast parallel backtracking algorithm for peg solitaire.

I tested this new optimization on long_test1.in, long_test2.in, and long_test3.in. long_test1.in was another case of a worker discovering a path abnormally fast, and returned a result after just .001s. long_test2.in, and long_test3.in, however, could not be solved in a reasonable time with just the globalFound optimization and parallel_coarsen. This makes sense because these boards do not have any solutions, which means that parallel_coarsen is stuck searching the entire space for a long time.

4 Cagnode Crashes in Conjunction With Mounting and Eclipse

To deal with boards that have large search spaces, and no solutions (or very few), like long_test2.in and long_test3.in, I decided to integrate parallel_coarsen and globalFound with pruning techniques developed by Kiyomi and Matsui. Kiyomi and Matsui use integer linear programming to compute an upper bound on the number of times a jump can occur in the solution sequence. By applying this pruning technique to a backtrack search, a significant amount of the search space can be discounted.

I took Kiyomi and Matsui’s pruning techniques and applied them to parallel_coarsen. I verified that they worked and that parallel_coarsen could now solve long_test2.in and long_test3.in (I have no data saved on the exact run times, but they were reasonably fast). Unfortunately, the cagnode machines went down. Normally, this wouldn’t be a problem, but I mount my labs from cagnode5 to my laptop and edit in eclipse. Eclipse tried to refresh the files while I was unmounting and remounting, and wiped all of the files I had open. My parallel backtrack search solver was overwritten and lost. At this point, I had already invested lots of time into the lab, and all I had left to do was extend the lab, so I chose not to re-implement my parallel backtrack search.
Lazy Kiyomi and Matsui Pruning

For my extension, I explored the idea of lazily computing the upper bounds, $R(J)$, as needed, instead of relying on a pre-computation step. The hope was that if not every jump had to be considered before finding a solution, lazy computation could avoid unnecessary work. Since parallel_coarsen in conjunction with Kiyomi and Matsui pruning was wiped, I was only able to edit the serial version of Kiyomi and Matsui pruning. However, since the serial and parallel versions should visit a similar number of jumps, it is reasonable to assume that any benefits to a serial version with lazy Kiyomi and Matsui pruning would be applicable to parallel_coarsen. Preliminary estimation of speedups due to lazy computation of upper bounds yielded mixed results.

Rather than implement lazy computation of $R(J)$, I ran some preliminary tests to weigh benefits against costs. I added a boolean array to PegSolverStats of length $num$ _jumps_ and set the index corresponding to a jump to True whenever that jump was visited. After solving a peg board, the number of True’s in the boolean array represented the number of unique jumps visited, which is the number of $R(J)$’s that would have to be computed in a lazy implementation.

For each board provided with solutions, I printed the number $R(J)$’s that would need to be computed using lazy computation and precomputation. The results are summarized in the table below:

<table>
<thead>
<tr>
<th>Board</th>
<th>number of $R(J)$ computed (lazy computation)</th>
<th>number of $R(J)$ computed (pre-computation)</th>
<th>speedup</th>
<th>pre-computation time</th>
<th>search time</th>
</tr>
</thead>
<tbody>
<tr>
<td>test2.in</td>
<td>41</td>
<td>76</td>
<td>1.85</td>
<td>.427 s</td>
<td>.006 s</td>
</tr>
<tr>
<td>test3.in</td>
<td>39</td>
<td>76</td>
<td>1.95</td>
<td>.574 s</td>
<td>.004 s</td>
</tr>
<tr>
<td>test5.in</td>
<td>35</td>
<td>52</td>
<td>1.48</td>
<td>.101 s</td>
<td>.004 s</td>
</tr>
<tr>
<td>test8.in</td>
<td>69</td>
<td>76</td>
<td>1.10</td>
<td>1.776 s</td>
<td>21.75 s</td>
</tr>
<tr>
<td>long_test1.in</td>
<td>51</td>
<td>52</td>
<td>1.02</td>
<td>.162 s</td>
<td>20.01 s</td>
</tr>
</tbody>
</table>

These tests revealed decent gains from lazy computation of $R(J)$ on small boards. The speedup column, estimates the gains from lazy computation by dividing number of $R(J)$’s computed in the pre-computation version by the number of $R(J)$’s computed in the lazy version. As indicated, the speedups for test2.in, test3.in, and test5.in were decent: 1.85, 1.95, and 1.48 respectively. Since the pre-computation time dominated the search time, these lazy computation would significantly improve runtimes on small boards.

However, on large boards, lazy computation wasn’t nearly as effective. Speedups were only 1.10 and 1.02 for test8.in and long_test1.in respectively because $R(J)$ needed to be computed for nearly every jump. In addition, in test8.in and long_test1.in, the search time dominated the pre-computation time (21.75 s vs 1.776 s and 20.01 s vs .162 s for test8.in and long_test1.in respectively), so gains from lazy computation would contribute even less to the total runtime.

In summary, it is not certain how useful lazy computation would be. While it helps a lot on small boards, the backtrack solver can already solve them very quickly. We care mostly about large boards because they take a long time to solve. Since the speedups in calculating $R(J)$ diminish as the boards grow larger, lazy computation of $R(J)$ has diminishing returns as the problem size grows. In addition, optimizing the time taken to calculate $R(J)$’s is less worthwhile on large boards because the time spent computing $R(J)$’s is smaller relative to the overall time searching. However, since the backtrack search has a huge amount of parallelism, it is conceivable that if we ran large boards on machines with a significant number of cores, the pre-computation time of $R(J)$ might start to be a larger factor.

To explore this further, it would be interesting to search harder for large graphs where a small fraction of jumps are visited before finding a solution. However, I hypothesize that if many jumps were not visited, the solution would appear pretty early in the search space. If this is the case, the program would probably run sufficiently fast without lazy computation.
Numbers were collected on a four-core, dual-hyperthreaded Intel Core i7 system with 6 GiB of memory.

(a) There are several input characteristics which influence the performance of the simple solver; a worst-case input would satisfy all of them. First, since the solver short-circuits when it finds a solution, problems without solutions are most expensive (since the entire search space must be exhausted). Also, boards with more pegs (in the extreme, boards with only a single open space) produce vastly larger search spaces when paired with goal states with few pegs (in the extreme, a single peg).

As the size of the board increases (and the number of pegs, in configurations with only one vacancy), the solver becomes dramatically slower. For example, a $4 \times 4$ board took 6.0 seconds (this was the largest that could be solved in two minutes); a $4 \times 5$ board took approximately 21,000 seconds or 5.83 hours.

(b) The parallel solver compiles and works correctly. In this situation, a valid solution with path length 24 was reported in 19.43 seconds.

(c) Ideally, we might achieve perfectly linear speedup if the processors can work together without overhead to explore the search space. The machine I was using had four physical processors (or eight, if you are a fan of hyper-threading); this is not nearly enough to close the gap to the next-largest board size ($4 \times 5$, with a serial execution time of hours).

There was also some significant overhead added by parallelization; this stems from the need to copy board configurations. Since the solver modifies each board as it searches, two workers cannot share the same copy. I attempted to cut down on this overhead by making the copy operation faster and by coarsening chunks of the search space (within which the normal backtracking method can be used) but it was still significant (almost a factor of two).

I also saw some unusual timing behavior that prevented me from collecting data in a conventional fashion. In situations where the solver short-circuited, timings varied from precisely the same as the serial version to nearly $\frac{1}{p}$ as long (the speedup we would expect). I believe that this variation is a result of scheduling; when one processor is left to explore the branch that will eventually result in the short-circuiting solution, we get times just as with the single-worker version, but when the work happens to be well-divided, that solution can sometimes be found more rapidly.

<table>
<thead>
<tr>
<th>Configuration</th>
<th>Time</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>$T_{\text{serial}}$</td>
<td>6.01 sec</td>
<td>1.00</td>
</tr>
<tr>
<td>$T_1$</td>
<td>10.53 sec</td>
<td>0.57</td>
</tr>
<tr>
<td>$T_{8,\text{min}}$</td>
<td>2.42 sec</td>
<td>2.48</td>
</tr>
</tbody>
</table>

(d) I then modified this parallel solver to use Kiyomi and Matsui’s heuristic (the $R$ function) to prune the search space. My initial implementation of this idea precomputed the
heuristic, after which it is shared in a read-only fashion, eliminating most concurrency issues. However, the parallel speedup of this algorithm is not quite as good as that from the heuristic-free algorithm. Times given are for executing `long_test1.in`. The same unusual timing variations occurred.

<table>
<thead>
<tr>
<th>Configuration</th>
<th>Time</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>$T_{\text{serial}}$</td>
<td>19.43 sec</td>
<td>1.00</td>
</tr>
<tr>
<td>$T_1$</td>
<td>41.34 sec</td>
<td>0.47</td>
</tr>
<tr>
<td>$T_{8,\text{min}}$</td>
<td>9.86 sec</td>
<td>1.97</td>
</tr>
</tbody>
</table>

I then switched to precomputing portions of the heuristic lookup table as they were needed instead of computing them in advance. This provided a significant improvement to the serial performance of my implementation, but did not scale as well, since workers’ lazy evaluation of the heuristic results in redundant work. However, at 8 processors, the change is still advantageous.

<table>
<thead>
<tr>
<th>Configuration</th>
<th>Time</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>$T_{\text{serial}}$</td>
<td>19.43 sec</td>
<td>1.00</td>
</tr>
<tr>
<td>$T_1$</td>
<td>31.85 sec</td>
<td>0.61</td>
</tr>
<tr>
<td>$T_{8,\text{min}}$</td>
<td>8.20 sec</td>
<td>2.37</td>
</tr>
</tbody>
</table>

**Extension: Back-and-forwards search**

One of the suggestions provided in the assignment was to search both forwards from the initial configuration and backwards from the final configuration, and look for the frontier where the two meet. This should greatly reduce the portion of the search space which must be traversed; we are effectively evaluating something the same size as the “top half” of the search tree twice.

However, the search tree is not nearly complete; it does fan out considerably as board positions open up, making more moves possible at each step, but then as the number of pegs begins to decrease, the number of possible moves decreases again. There is also the matter of synchronization overhead.

The performance improvement from this optimization was roughly proportional even as the number of processors improved, which I didn’t expect (I expected improvement in the single-processor case, and that synchronization overhead would begin to dominate). The improvement was by a factor of approximately four in the `long_test1.in` test case, but became more pronounced on larger, longer problems.

**Extension: Bit-vector board storage**

I also experimented, unsuccessfully, with using bit vectors to pack board information. I was hoping that this would cut down on the overhead of board copying in the parallelized solver, and perhaps even that I might be able to use bitwise operators (that is, “bit tricks”) to accelerate the operations in question.
However, even after spending several hours attempting to optimize my implementation of a bit-vector-based board, it was slower; the initial slowdown was by a factor of approximately 7, which I was ultimately able to reduce to 2.

Part of the complexity of the bit-vector-based solution is that there are three states for each cell instead of only a single one. I did experiment with layouts that used two-bit segments to describe each cell vs. layouts that used two separate bit-vectors to describe which spaces existed and which were filled. The latter was slightly faster.