High Efficiency Computation of Game Tree Exploration in Connect 4

Justin Yokota
Dan Garcia, Ed.
James Demmel, Ed.

Electrical Engineering and Computer Sciences
University of California, Berkeley

Technical Report No. UCB/EECS-2022-219
http://www2.eecs.berkeley.edu/Pubs/TechRpts/2022/EECS-2022-219.html

August 19, 2022

Acknowledgement

Credit goes to Oscar Chan and Isaac Merritt, who helped write the \texttt{openmp} solver as part of a joint CS 267 project, and Gamescrafter members Jatearoon Keene Boondicharern and Robert Shi, who helped write helper functions in \texttt{Connect4.c} and \texttt{memory.c}. This paper was written under the guidance of Dan Garcia.

Finally, a thank you: to my teachers, who taught me how to learn; to my parents, who taught me how to think; and to my brother, who taught me how to teach. To Devin Kushi; to Dana Hagen; to Larry Hada and the JCI; to Tatsuo, Phuclan, and Curtis Yokota; and to the countless others who made me who I am today; thank you.

High Efficiency Computation of Game Tree Exploration in Connect 4

by

Justin Yokota

A thesis submitted in partial satisfaction of the

requirements for the degree of

Master of Science in Computer Science

in the

Graduate Division

of the

University of California, Berkeley

Committee in charge:

Professor Dan Garcia, Chair
Professor James Weldon Demmel, Co-chair

Summer 2022

The thesis of Justin Yokota, titled High Efficiency Computation of Game Tree Exploration in Connect 4, is approved:

Chair _____ Date _____

Co-chair _____ Date _____

_____ Date _____

University of California, Berkeley

High Efficiency Computation of Game Tree Exploration in Connect 4

Abstract

High Efficiency Computation of Game Tree Exploration in Connect 4

by

Justin Yokota

Master of Science in Computer Science in

University of California, Berkeley

Professor Dan Garcia, Chair

Professor James Weldon Demmel, Co-chair

Strongly solving abstract strategy games is generally a computationally intensive task; for large games, computation must be parallelized to complete within a reasonable time. Prior solvers have used tools such as MapReduce to distribute work; however, this approach is hampered by high disk use. This report details the development of a new **shard** solver, which allows for the efficient solving of certain games on large-scale distributed computing nodes, while significantly reducing memory use. A particular target of this project was a fast and memory-efficient strong solve of the game *Connect 4*. Optimizations made in this project, as well as the improved solving paradigm provided by the shard solver, allowed for a solve in less than 6 hours on a 960-node system, while using less than one-eighth of the disk space required for a MapReduce Solve. In addition, a new compression scheme was developed for storing the resulting database, reducing the database size from a naive 32 TiB size to 557 GiB.

Contents

List of Figures

List of Tables

Acknowledgments

Credit goes to Oscar Chan and Isaac Merritt, who helped write the `openmp` solver as part of a joint CS 267 project, and Gamescrafter members Jatearoon Keene Boondicharern and Robert Shi, who helped write helper functions in `Connect4.c` and `memory.c`. This paper was written under the guidance of Dan Garcia.

Finally, a thank you: to my teachers, who taught me how to learn; to my parents, who taught me how to think; and to my brother, who taught me how to teach. To Devin Kushi; to Dana Hagen; to Larry Hada and the JCI; to Tatsuo, Phuclan, and Curtis Yokota; and to the countless others who made me who I am today; thank you.

Chapter 1

Introduction

The GamesCrafters group is a UC Berkeley research and development group focused on computational game theory. One aspect of this group is the development of solvers, which can strongly solve games in reasonable time, and create a database that can be used to play abstract strategy games perfectly. This report details the development of a new **shard** solver, which allows for the efficient solving of certain games on large-scale distributed computing nodes, extending the current single-threaded solvers. A particular target of this project was a fast and memory-efficient strong solve of the game *Connect 4*. Prior solvers were able to solve Connect 4 in two weeks using the *MapReduce* framework on 30 8-core computers, but the long runtime and large database rendered this solve barely within the realm of practicality. Optimizations made in this project, as well as the improved solving paradigm provided by the shard solver, allowed for a solve in less than 6 hours on a 960-node system. In addition, a new compression scheme was developed for storing the resulting database, reducing the database size from a naive 32 TiB size to 557 GiB.

This document will serve both as a summary of the results obtained, as well as an onboarding document to aid new Gamescrafters members and further development of the shard solver. Thus, we will begin with two background sections:

- A section designed to explain material specific to computational game theory and the problem scope. This roughly corresponds to material covered in initial Gamescrafters meetings.
- A section designed to explain lower-level and parallel programming paradigms, which are likely not well-understood by incoming Gamescrafters members. This section assumes as a prerequisite moderate familiarity with a C-like language with explicit memory management, approximately equivalent to UC Berkeley's CS 61C.

After this and an analysis of previous results, we will discuss the project as a whole, before going in-depth in the individual components provided by this project and resulting solve speeds. The final section discusses potential improvements and games that may be newly targeted for solving. Of particular note is the game *Othello*, whose size puts it just

beyond the realm of feasibility with the current solver (with an estimated 16 day solve time on the current solver). An efficient Othello hash is provided, which would allow this game to be solved with the shard solver.

Chapter 2

Background: Game Theory and Connect 4

2.1 Abstract Strategy Games

An **abstract strategy game** is, informally, any game satisfying the following properties:

1. The game state is fully known to all players. No information is hidden or private.
2. No randomness is involved.
3. Gameplay consists of two adversarial players who take turns sequentially.

We additionally restrict our focus to abstract strategy games fulfilling the following additional properties:

4. The game has finitely many states. As a corollary, each player's move choices can lead to finitely many end states.
5. The two players alternate turns. Games where players do not alternate turns can be reduced to this by considering only the end results of any sequence of a given player's chain of moves, or by having the "skipped" player make a "null" move (returning control to the other player). This can be codified within the state, by including a bit representing the player whose turn is next.

We can thus define a game formally as follows:

Definition 1. An **abstract strategy game** is a tuple (G, u, p), where:

- G is a finite directed bipartite graph $G = (U, V, E)$ with nodes in U and V representing game states during which the next move belongs to players 1 and 2, respectively, and E the set of edges between states, where an edge between i and j exists if and only if a valid move exists in game state i that yields game state j

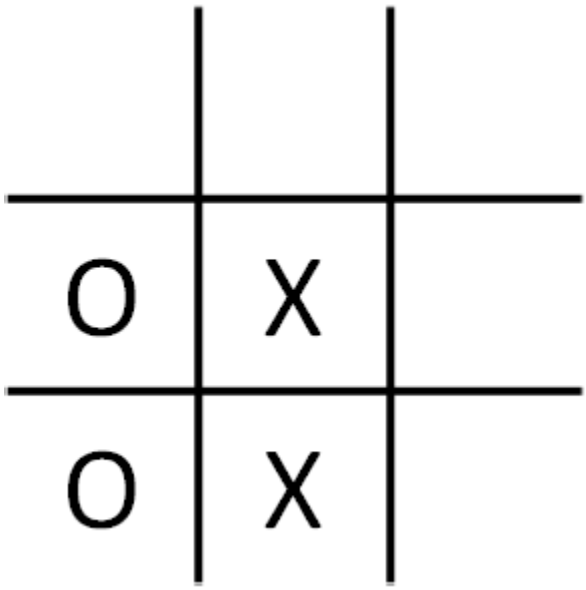

Figure 2.1: Example Tic Tac Toe Position with label `---ox-ox-`

- $u \in U$ is the designated start state of the game
- $p : \{i \in U \cup V, \text{i is terminal}\} \to \{\text{win, loss, tie}\}$ assigns to each terminal state the result (win, loss, tie) of completing a game in this state, relative to the player whose move would have been next. Thus, a terminal position in U assigned "loss" is a loss for player 1, and therefore a win for player 2.

Example. Consider the game Tic-Tac-Toe. Then we can construct the game in this manner as follows:

- U and V are sets of nodes whose states are 9-character strings composed of the three characters "`xo-`". U has exactly one node for each state where the number of "`x`"s is the number of "`o`"s, and V has exactly one node for each state where there is exactly one more "`x`" than "`o`". Edges are added from state i to state j if and only if i's label differs from j's label in exactly one character, the different character in i's label is `-`, and i does not have three "`x`"s or "`o`"s on the same orthogonal or diagonal encoded. Informally, this is referred to as a "3-in-a-row".
- u is the state with label "`---------`"
- p assigns to every position with a valid 3-in-a-row the value "loss", and assigns to all states with 0 "-"s the primitive value "tie". Note that this assigns values to several

states that are unreachable from the starting state; this is innocuous, and can be safely ignored.

Figure 2.1 thus corresponds to a node in U with label `---ox-ox-`. This state has five children, all of which are nodes in V, which correspond to states `x--ox-ox-`, `-x-ox-ox-`, `--xox-ox-`, `---oxxox-`, and `---ox-oxx`. Of these, `-x-ox-ox-` is a terminal position, and is assigned value "loss". Since this position is in V, this position is considered a loss for Player 2, and thus a win for Player 1.

Definition 2. A **primitive position** is a terminal state that is connected to the start state. A **non-primitive position** is a non-terminal state that is connected to the start state. The set of primitive and non-primitive positions together form the set of **positions**. $U \cup V$ form the set of **states**, which may be a strict superset of the set of positions.

2.2 Game Tree Exploration

When considering games, it is often of interest to determine for a given position what the result of that position would be, assuming perfect play from both players.

Definition 3. A position's **result** is one of "win", "loss", "tie", or "draw". A terminal position i's result is equal to $p(i)$. A nonterminal position's result is considered a draw if, assuming perfect play, the game continues indefinitely. A nonterminal position's result is considered a win (respectively, loss, tie) if, assuming perfect play, the game terminates in either a position marked "win" (resp. loss, tie) for the current player or a position marked "loss" (resp. win, tie) for the opposing player.

Definition 4. Perfect play is defined to prefer wins over ties and draws, and ties and draws over losses. Further, wins with low remoteness and losses with high remoteness are preferred. We arbitrarily define perfect play to prefer ties with low remoteness, and therefore to prefer ties over draws.

Definition 5. A position's **remoteness** is the number of moves before the game reaches a terminating state, assuming perfect play, if the current position's result is either "win", "loss", or "tie", and undefined otherwise.

Definition 6. A position's **value** is the tuple composed of its result and remoteness, if applicable. Alternatively, a **win (resp. loss, tie) in i moves** is a position whose value is ("win" (resp. "loss", "tie"), i).

Example. The position in Figure 2.2 has five children: `x--ox-ox-`, `-x-ox-ox-`, `--xox-ox-`, `---oxxox-`, and `---ox-oxx`. Of these, the second move is a primitive loss, and therefore a loss in 0 moves. Thus, choosing this move would assign to the original position a win in 1 move. The other children are a loss in 2 moves, a win in 1 move, a win in 1 move, and a

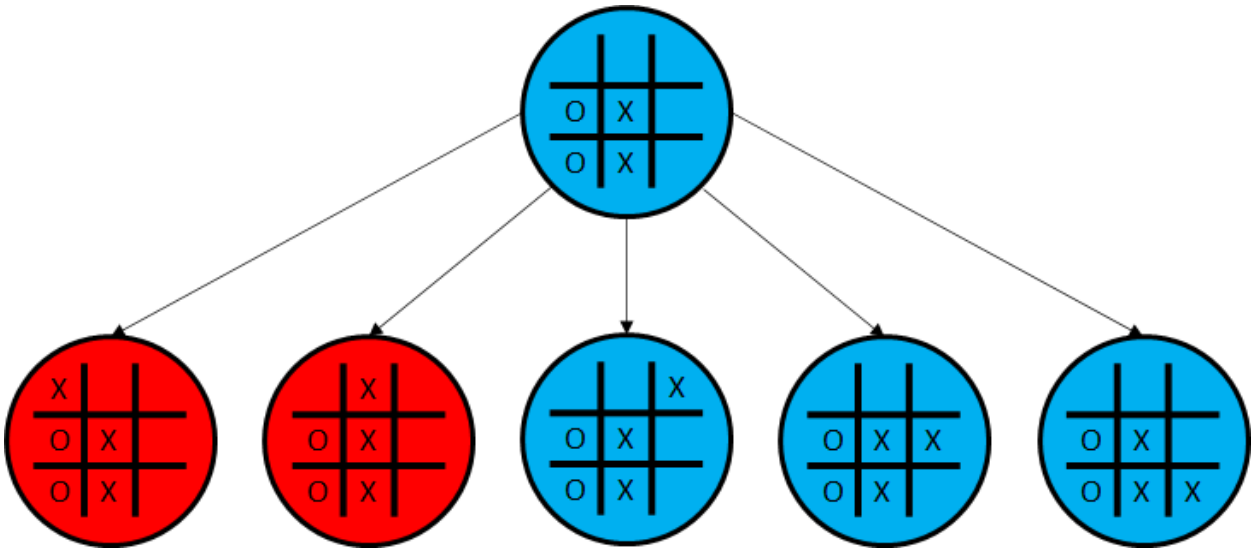

Figure 2.2: Example Tic-Tac-Toe Position with all children moves. Blue circles denote winning positions, while red circles denote losing positions.

win in 1 move, respectively, so choosing these moves would give the original position a value of "win in 3 moves" or "loss in 2 moves". Per the perfect play convention, the second move provides the best result, so we assign this position the value "win in 1 move".

Accordingly, the following recursive definition permits an assignment of values to every position fulfilling the above properties:

Theorem 1. *The following algorithm determines the values of all positions in a game:*

1. *A terminal position's value is assigned its corresponding result with a remoteness of 0.*
2. *Any nonterminal position with at least one losing child is assigned a "win" result. The assigned remoteness is the minimum remoteness among all losing children, plus one.*
3. *Any nonterminal position with no losing child but at least one tieing child is assigned a "tie" result. The assigned remoteness is the minimum remoteness among all tieing children, plus one.*
4. *Any nonterminal position whose children are all winning is assigned a "loss" result. The assigned remoteness is the maximum remoteness among all children, plus one.*
5. *Any position which cannot be assigned according to this is assigned a "draw" result.*

Proof. The perfect play conventions lead directly to the definitions provided for wins, losses and ties, as a position's value is determined by the best available move if either all moves

are determined, or if at least one child has value better than a draw. Any remaining positions must therefore have only losing moves and moves to other undetermined positions. From these positions, it is therefore advantageous to move (indefinitely, for all undetermined positions have at least one move to another undetermined position) toward undetermined positions, and therefore perfect play would continue indefinitely. Thus, these positions are necessarily draws. □

Definition 7. A game is said to be finite-length if its graph of positions is acyclic.

A game that is finite-length necessarily has no positions that are draws, though the converse is not true. For example, a game where the only options are to win the game or to pass to the next player would not be considered finite-length, despite all positions being wins or losses. For games of finite length, the impossibility of draws allows for the computation of a position's value through the following recursive pseudocode:

```
Define positionvalue(position):
    ChildrenResults = []
    If position is terminal:
        Return (p(position),0)
    For child in position's children:
        ChildrenResults.append(positionvalue(child))
    If Loss in ChildrenResults:
        Return (Win, min loss remoteness + 1)
    If Tie in ChildrenResults:
        Return (Tie, min tie remoteness + 1)
    Return (Loss, max remoteness + 1)
```

Definition 8. A program is said to **strongly solve** a game if, given the game as input, it is able to output a mapping from every position to its value within a reasonable time.

Once a game has been strongly solved (and the resulting output stored in a file, generally as a hash table), it is relatively simple to create a program to perfectly play the game, by reading the computed values of all children from the current position and selecting the best move.

2.3 Connect 4

The game Connect 4 is a simple game where players place tokens on a board, with the goal being to place four of their pieces consecutively in an orthogonal or diagonal. Unlike the game Tic-Tac-Toe, Connect 4 has a gravity property; pieces fall downward, such that a piece may not be placed above an empty space. Formally, we can define the game of Connect 4 as follows:

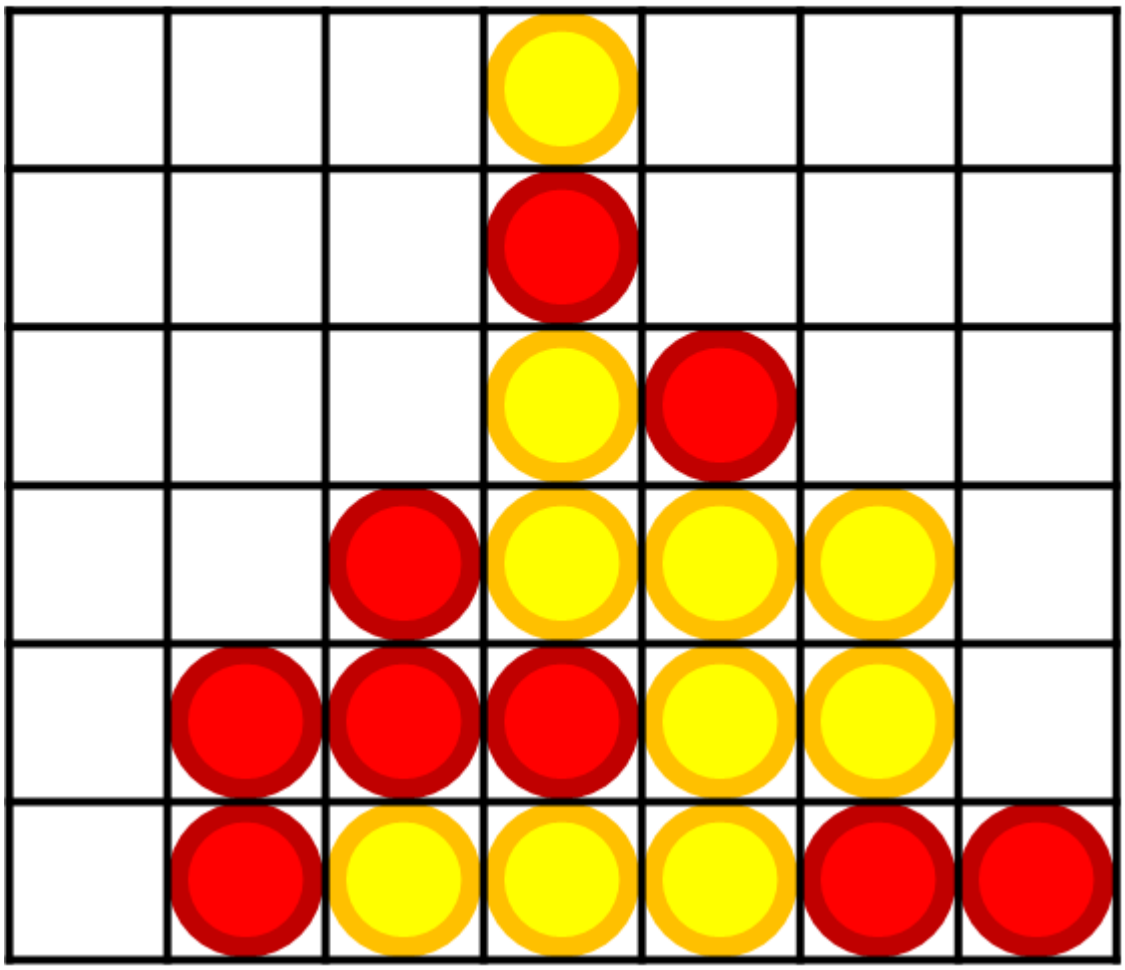

Figure 2.3: Example Connect 4 Position. Red is the next player to move, and can place a red piece at the bottom empty cell of any column except the middle column. This position is a loss in 8 moves; Player 1's ideal move is either column 1 or column 2.

Definition 9. The game Connect 4 [5] is a game parameterized on three values (r,c,n), corresponding to the number of rows, columns and consecutive pieces needed to win, with the original game being (6,7,4) Connect 4. Its state set is defined by an array of c columns, each containing r values among {red, yellow, empty} such that no cell marked empty is after a cell marked red or yellow. Moves are made by converting the bottommost empty cell of a column to a red or yellow cell (red for player 1, yellow for player 2), if the position does not already have a line of n nonempty consecutive cells (horizontally, vertically, or diagonally) with the same value (informally known as an n-in-a-row). As with Tic-Tac-Toe, a primitive is defined as a loss if an n-in-a-row exists, and a tie otherwise – note that it is impossible to play a move in a way to cause your opponent to win immediately.

This game is trivially finite-length, as every valid move necessarily decreases the number of empty spaces by exactly one. This game further has several useful properties that allow for a more efficient solve (described in further chapters), and which can be generalized to several other games.

In addition, generalized Connect 4 allows for periodic benchmarks leading up to the original game, which allows for evaluation of various solvers. The original game has on the order of a few trillion positions, which thus presents challenges both in terms of runtime (a single-threaded solver would take an infeasible amount of time, while parallelizing to a reasonable cluster makes this feasible) and memory use (a naive implementation of a hash table would take half a petabyte, while lossless optimizations get it down to a few terabytes).

Chapter 3

Background: Parallelism and Parallel Computing Paradigms

As a decent portion of this paper discusses the various methods to parallelize a game tree solver, it will be useful to cover a brief overview on some common parallelism paradigms. At its core, the goal of any parallel program is to split a large workload among multiple workers. Ideally, parallelizing to n identical workers reduces runtime by a factor of n; however, several factors prevent this ideal from being reached:

- This ideal is only attainable if all workers receive the same fraction of work. It is often impossible to fully split a large workload evenly, due to some steps that cannot be parallelized and natural variations in processor speed. Single-threaded sections of parallel code end up upper-bounding the maximum speedup through Amdahl's Law.
- The ideal case requires that each workload be fully independent. Dependence between workers at best results in communication overheads, and at worst causes one worker to idle while waiting for another worker. Incorrectly setting up a communication protocol can lead to nondeterministic bugs like data races and deadlocks, which thus increases development costs.

These inefficiencies often scale with both the number of workers and with problem size; thus two common metrics are used to determine the effectiveness of a parallel program:

Definition 10. Strong scaling measures the speedup attained for a fixed problem size as the number of processors increase. **Weak scaling** measures the speedup attained as the problem size scales with the number of processors.

Under the ideal case, strong scaling is linear (that is, parallelizing to n workers reduces runtime by a factor of n), while weak scaling is constant (that is, increasing both workload and the number of workers by a factor of n yields the same runtime), and the efficiency of a parallel program is a measure of how close to these ideals the parallel program attains.

Due to various architectures available for parallel computing, a number of different parallel computing paradigms have been developed to optimally target certain systems.

3.1 OpenMP and Multi-Threaded Computation

The OpenMP library is a library designed for simple access to **multithreaded** programs [16]. Under the multithreaded model, threads are intended to share main memory, with an (ideally small) amount of private memory in individual stacks. This significantly reduces communication overhead, but also limits the maximum amount of parallelism to a single **node**. As a consequence of the shared memory system, data races and false sharing are common bugs; thus, synchronization primitives are often necessary when terminating a multithreaded section.

Syntax

OpenMP is designed to support multiple platforms and has syntax for C/C++ and Fortran, but due to the language of the solver, we will exclusively focus on OpenMP's C syntax.

The primary parallelism primitive is the `#pragma omp parallel` directive, which initializes a multithreaded environment. Code in the following code block is run on all threads; variables initialized outside the parallel block are shared, while variables initialized inside the parallel block are private. In order to differentiate threads, each thread is given a unique thread ID (from 0 to `max_threads`-1), which can be accessed by calling `omp_get_thread_num()` in a multithreaded segment. Similarly, `omp_get_num_threads()` can be called to get the total number of currently-running threads.

Additional directives such as `#pragma omp parallel for` are also available, but they ultimately serve as syntactic sugar to help automate some common multithreaded programming structures; as they are not critical to running an OpenMP program, their specific syntax will be omitted.

An OpenMP program requires the `-fopenmp` flag when compiling through `gcc`, and the `<omp.h>` library in the `.c` file.

Synchronization

The OpenMP library supports **critical segments** through the `#pragma omp critical` directive; only one thread is allowed to run a given critical segment at a time. For more complicated code, the lock ADT is also implemented through OpenMP. A lock is a data structure functioning similar to a bathroom stall lock, fulfilling the following API:

- `acquire()`: If the lock is not held by anyone, acquire the lock (so that the current thread is now holding the lock). If the lock is held by another thread, wait idly until the lock can be acquired.

- `release()`: If the lock is not held by the current thread, this produces undefined behavior. If the lock is held by the current thread, release the lock so that another thread can acquire it.

Locks are often used to create critical segments, since only one thread can run code between a lock acquire and lock release at one point. The lock ADT itself also allows more variation in possible critical segments; for example, it is possible with an explicit lock to protect a shared object over multiple disjoint critical segments, such that only one thread may modify the shared object at a time.

3.2 MPI and Multi-Process Computation

While OpenMP is effective at parallelizing a computation in a shared-memory system, it is not able to extend to systems with distributed memory, such as multi-node systems. For these systems, the **OpenMPI framework** [10] offers a low-level environment for parallel computing across multiple nodes.

The OpenMPI framework distributes work over multiple processes, instead of multiple threads. While the concepts are similar in that both processes and threads represent distinct instruction sequences running in parallel, they differ in that threads are inherently considered part of the same program, and therefore are expected to share the majority of their memory. Different processes are considered entirely different programs, and therefore have fully independent memory segments, and can be run on independent nodes. The OpenMPI framework primarily provides a communication protocol through which nodes can send information in (ideally small, infrequent) packets; this provides a way for an OpenMPI program to coordinate its work.

Syntax

As with OpenMP, we will focus primarily on the C syntax for OpenMPI. Unlike in a multithreaded program, which allows for a fork-join model, OpenMPI programs are fully parallel. Thus, an OpenMPI program starts with `MPI_Init(&argc, &argv)` and ends with `MPI_Finalize()`. As with OpenMP, all code between these two functions gets run on all processes. Processes can individually call `MPI_Comm_size()` and `MPI_Comm_rank()`, which provide the total number of nodes and the current process's ID, respectively.

Programs written with OpenMPI are compiled with the command `mpicc`, which acts similarly to `gcc`. Unlike OpenMP, an OpenMPI program must be run through the command `mpirun`; this command helps set up the OpenMPI environment, and allows setting the number of nodes. Since processes are more independent, the number of cores available in an MPI framework is limited only by the cluster in use; it is thus recommended to run MPI programs primarily on the Savio clusters or similar large-scale compute clusters, instead of locally. Section 3.5 will go into further detail on the Savio cluster, and an example run will be provided in Appendix B.3.

Communication

One important thing to note is that in an MPI framework, communication is extremely expensive, relative to standard operations. Data transfer per byte is often orders of magnitude slower than floating point operations, and initializing a transfer is often orders of magnitude slower than that. Thus, each transfer incurs a significant time penalty; ideally, messages are as infrequent as possible, and, assuming equal frequency, as short as possible. The MPI framework offers several options for communication, but the most basic options are the `Send` and `Recv` functions, defined as follows (function inputs are simplified from the original C functions):

- `MPI_Send(Message, Destination)` sends the given message to the process specified by `Destination`. In order for this to work, the destination process must be expecting this message; thus, the source process idles until the destination process runs a corresponding `Recv` call.
- `MPI_Recv(MessageType, Source)` sets the current process to expect a message of the given type from the process specified by `Source`. As with `Send`, the current process idles until the source process runs a corresponding `Send` call.
- `MPI_Isend` and `MPI_Irecv` act similarly to the above two, except the process does not idle while waiting for the message to send/be received. This still requires the other process to perform a corresponding `Recv/Send`; thus, the message is not completed until then.

Under most circumstances, both `Send` and `Recv` must specify the destination/source process to which communication will occur. However, a `Recv` is allowed to specify a source process of `MPI_ANY_SOURCE`, which allows the current process to receive a message from any source. `Recv` also creates a status object containing, among other metadata, the source process ID; this can be used to respond to a message received from an `ANY` source. This allows an MPI process to act as a manager in a manager-worker model, where one process coordinates the tasks of workers, assigning additional work as workers complete their tasks.

It can be difficult to develop an intuition for communications protocols; in my experience, task coordination tends to be one of the harder topics for undergraduates, even when working with the simpler OpenMP model. The Zachtronics video game "TIS-100" [6] is an excellent resource for developing this intuition. Primary gameplay revolves around programming the titular "TIS-100" computer to solve simple problems. The TIS-100 consists of twelve subprocessors arranged in a grid, which can each run a highly simplified assembly language. While early levels are solveable on a single processor, resource constraints quickly force the player to use multiple processors and develop effective communication protocols between processors. Since this is a commercial game, the difficulty curve is well-balanced to teach a nonspecialist these skills incrementally, and provide a continuous sense of progress. This can thus serve as a viable, if unconventional, means of onboarding new undergraduates to a distributed-programming research group. For students less-versed in assembly languages,

the game "7 Billion Humans" [3] offers similar gameplay using a block-based programming language.

3.3 MapReduce

As the number of processors in a parallel cluster increases, it becomes increasingly harder to evenly distribute work with minimal overhead. The **MapReduce** framework thus offers a standardized framework through which a program can be parallelized to a large cluster.

The premise of the MapReduce framework [8] assumes a near-infinite number of parallel processors, and relies on restricting a program to a specific set of operations that parallelize well under this assumption. In particular, the MapReduce framework focuses on the following core loop:

- Map operations receive as input a large array `[a0, a1, a2, ···]` and a function f, and outputs `[f(a0), f(a1), f(a2), ···]`. Assuming that f has no side effects (that is, f outputs the same value for a given input regardless of program state and doesn't affect anything outside f), it is possible to split work to a large cluster by assigning one array element per processor. Under the MapReduce framework, f is allowed to output multiple values; these values then get concatenated to form the final output list. Further, f's output values tend to be (key, value) pairs, with the intent being to produce unique outputs per key.

- After a map step, an implicit sorting step optionally occurs, which partitions the array of (key, value) pairs outputted by a map step according to their key. This creates a list of values associated with each key. This sorting can be split over a large cluster, by sorting subsets of the (key, value) pairs in each processor.

- Reduce operations receive as input a list of values associated with a given key, and outputs a list of result values. This tends to be a "combining" step, with output generally being a single value representing the sum (or product, or concatenation) of input values. This can also be efficiently parallelized over a large cluster, since each unique key can be assigned to a separate processor.

The map and reduce functions are provided by the programmer; ideally, these functions are short, independent of computation order, and have low runtime and memory costs. Often, multiple map-reduce tasks are cascaded to translate the original dataset into a target result. Not all programs are able to be translated into a MapReduce format; however, those that can be rewritten in this framework tend to get high parallel efficiency from this approach.

3.4 GPU programming

Similar to how MapReduce provides a restricted framework through which a program may be efficiently distributed a large number of nodes, **GPUs** offer a restricted framework for distributing a workload over a large number of threads. The GPU is a processor originally designed for graphics processing; graphics tend to require large amounts of linear algebra, so GPUs are specifically designed for parallel mathematics. Thus, a GPU contains hundreds or thousands of simplified cores, well more than the 10-20 normal cores available on a standard node. Through libraries such as CUDA [18], a program can be run partially on a GPU, taking advantage of the large number of cores available.

As a caveat, these GPU cores are not fully independent; instead, multiple threads combine to form **warps**, which share a program counter. In the ideal case, code intended for GPUs have no branches; in this case, all threads on a GPU warp run the same instruction at the same time. For code with branches, the program counter increments according to the slowest thread in the warp, with threads "skipping" instructions that don't apply to them. In particular:

- For if-else statements, threads end up running through both cases, ignoring the case that doesn't apply. This generally means that it is hard to differentiate threads in a warp; if a thread is supposed to either run code A or code B on a value, the effective runtime penalty would be the sum of the runtimes of code A and B.
- For loops, threads end up delayed according to the thread with the most iterations. This generally makes loops with no upper bound on their iteration count impractical.
- If one thread is forced to idle (perhaps due to waiting for a lock acquire), all other threads in the same warp also idle. This severely limits the ability of GPU programs to run synchronization primitives without risking deadlock. Thus, GPU programs are generally most effective when performing tasks on independent data values.

Warps also share L1 caches, which, contrary to most multithreaded frameworks, runs faster on interleaved memory accesses than on blocked memory accesses.

Given the constraints on GPUs, this approach is generally limited to map operations and simple programs. However, as a shared-memory system (albeit one with limited memory space), a GPU program would generally be more efficient than a similar problem applied to a similar-sized MPI cluster.

3.5 The Savio Cluster

Berkeley's **Savio Cluster** [1] was the high-performance computing facility used for this project, and therefore the architecture targeted by this solver. This section will detail the resource configuration of the cluster, as well as a general usage guide.

Resource Configuration and Limitations

The Savio cluster is divided into multiple **partitions**, with each partition roughly corresponding to a specific use case and a specific upgrade of the cluster. Code can be run on a given partition by submitting a **job** to that partition; a scheduler then schedules an allocation of the requested resources, or rejects the job if it exceeds resource limits. Most jobs for this report were run on the `savio3` partition, which consists of:

- 112 32-core nodes with a 2.1 GHz Intel Xeon Skylake 6130 CPU model and 96 GB RAM
- 72 40-core nodes with a 2.1 GHz Intel Xeon Skylake 6230 CPU model and 96 GB RAM

Other partitions available include the `bigmem` and `xlmem` partitions (which have 384 GB and 1.5 TB of RAM, respectively), the `htc` partition (which can be reserved per-core rather than per-node), and the `gpu` partition (which gives access to 4352-core GPUs). Except the `htc` and `gpu` partition, all jobs reserve full nodes; for example, a single-threaded program run on the `savio3` partition would still receive a resource allocation of an entire 32-core node.

For jobs that are not submitted by Condo users (who contribute nodes to the Savio cluster), resource constraints are imposed on individual jobs to ensure that no job reserves the entire partition. Most jobs have a resource limit of three days and 1000 cores (the latter empirically determined; the 1000-core limit is frustratingly not documented on the Savio website), while long-running jobs can be run with a ten-day and four-core limit.

Further, there is a per-year limit of 300,000 **service units** allocated per project. Each service unit corresponds roughly to one core-hour, with older partitions having a lower cost (ex. 0.5 service units per core-hour for the `savio` partition) and resource-intensive partitions having a higher cost (ex. 2.67 service units per core-hour for the `savio3_bigmem` partition). Thus, a six-hour job using 960 cores on the `savio3` partition (which consumes 1 service unit per core-hour) would consume $6 * 960 = 5,760$ service units, or about 2% of the yearly allocation.

For short-term disk storage, there is additionally a shared 3.5 PB disk available in the `/global/scratch/users/` directory (known as the **scratch** directory), which can be used to store temporary files. As files in this disk get purged after 120 days without an access, this should be considered temporary storage; end products should be transferred to an external disk once a job or set of jobs complete [2].

Usage Guide

A job script can be created as a bash script, prepended with cluster-specific information. For example, the following bash script (`mpi-run.sh`) was used to run the Connect 4 solver:

```
#! /bin/bash
#SBATCH --job-name=connect4
#SBATCH --account=jyokota
```

```
#SBATCH --partition=savio3
#SBATCH --ntasks=960
#SBATCH --time=24:00:00

mpirun -n 960 ./build/connect4mpi.exe workingfolder
```

The commented lines beginning with `SBATCH` correspond to job configurations; this creates a job named "connect4", under the account "jyokota", using 960 nodes on the `savio3` partition, with a one-day runtime limit before the job is automatically terminated.

This job can be submitted using the command `sbatch mpi-run.sh`, which creates a job and assigns it a job ID. Job progress can be checked using the command `sq -u <username>`. Once a job begins running, output (such as through print statements) is piped into the file `slurm-<jobID>.out`. All components in this project were designed to output progress reports periodically as the solver runs; thus, checking the current state of a job's corresponding `.out` file can provide useful estimates on how far a job has completed. For continuous reading (similar to how one might see output if the program was run locally or on an interactive node), the command `tail -f <filename>` can be used to monitor the output file as it gets written.

Chapter 4

Background: Prior Work on the Strong Solving of Games in General and Connect 4 in Particular

The Gamescrafters group and several other groups had already separately developed solvers capable of strongly solving Connect 4 and other abstract strategy games. The following chapter details their individual results towards this.

4.1 Tiered Games

One useful property of Connect 4 is that it is a tiered game, defined as follows:

Definition 11. A **tiered game** is a game of finite length in which the set of positions can be partitioned into sets $0, 1, \cdots, n$ such that the starting position is in set 0, and such that for all positions in the set marked i, all children positions are in the set marked $i + 1$. If such a partition exists, then it is unique. In this case, each set is referred to as a tier, and a position is said to be in tier i if its associated set in this partition is assigned number i.

Tiered games lend themselves well to large-scale parallelism, as positions within the same tier can be computed independently.

4.2 Tiered Solvers

The overall structure of the tiered solver involves computing each tier as it becomes viable. Solving can be divided into two distinct phases per tier; a **discovery** phase, and a **solving** phase. During discovery, all positions in a tier are expanded to determine their children. Thus, discovering tier 1 creates a list of all positions in tier 2. During solving, all positions in the tier are computed, based on the solved results of the subsequent tier. Thus, solving

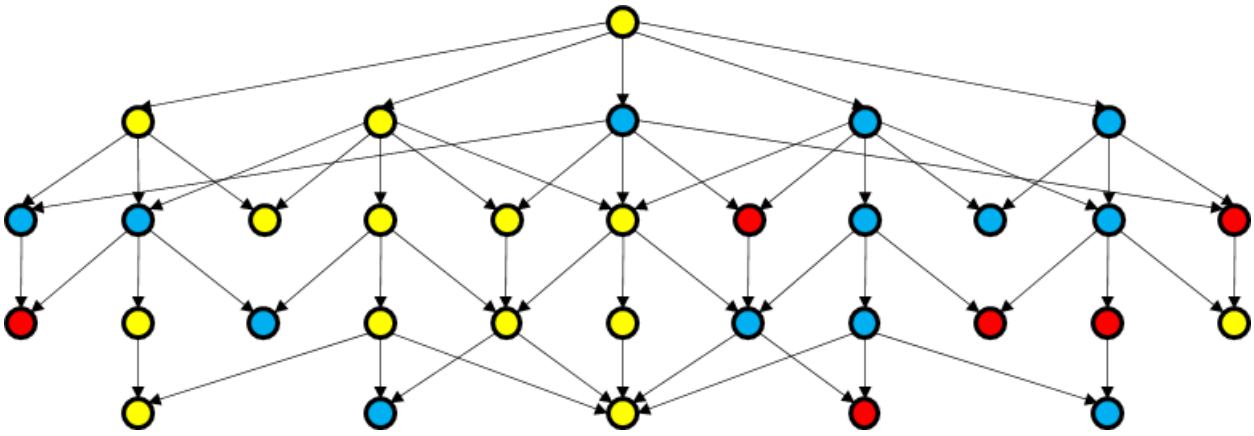

Figure 4.1: Example tiered game. Each position has children only in the row beneath it; thus a solver can compute all values in a tier independently, using information from the previous and next tier. Blue, red, and yellow circles denote wins, losses, and ties, respectively, with primitive results assigned randomly.

tier 1 assigns to each position in tier 1 a value, based on an already-solved tier 2. In general, tier $i-1$ must be discovered before tier i is discovered, and tier i must be solved after both tier $i-1$ is discovered, and tier $i+1$ is solved (if they exist). The following pseudocode thus details the general outline of the tiered solver:

```
Define SolveGame():
    Let n = Max Tier
    Let tier 0 = [Starting Position]
    For i in [0,1,2,3,...n-1]:
        Let tier i+1 = Discover(tier i)
    Let tier n+1 = []
    For i in [n,...,0]:
        Solve(tier i, tier i+1)
    Save results to file
```

Discovery and Solving can both be written in MapReduce fairly simply, using the following pseudocode:

```
Define Discover(tier):
    Let children = Map(tier, position -> List of children of that position)
    Let newtier = unique(children)
    return newtier
Define Solve(currenttier, nexttier):
    Let results = Map(currenttier, position ->
```

```
        (position, positionvalue(position)))
    return results
```

4.3 Previous Solvers

Specialized Solvers for Connect 4

(6,7,4) Connect 4 was first **weakly solved** in 1988, independently by James Allen [4] and Victor Allis [5]; using various heuristics, the game's start state was proven to be a winning position, and an explicit strategy was provided to guarantee this win. At this time, a brute-force strong solve was deemed impractical, due to computation limits of the time.

A strong solve of (6,7,4) Connect 4 was completed by John Tromp [21] in 40000 hours in 1995, but his methods still required heuristics specific to Connect 4, which do not generalize well to other games. This solver became the basis of the Fhourstones benchmark [22], which can solve a few million positions per second on modern single-core machines.

Gamescrafters Solvers

Gamescrafters has previously created multiple solvers [11], with varying degrees of generality. Two particular solvers are relevant to solving Connect 4; the generalized C solver (under the name **Gamesman Classic**), and the tier-based Java solver (under the name **Gamesman Java**).

Gamesman Classic [12] is a C-based single-threaded solver written in 1999 by Dan Garcia, and iterated continuously to this day; it is able to strongly solve (5,5,4) Connect 4 in a few seconds using a Connect 4 implementation written by Michael Thon. This solver is designed to be able to handle games which are not acyclic, and therefore can be applied to most games. However, due to being single-threaded, it is ultimately unable to solve (6,7,4) Connect 4 within a reasonable amount of time or memory footprint.

Gamesman Java [13] is a tier-based solver written in Java to take advantage of Hadoop MapReduce libraries, written by David Spies in 2010. This was the first Gamescrafters solver capable of solving (6,7,4); however, a combination of Java inefficiencies and large data transfers caused this to take approximately 2 weeks running on 30 machines with 8 cores each.

4.4 A Memory-efficient Connect 4 Hash

In order to properly record position values, it is useful to assign to each position a unique numeric identifier, commonly known as a hash. In this case, a position's hash is ideally short (as the hash length determines overall memory use), easy to compute (as hashing game positions constitutes a significant portion of runtime), and easy to revert (as hashes

tend to be shorter and therefore more efficient to communicate between parallel nodes). The following hash has these properties:

Definition 12. The Connect 4 hash h : (r,c,n) Connect 4 states $\rightarrow \{0,1\}^{c*(r+1)}$ is defined as follows: Each column gets stored as a sequence of $r+1$ bits. The least significant bits of this sequence correspond to the colored pieces in the column; a red cell is assigned bit 0, and a yellow cell is assigned bit 1. The next bit is assigned 1, while all remaining bits in the column are assigned 0.

Example. In the following examples, "`-`" is used to denote an empty cell, while "`r`" and "`y`" are used to denote a red and yellow cell, respectively. In figure 4.2, the sequence "`------`" is assigned the bit pattern "`0b000001`", as 0 spaces are filled. The sequence "`---rry`" is assigned bit pattern "`0b0001001`", because the last three spaces are filled, and correspond to binary "`0b001`". The sequence "`yryyry`" is assigned bit pattern "`0b1101101`", as this column has 6 filled spaces, which correspond to the last six bits of this pattern.

Example. The full Connect 4 board is stored by concatenating the representations of each individual column. As such, the overall representation of the starting board in a (6,7,4) Connect 4 game is "`0b0000001 0000001 0000001 0000001 0000001 0000001 0000001 = 0x00408 1020 4081`".

This hash has a number of useful properties, which inspire solving strategies. Further discussion is presented when relevant, but for now, the following properties are presented:

Theorem 2. *This hash is injective over the position set of any given Connect 4 game.*

Proof. In order to demonstrate that this hash is easy to revert, we will define the inverse hash algorithm. Split the hash value into columns. By the hash's definition, there is necessarily at least one bit that is 1 in this column; note the position of the most significant on bit. For each bit from the least significant bit to the most significant 1 bit (not including this bit), select a yellow cell for every 1 bit, and a red cell for every 0 bit. Fill the remainder of the column with 0 bits. □

Theorem 3. *This hash is monotonic; if a move exists from position a to position b, then* $h(a) \leq h(b)$.

Proof. Adding a piece to a column moves the most significant one bit in that column by one; therefore, the numeric value of that column increases. Since every move consists of adding a piece to exactly one column and keeping all remaining columns the same, this necessarily increases the value of the hash. □

Note: This hash is memory-efficient; with the exception of the 0 value, every bit sequence corresponds to a valid column, so almost every hash corresponds to a valid state of Connect

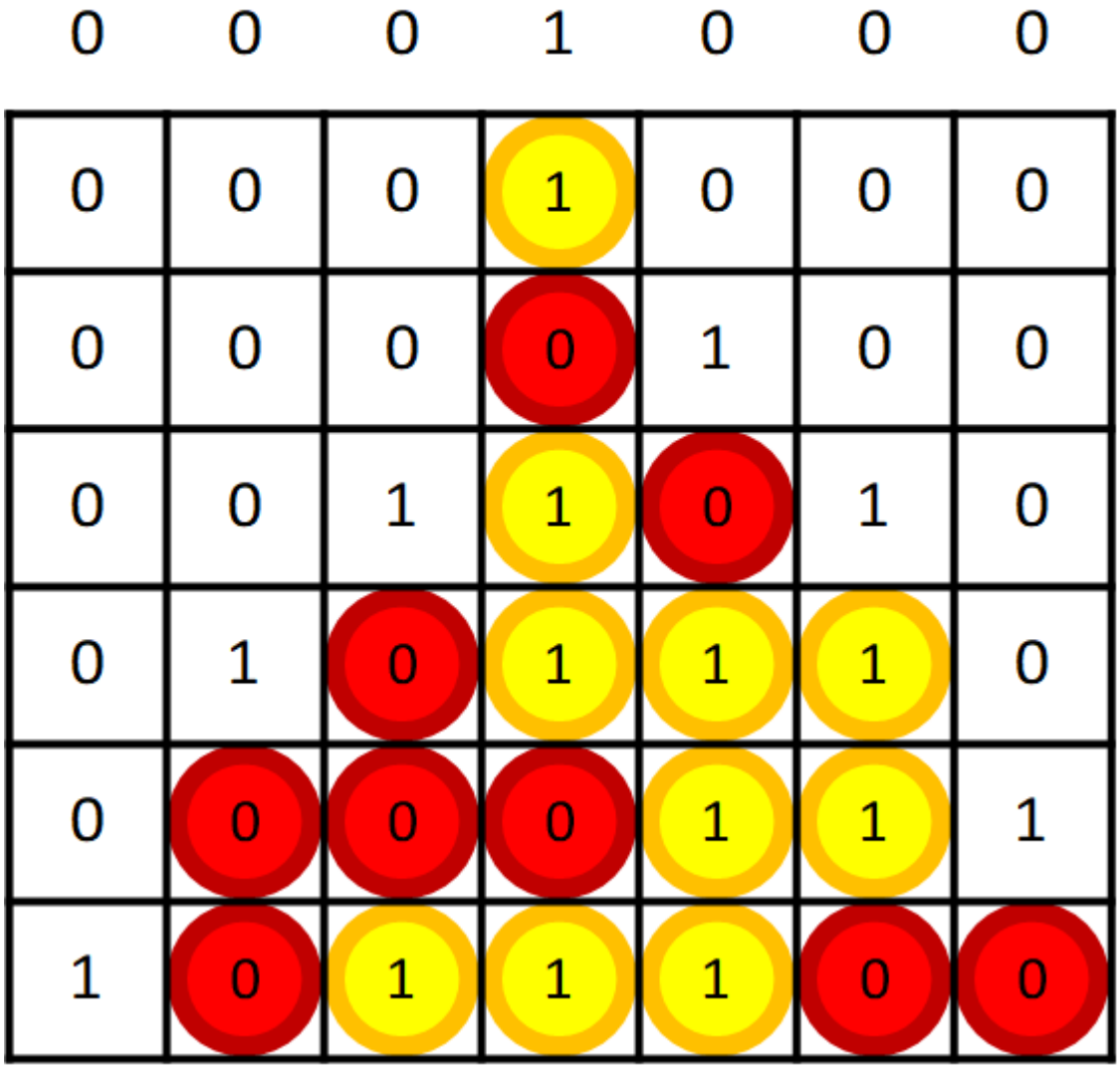

Figure 4.2: Connect 4 board annotated with bit values. The hash of this board is thus `0b0000001 0000100 0001001 1101101 0010111 0001110 0000010 = 0x0 0420 9DA5 C702`

4 pieces [1]. Improvements can be made by taking advantage of the fact that only a small fraction of possible states are actually reachable positions. Specifically:

- In all valid positions, the number of red and yellow pieces are balanced; either the number of red cells is exactly equal to the number of yellow cells, or there is exactly one more red cell than yellow cells. Of the 2^{2n} states with $2n$ filled cells, the number of states which fulfill this property is equal to $\binom{2n}{n}$, which can be approximated by $4^n/\sqrt{\pi n}$ through Stirling's Approximation. Thus, the fraction of states with $2n$ filled cells that are balanced is approximately $1/\sqrt{\pi n}$. For (6,7,4) Connect 4, the average state has 35 filled cells, so the approximate fraction of all states that are balanced is around $1/\sqrt{35\pi} \approx 1/8$. Thus, fully taking advantage of this property could in theory save up to three bits from the original 49-bit hash. Even without fully taking advantage of this, a low-effort optimization would be to omit the last bit of the above-defined hash; if both choices of the last bit yield valid states, changing the last bit from a 0 to 1 (or vice versa) changes the value (number of red cells - number of yellow cells) by two. Thus, at most one choice of last bit would be sufficiently balanced (with red-yellow either 0 or 1), so at most one choice of last bit yields a valid position. As such, omitting the last bit of the above hash still allows each position to be assigned a unique hash.

- In all valid positions, all n-in-a-rows must intersect at the most recent move; further, there must exist a sequence of alternating red-yellow moves ending in the most recent move that yields the state. For (6,7,4) Connect 4, there are a total of 4,531,985,219,092 positions [21]. Thus, in theory, a perfect hash could store a Connect 4 position in 43 bits, saving six bits in total from the defined hash.

Due to the specific memory compression algorithm used, and due to the slight increase in computation time this would cause for hash reversals, the one-bit reduction described above was not implemented in the remainder of this project.

[1] For (6,7,4) Connect 4, $(127/128)^7 \approx 95\%$ of all hashes correspond to a valid state

Chapter 5

Overall Project Structure

The primary goal of this project was to write a more efficient solver capable of solving Connect 4, in a manner that allows extension to further games. As such, the solver was rewritten entirely in C, with separate modules for the game and memory module. This section will detail the files currently written, along with the intended interface of the game and memory modules.

5.1 Solvers

Two types of solvers are provided: a single-threaded, **Stack-Based Solver** (described in Section 6.3), and a **Shard-Based Solver** (described in section 7.4). These solvers serve as the main entrypoint for these programs.

Stack-based Solver

Two versions of the Stack-Based Solver are provided:

- `solversinglethreadednomemorytest.c` runs the Stack-Based Solver on a single thread, and outputs data on the game; it is intended to verify correctness of newly written games.
- `solversinglethreaded.c` runs the Stack-Based Solver on a single thread, then attempts to save and reload the results to a file, and compares the reloaded values to actual results. This is primarily intended to verify correctness of the memory module. Its command-line interface receives as input a filename, which is used to save and retrieve results. The code for this file is provided in Appendix A.1.

Shard-based Solver

The shard solver ultimately divides the game into subcomponents, referred to as **shards**. The exact details of this (along with the terminology used) will be explained in section 7;

for now, the interfaces used will be presented here.

The shard solver is divided into two parts; the **main driver**, which is responsible for managing which shards get solved, and the **shard solver**, which is responsible for solving individual shards through the `solver.h` interface. Both the main driver and the shard solver have several versions available currently, which can be mixed and matched as needed.

Main Drivers

:

For all main drivers, the program receives from the command line a path to a folder, in which all results and intermediate computation steps will be stored. The code for these files are provided in Appendices A.2, A.3, and A.4, respectively.

- `maindriversinglethreaded.c` is used to run the solver using a single thread.
- `maindriveropenmp.c` is used to run the solver using a single process, with multiple threads in an OpenMP model.
- `maindrivermpi.c` is used to run the solver with MPI, and can thus be scaled to a large number of processes.

5.2 `solver.h`

`solver.h` defines the `shardgraph` data type, which defines a single node in the directed **shard graph**. Most components are designed to fulfill the interface of a directed graph; in addition, the node includes an integer value `discovered` noting the current solve status of that node (0 before discovery, 1 before solving, 2 after solving), and a `shardgraph` pointer `nextinqueue` used to implement a linked list for the work queue.

Any implementation of solver.h must implement the following functions for creating a shard graph:

- `shardgraph* getstartingshard(shardgraph* shardlist, int shardsize);`
- `int initializeshardlist(shardgraph** shardlistptr, uint32_t shardsize);`
- `void freeshardlist(shardgraph* shardlist, uint32_t shardsize);`

$\cdots$ and must implement the following functions for discovering and solving a shard, respectively:

- `void discoverfragment(char* workingfolder, shardgraph* targetshard, char fragmentsize, bool isstartingfragment);`
- `void solvefragment(char* workingfolder, shardgraph* targetshard, char fragmentsize, bool isstartingfragment);`

The code for `solver.h` is provided in Appendix A.5. Two versions of the shard solver are provided:

- `solvermpiinitial.c` runs a modified version of the single-threaded stack-based solver on a single shard when running through discovery or solving, and saves both its data transfer and its results to the working folder under file prefixes "transfer" and "solved", respectively.
- `solver.c` does the same, but includes an optimization (discussed in Section 8) that significantly reduces transfer requirements, at a slight additional runtime cost. This code is provided in Appendix A.6

5.3 `Game.h`

`Game.h` defines the interface needed for a game to be solved by the shard solver. It additionally defines several constants associated with the position value encoding described in section 7.2, as well as a docstring fully defining the encoding. The solver defines two ADTs:

- The `game` ADT stores all information about the current state (including the current person to play), and is required to have a distinct value for every valid position. By default it is set to be an unsigned 64-bit integer.
- The `gamehash` ADT is used to store the hash of a given position. By default it is set to an unsigned 64-bit integer.

In addition, `Game.h` requires the following functions defined per game:

- `void initialize_constants()` is run once at the beginning of the program (from all processes when applicable), and is intended for precomputing values.
- `game getStartingPositions()` returns the start position of a game.
- `int getMaxDepth()` and `int getMaxMoves()` returns the greatest depth (defined as the longest sequence of moves possible within a game) and the maximum number of moves available from any position, respectively. For finite games, these are necessarily finite; further, the solver works most efficiently when their product is small.
- `int generateMoves(char* retval, game position)` returns the number of moves available from a given position. Retval is set to a sequence of chars, each representing one of the valid moves available.
- `game doMove(game position, char move)` returns the game state that results from applying the given move (whose identifier was returned from `generateMoves`) on the given position. Note that this function is separate from the `generateMoves` function in order to better interface with a game player, and to allow use of the most recent move in determining if a position is a primitive.

- `char isPrimitive(game position, char mostrecentmove)` returns if a given position is a primitive, assuming the most recent move was as stated. It is assumed that a given position will always have the same value (that is, if two different moves could yield the same position, sending either value as `mostrecentmove` will result in the same output), but it is considered undefined behavior if the input `mostrecentmove` does not correspond to a valid move that yields this position. `isPrimitive` returns WIN, LOSS, or TIE on a primitive (depending on its result), and NOT_PRIMITIVE otherwise.

- `gamehash getHash(game position)` returns the hash of a given position. `gamehash maxHash()` and `int hashLength()` return an upper limit on the maximum hash, and the length of hashes in bits, respectively. Thus, for all positions p, getHash(p) $\leq$ `maxHash()` $< 2^{\texttt{hashLength()}}$.

- `game hashToPosition(gamehash hash)` returns the game corresponding to the given hash. It is considered undefined behavior if `hash` does not correspond to a valid hash.

- `int getchildrenshards(uint64_t** childrenshards, char shardsize, uint64_t parentshard)` returns the number of children shards of the shard with the given shard ID (of the given shard size), and sets `childrenshards` to an (allocated) array of children shard IDs.

The code for `Game.h` is provided in Appendix A.7. Currently, Connect 4 is the only game fully implementing this interface (code provided in Appendix A.8). However, a number of games will be discussed in section 10.2 that can likely be efficiently solved by this solver.

5.4 memory.h

`memory.h` contains the **hashtables** needed to solve games. The exact details are discussed in section 7, but overall, two versions are provided; a **solver-side hashtable** which allows for reads, writes, and contains operations, and a **player-side hashtable** which only allows reads. These are defined by the ADTs `solverdata` and `playerdata` respectively.

- `solverdata* initializesolverdata(int keylen)` initializes an empty solver-side hash table which can contain hashes of length `keylen`. Each hash can store up to one nonzero byte, corresponding to the value of a position. Likewise, `freesolver(solverdata* data)` frees the solver.

- `solverinsert` and `solverread` act as insertion and reads from the hashtable, with reading returning 0 if the hash has not been stored yet.

- `void solversave(solverdata* data, FILE* fp)` saves the hashtable to the given file. Note that this saving need only allow for restoration of the `playerdata`, not the original `solverdata`.

- The functions `initializeplayerdata`, `playerread`, and `freeplayer` act analogously to their solver counterparts, with `initializeplayerdata` defining a player-side hash table from a saved file instead. Note that `playerdata` does not support writing, as the solver alone is expected to fully define the hash value.

- `bool verifyPlayerData(solverdata* sd, playerdata* pd)` is a simple test function designed to confirm the correct working of the memory system. Formally, it returns true if `solverread` and `playerread` yield the same results on all values saved in the solver.

The code for `memory.h` is provided in Appendix A.9. Several memory modules are provided with the code, as follows:

- `memorynaive.c` and `memoryoptimized.c` provide early precursors of the current memory system; they are provided as ways to test systems without a complex memory module, and correspond to the array approach.

- `memorytier.c` and `memorylessoptimizedmanualtier.c` are experimental forms which as of current produce less efficient results. However, further development may find these approaches useful.

- `memorymoreoptimized.c` and `memorywithplayer.c` implement the tree approach, with logarithmic time accesses.

- `memoryfastretrieval.c` currently represents the most efficient system, allowing constant time accesses and better compression at the cost of initial overhead and RAM memory usage. The code for this module is provided in Appendix A.10

5.5 Compilation Scripts

Due to issues with makefiles on my local computer, compilation is done through .sh files. All compilation results get sent to the build folder. The code for these files are provided in Appendices A.11, A.12, A.13, and A.14, respectively.

- `maketestmemory.sh` compiles the memory tester, which can be used to test if the memory system is consistent.

- `makesinglethreaded.sh`, `makeopenmp.sh`, and `makempi.sh` compiles the solver using their corresponding main driver. The former two can be run directly, while the latter can be run on the Savio clusters using the script `mpi-run.sh`.

Chapter 6

Single Threaded Improvements to Connect 4 computation

Before parallelizing the game solver, several optimizations were developed for single threaded solving. This was primarily focused on three areas: the algorithms for playing Connect 4, the storage system of a position, and game tree exploration using a stack.

6.1 Algorithmic Improvements to Connect 4

The original Java solver defined a Connect 4 board as a 2D array of pieces, which was both memory-inefficient, and which incurred significant time costs when hashing and unhashing. A large optimization here was having a `game` struct extremely similar to that of the `gamehash` struct, in order to minimize hashing and unhashing times. As it turns out, the Connect 4 hash defined earlier allows for efficient implementations of game functions applied directly to the hashes.

For any given position, the available moves correspond to the most significant 1 bit of every column, which are not the most significant bit of their column. We associate to these moves their bit index for the purposes of `generateMoves`.

For a move n from the position with hash h, the child position has hash $h + (1 << n)$ for red moves, and $h + (1 << (n + 1))$ for yellow moves. Further, since the bit pattern effectively mimics a 2D array of bits, all wins correspond to one of four fixed bit patterns (corresponding to the four possible directions of a Connect 4), which can be precomputed, and checked around the most recent move. Ties can be easily checked by checking if all bits on top are 1 (and there are no wins).

Thus, a game's internal state is stored as its hash (assumed to be 63 bits or shorter), with the most significant bit instead tracking the current player's turn (for fast computation). This isn't the most efficient option in any individual metric, but overall balances runtime of unhashing and hashing, move computation, and the memory footprint of a game.

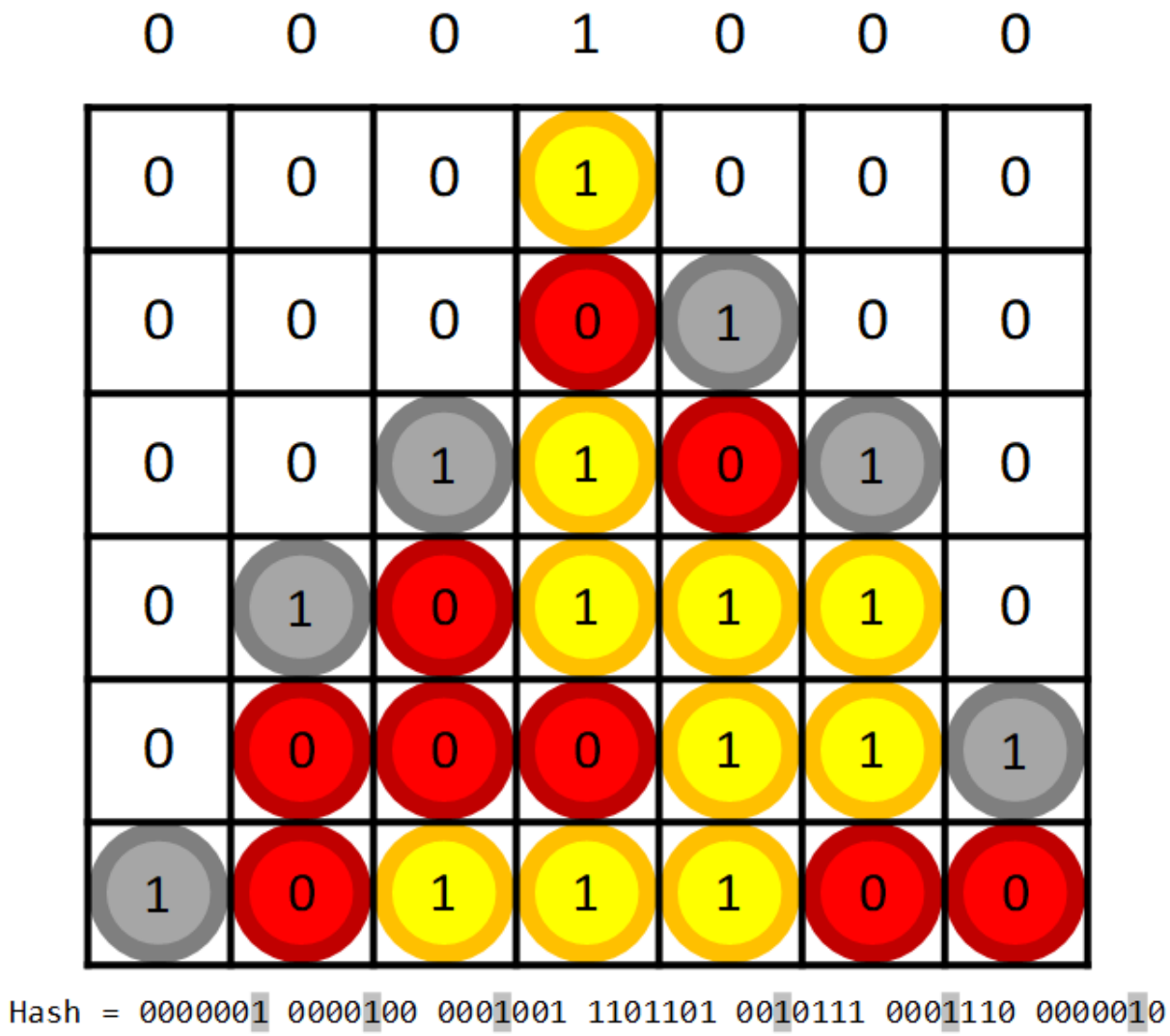

Figure 6.1: In this position, the grey pieces denote possible moves for red. When mapped to the hash, they correspond to the leading 1 bits of their column (except for full columns). These moves are assigned numbers 42, 37, 31, 18, 10, and 1, from left to right, corresponding to their bit index

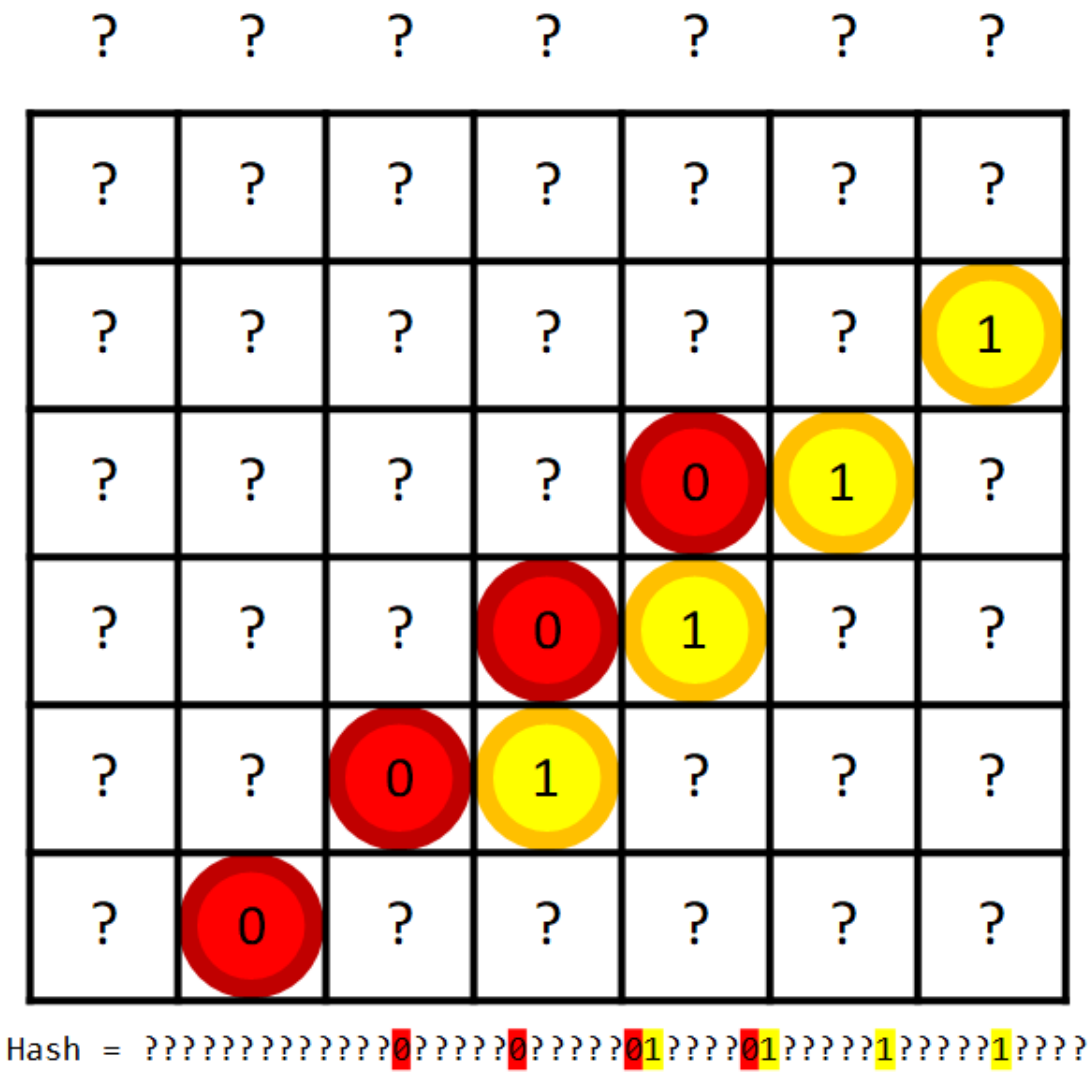

Figure 6.2: Two diagonal wins in Connect 4. When mapped to this position's hash, the wins form a consistent bit pattern, which can be precomputed. Similarly, wins in the other three directions form consistent bit patterns

Example. The starting position in $(6, 7, 4)$ Connect 4 is represented by the game 0b0 00000000000000 0000001 0000001 0000001 0000001 0000001 0000001 0000001 = 0x0000040810204081. Its children include the move 0 (bolded), corresponding to a move in the rightmost column. This move goes to position 0b1 00000000000000 0000001 0000001 0000001 0000001 0000001 0000001 0000010 = 0x8000040810204082, with hash 0x0000040810204082.

6.2 Algorithmic Improvements to Position Value Determination

The original Java solver stored a position's value as a result-remoteness tuple. Therefore, computing which of several moves was best was computationally expensive, and positions would require two bytes of storage. The new shard solver instead uses one byte to save a position's value, with the following encoding:

```
0: RESERVED
1: Loss in 0 moves
2: Loss in 1 move
3: Loss in 2 moves
...
63: Loss in 62 moves
64: Draw
65-124: UNUSED
125: RESERVED
126: RESERVED
127: RESERVED
128: Tie in 0 moves
129: Tie in 1 move
...
191: Tie in 63 moves
192: Win in 63 moves
...
254: Win in 1 move
255: Win in 0 moves
```

Games with remoteness values greater than 63 are not supported by this, but the concepts here can generalize to higher remoteness by increasing the number of bytes in a value.

Notably, this order matches the order of preference, in that for any position, the optimal move under perfect play is the one which moves to the position of minimal value. This greatly simplifies position value computations. Further, the particular values assigned allow

computation of a current position's value (given the value of its minimal child) using two comparisons and one addition, matching the runtime of the two-byte solution.

The values 0, 125, 126, and 127 are specially reserved in order to simplify the memory system for the former, and to implement the optimization in `solver.c` for the latter three.

6.3 Stack-Based Solver

Finally, the single-threaded solver was rewritten in order to be more efficient. The prior solver, in keeping with a MapReduce format, used a tier-based solver, where tiers are solved one at a time. This significantly increases the amount of memory needed, and thus reduces the cache efficiency of the solver. Thus, the overall structure of the single-threaded solver was reworked around a stack-based solver. The pseudocode below describes the general algorithm used, assuming the existence of a hash-table-like memory component:

```
Define Solve(Game):
    Insert Starting Position to Stack
    While Stack is not empty:
        Let pos = Top element of Stack
        If pos in Hash Table:
            Remove pos from stack
            Continue
        Let arr = List of children positions from pos
        For child in arr:
            If child not in Hash Table:
                If child is primitive:
                    Save primitive value to Hash Table
                Else:
                    Put child on Stack
        If all children in Hash Table:
            Let k = minimal value among all children
            Let val = value of pos, given k
            Save (pos,val) to Hash Table
            Remove pos from stack
```

The stack size gets upper-bounded by the depth of the game multiplied by the maximum number of moves per position, so this is generally most efficient when the product of these two parameters are relatively small.

Chapter 7

Data Compression of Connect 4 Game Tree

When working with large games such as Connect 4 (with terabytes of data), memory accesses and data transfers tend to dominate the overall runtime. Further, the final database size for large games end up infeasible to store naively. The Gamescrafters group has access to a 16 TiB drive; thus, it was necessary to develop a database and hashtable format that could reduce total memory use below this threshold.

7.1 Requirements

As noted above, a game is said to be solved if, for every position reachable from the starting state, we have determined if the position is a win, a loss, or a tie, and the corresponding remoteness. As it is generally infeasible to compute the entire game tree before each move in real play, it is necessary to store the results of computation in a manner that allows efficient retrieval of a position's value. Since solving is more computationally expensive than playing, it is prudent to have two distinct formats for the game tree; one for use during solving, and one for use during play. The solve-state game tree must thus support the following operations:

- Insert a given position's value, given the position's hash.
- Read a given position's value if it has previously been computed, given its hash, or indicate that the position has not yet been computed
- Save the game tree as a play-state game tree

The play-state game tree must support the following operations:

- Load the game tree from a file

- Read a given position's value, given the position's hash.

Notably, the play-state game tree does not need to insert data, and is permitted to return any value for hashes that do not correspond to positions reachable from the initial state. Since reachable positions tend to encompass only a small fraction of valid hashes (around 9% of positions for Connect 4 5x5, and 1% of positions for 7x6 [21]), solvers are allowed a significant amount of freedom when saving results. This allows for much higher compression ratios than can be attained through general-purpose lossless compression methods (like gzip).

In this section, we will analyze various options on compressing game-tree hash tables. In this analysis, we will assume that hashes (and therefore keys) are $\log_2(n)$ bits long, and that there are k valid hashes. As an example, we will provide analysis of the two naive approaches:

Approach 1: Save an array of bytes, such that the value at index i corresponds to the value with key i. This has a total memory use of n bytes, and has access time of $O(1)$. For (6,7,4) Connect 4, this database would use 512 TiB of memory, so this approach would not be usable for final play-state storage.

Approach 2: Save (key, value) tuples, sorted by key. This has a total memory usage of $O(k \log(n))$, and access time of $O(\log(k))$. Converting from Approach 1 to Approach 2 takes $O(n)$ time, while converting from Approach 2 to Approach 1 takes $O(k)$ time. For (6,7,4) Connect 4, this approach would take around 28 TiB, which while better than Approach 1, is still infeasible.

7.2 Compression of Sparse Clustered Game Trees

One useful note is that the Connect 4 hash tends to be clustered; in Connect 4, pieces that are deeply buried can no longer be included in a line of contiguous pieces, so their actual color doesn't affect the value of the position. Thus, if two positions have hashes that differ in relatively few bit positions, it is more probable than random chance that the two positions share the same hash. As such, the array of position values tend to have "chunks" of similar-valued data, which themselves generally repeat periodically.

A simple method to compress this data (Approach 3) is to assign to each non-position state the value of an adjacent position. This allows existing compression methods like zip to compress these results further than naively saving the data. In order to make this approach most efficient, boundaries (two adjacent hashes with different assigned values) are placed at multiples of the highest power of two possible. This alone provides modest improvements to the zip-compressed file size (around 20%), and can be constructed from Approach 1 in $O(n)$ time. As an improvement on this, it is possible to save instead a binary tree, with leaves corresponding to segments of length 2^n of constant data. Formally, if all nonzero values in a hashtable are equal, the tree representation of the hashtable is a single leaf node containing that unique value, or 0 if no value exists (Using the aforementioned position value representation, it is possible to store a given position's value as a single nonzero byte. This

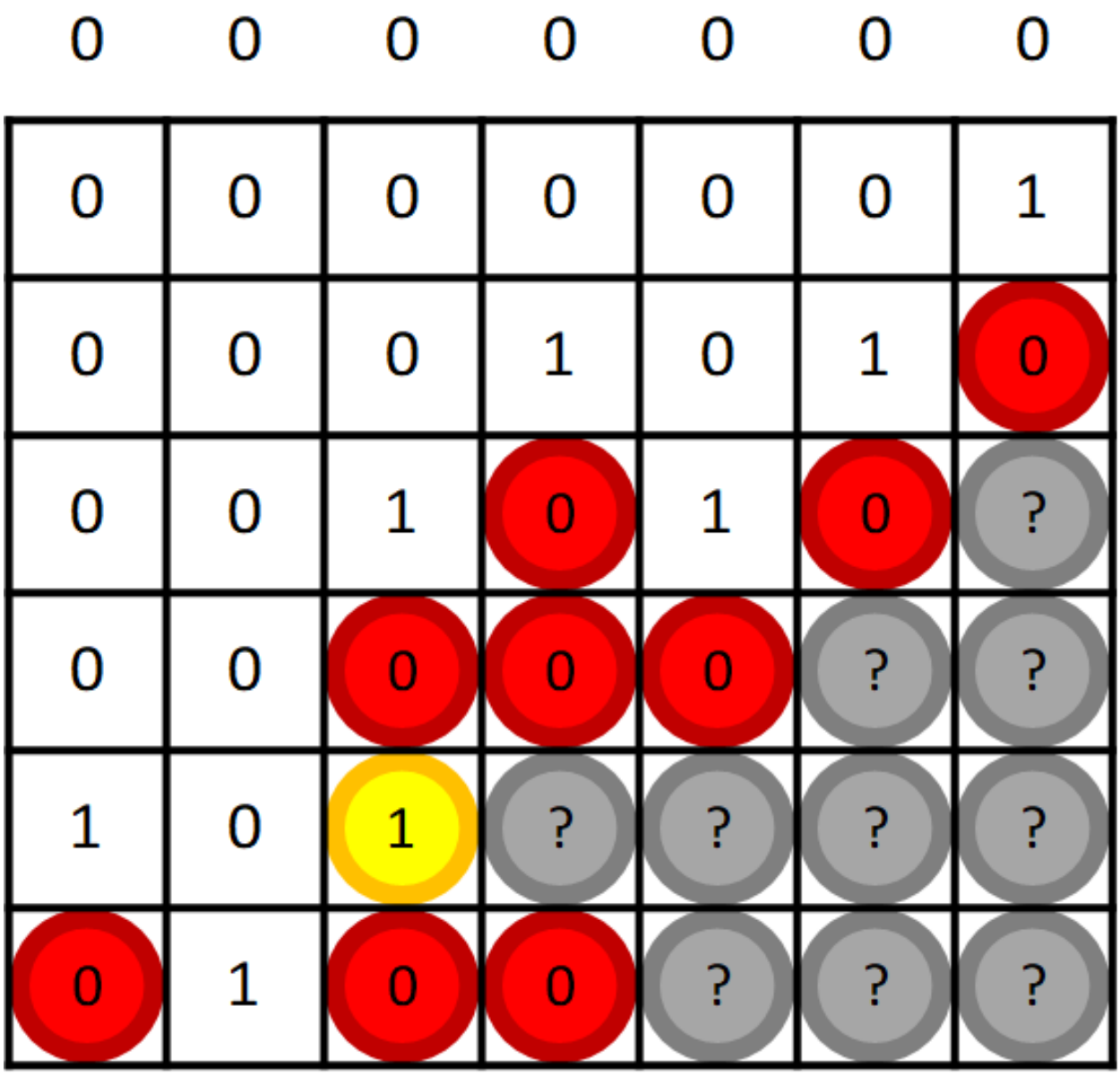

Figure 7.1: Regardless of the colors of the gray pieces, this position has the same value; a win in 1 if red is to move, and a loss in 2 if yellow is to move. This creates a "chunk" of contiguous hashes that must always be one of these two values, which can thus be compressed. In general, pieces deeply buried in a position don't contribute to a position's value.

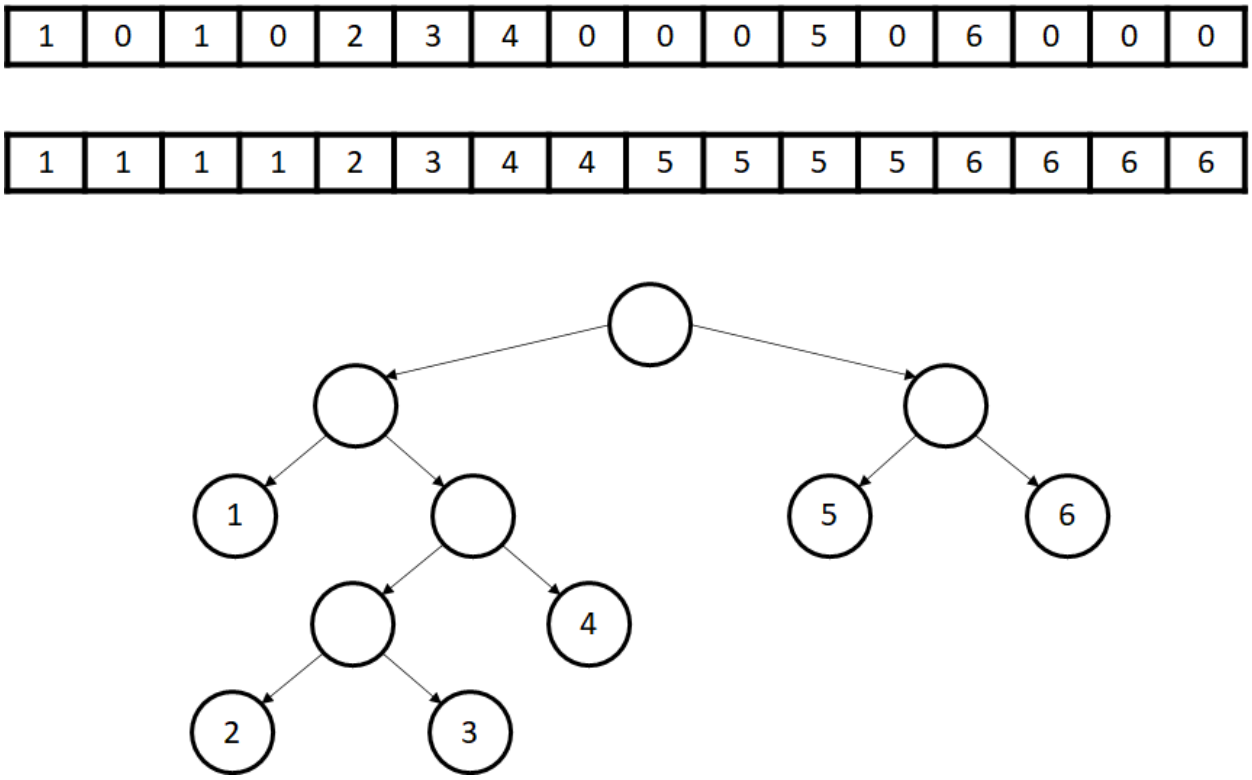

Figure 7.2: Approach 3 for memory compression, and the resulting binary tree. Approach 3 assigns values to irrelevant indexes in order to simplify compression. This can then be translated into a binary tree, which can be saved according to Approach 4 or 5.

allows for the use of 0 as an indicator that the given leaf has no defined values). Otherwise, the hashtable is split into two subtables (corresponding to hashes with a MSB of 0 and 1, respectively), and the tree representation of the hashtable is a single node with left and right children corresponding to the tree representations of the subtables.

This can be further improved by introducing a "duplicate node"; if one of the branches of a node is a zero leaf (signifying that no child exists there), we instead store only one child of that node, and specially mark that node as having only one child.

Approach 4: Store a branching node as a concatenation of (x, l, r), where x is the length of l in bytes, l is the left branch, and r is the right branch. Store a leaf as a concatenation $(0, n)$, where n is the unique value stored in that segment. Store a duplicate node as a concatenation of $(1, l)$, where l is the child tree. Note that any leaf is at least 2 bytes long, so values 0 and 1 can never correspond to valid x values.

Theorem: The tree structure defined in Approach 4 can be encoded in $O((k)(\log(\epsilon))^2)$ bytes for $\epsilon = n/k$, and has $O(\log(n))$ access time.

Proof: Let d be the number of duplicate nodes. Binary trees trivially have traversal times

of $O(\log(n))$, provided internal nodes are composed of pointers to their children. Note that at most k leaves exist, since each any given valid hash contributes to exactly one leaf, and no leaf contains zero valid hashes (otherwise it would be a 0 leaf with no parent; thus, the hash table stores zero valid hashes). Each leaf can be stored with 1 byte and 1 pointer, so this contributes $O(k)$ bytes and k pointers. The remainder of the memory use consists of internal tree nodes. Of these, there can be at most $k-1$ nodes with two branches, and d duplicate nodes. These nodes have memory equal to one pointer, so in total, they contribute $d+k$ pointers. By induction, a node that corresponds to a table of size h needs $O(\log(h))$ space to store its pointer. Let the nodes corresponding to tables of size h be defined as "layer h". Then the number of pointers in layer i is at most $2^{\log(n)-i}$. In the worst case, the $2k+d$ pointers are at their highest layer possible. This corresponds to the bottom filled layer being layer $-\log(\epsilon)$, and the top filled layer being layer $\log(k)-\log(\epsilon)$. Since each layer has twice as many pointers as the next, this averages to $2-\log(\epsilon)$ bytes per pointer. Thus, the total memory use of these pointers is $O((k+d)(1-\log(\epsilon)))$. Thus, the total memory use of the entire tree structure is $O((k+d)(1-\log(\epsilon)))$. Finally, we note that at most $-\log(\epsilon)$ duplicate nodes can exist per leaf node, since each duplicate node necessarily requires half its children to be empty. Thus, $k+d=O(k(1-\log(\epsilon))$, so our total tree uses $O(k\log(\epsilon)^2)$ bytes.

Finally, we note that Approach 4 can be constructed from Approach 1 in a single linear pass, and thus can be computed in $O(n)$ time. Translating from Approach 4 to Approach 1 also takes $O(n)$ time.

Approach 5: The pointer value is only necessary when doing a $\log(n)$ access. If we forgo this, it is possible to simply store a 2 for any pointer instead; the original data can still be recovered by expanding the full tree. This reduces all pointers to a fixed size, which in turn reduces the memory use to $O(k\log(\epsilon))$, but also increases access time to $O(n)$. Converting from Approach 5 to Approach 1 still takes $O(n)$ time, as does converting from Approach 1 to Approach 5.

Note that Approaches 4 and 5 are fully compatible with existing lossless compression methods; zipping these files generally results in further memory use reduction.

As we will see later, the sharding process also ends up dividing the memory solver into chunks of about one billion positions each. This doesn't significantly affect the final memory use of the overall tree, but reduces access times and conversion times, thus rendering the deficiencies of Approach 5 minimal.

7.3 Specific Implementations of Memory Modules

For solver-side computation, fast access times generally take priority over memory use (assuming that hashes are reasonable length), so all current memory modules use Approach 1 internally. `memorynaive.c` is written to use Approach 1 when saving the resulting data as well, so it can be considered a useful benchmark.

`memoryoptimized.c` uses Approach 3 when saving, thus providing slightly smaller zipped files.

`memorymoreoptimized.c` and `memorywithplayer.c` both use Approach 4; the former does not have a player implemented, and serves more as a template from with tree-based approaches can be developed.

`memoryfastretrieval.c` is the current best memory module. It relies on Approach 5, and converts to Approach 1 on loading player files. This allows for both efficient storage and efficient retrieval during shard solving.

The remaining memory module attempts to save data under separate hash tables based on their tier. In theory, this can increase the number of contiguous blocks, thus reducing total memory usage, for example if a set of positions is likely a win for whichever player moves first. However, Connect 4 was sufficiently clustered that this worsened memory use. While this approach may be useful for other games, it is as of now less efficient than the current memory module.

Chapter 8

Shard-Based Solving

The core optimization of the new solver was the **sharding** process. In this section, we will discuss how this works, as well as the properties of Connect 4 that allow this to work efficiently.

8.1 Motivation

Tiered game solving is indeed embarassingly parallel[15], and thus would appear to be well-suited to a MapReduce framework. However, one major aspect distinguishes this problem from traditional MapReduce problems; due to the random accesses needed for positions to get the values of their children, a MapReduce framework requires a significant amount of memory use and data transfer. Using (6,7,4) Connect 4 as an example, the entire 512 TiB hashtable naively needs to be communicated twice for each tier, thus leading to a total memory transfer on the order of a few petabytes. Even assuming a theoretically optimal MapReduce system (where positions are transferred only when queried), each position has on average 4 children, and requires five bytes of data transfer per child (a four-byte position in discovery, and a one-byte result in solving), thus requiring around 80 TiB worth of data transfer, with around 8 trillion individual data transfers. In addition, the large size of the hash table (512 TiB using an approach 1 memory module, and several TiB for the largest tiers in an approach 2 memory module) is impractical to store within random access memory, so the MapReduce system either needs to store the table as key-value pairs (which significantly slows accesses compared to the array approach), or needs to save it within slower memory segments such as the disk.

One way around this is to split a large game into smaller subgames known as **shards**, which are small enough to compute effectively on a single node. This allows the majority of the hashtable to be ignored, while only a few GiB of the table at a time are worked on.

The sharding approach has been used previously in solving games such as Nine Men's Morris, where the game itself defines a natural sharding [14]. In Nine Men's Morris, players begin with nine pieces each, and attempt to capture opposing pieces until one side has only

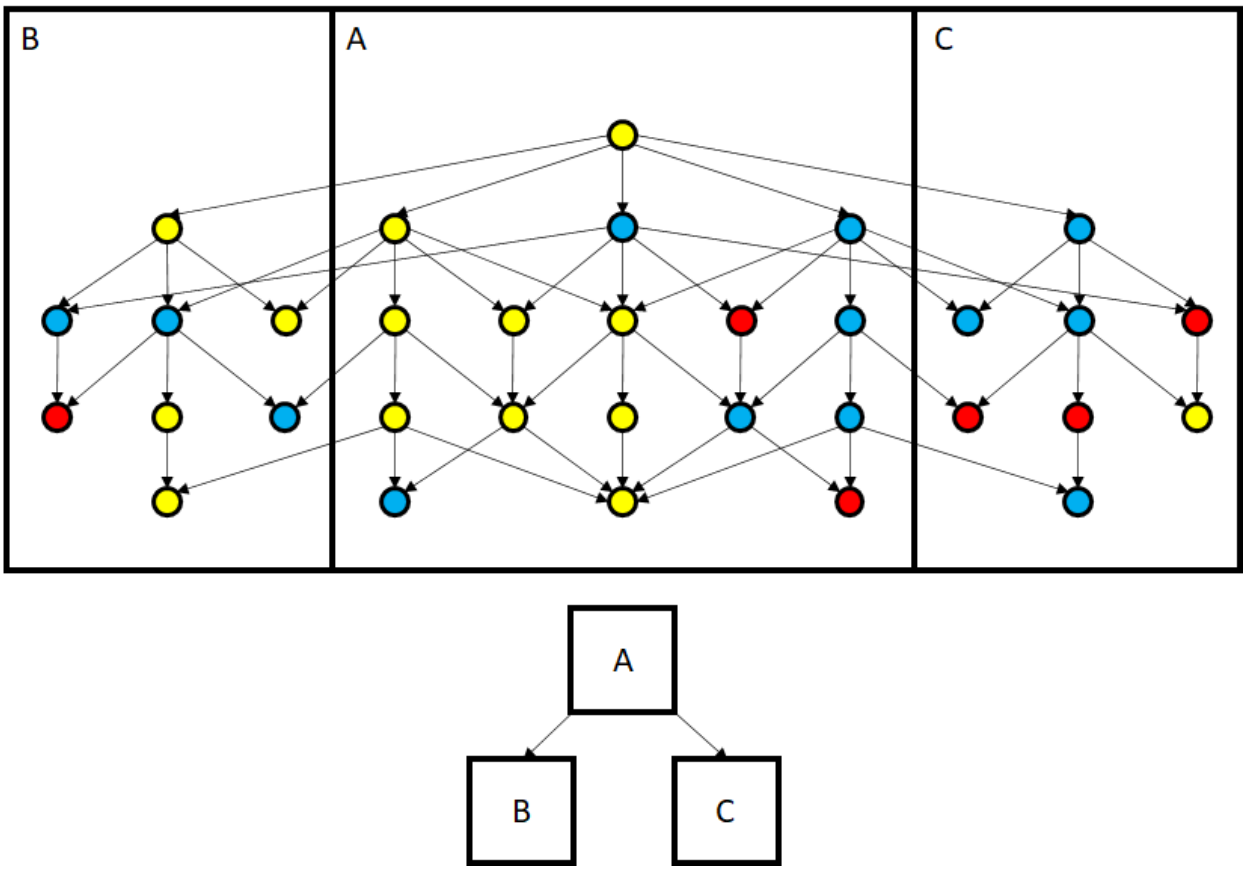

Figure 8.1: Example sharded game. Only a few moves go between moves in different shards; further, moves only go from shard A to B or C, and never the reverse. This thus creates a simplified shard graph, which is also acyclic. By assigning each shard to a separate process, it is possible to distribute a solve's workload while minimizing necessary communication.

two pieces left, or has no available moves. The overall game is cyclic (and indeed is a draw under perfect play); however, by splitting the game into 28 shards based on the number of pieces currently in play, it is possible to treat the higher-level shard dependency graph as an acyclic graph, and compute each shard separately. This was used in the Nine Men's Morris solve to reduce the game size that needed to be computed at one time.

In Connect 4, there isn't as obvious a "natural" sharding. It is, however, possible to divide the entire game tree into shards based on the state of the leftmost column. This splits the game into 128 different shards, each of which can largely be solved independently. If we split the game across 128 different nodes with one shard per node, each node can compute its own segment (using only 4 TiB of memory) and communicate only the moves that go between shards. Since only moves in the left column change shards, a majority of moves stay within the shard, so relatively little needs to be communicated between shards.

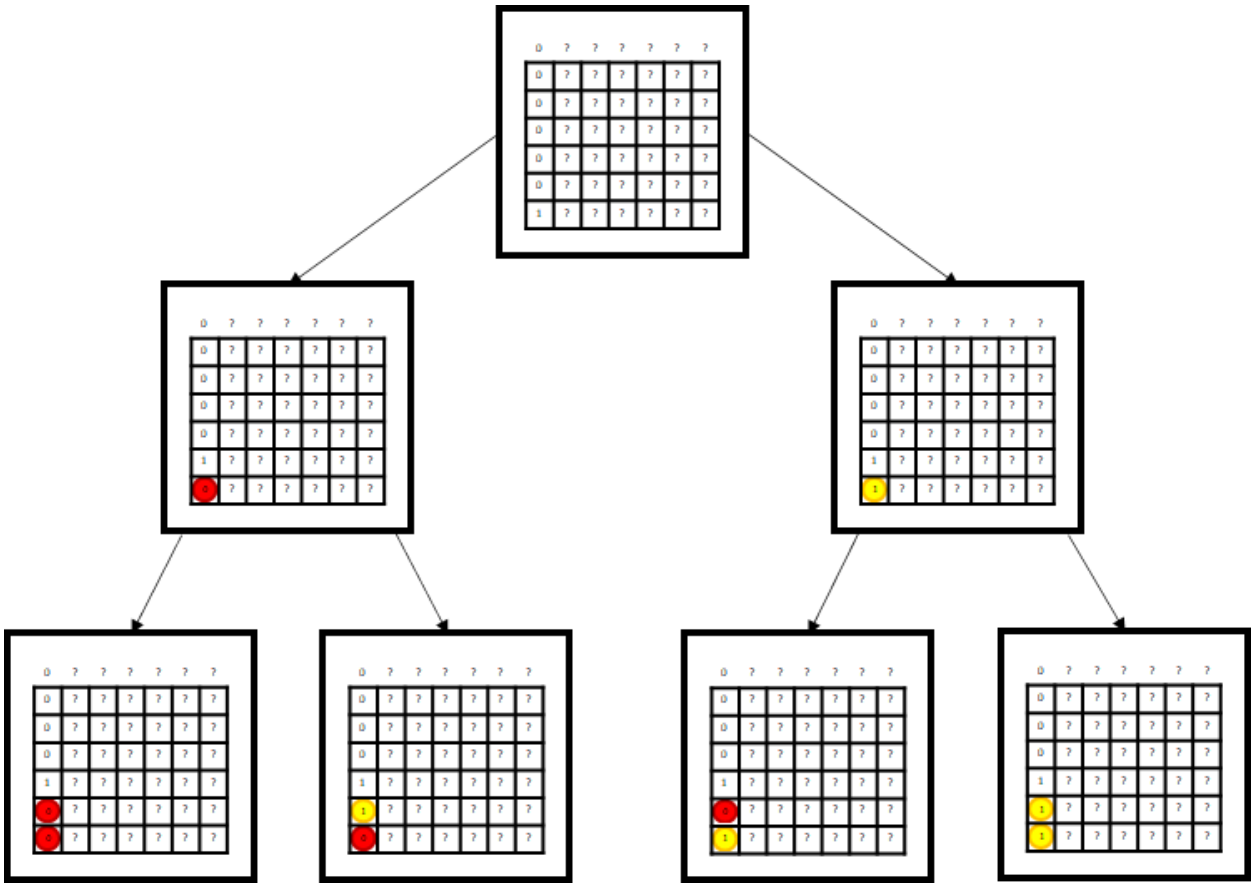

Figure 8.2: The first 7 shards of a Connect 4 sharding. Each shard contains all positions with the given left column. Only moves in the left column move between shards, and each shard has only at most two children shards.

Notably, the choice of sharding by the left column is arbitrary, which allows for multiple shardings of different sizes to be chosen. Instead of using just the leftmost column as our shard identifier, we can instead use the leftmost three columns, thus having each of 2 million shards use a reasonable 256 MiB of memory. If run on 1000 nodes, 1000 shards can be run at a time, with communications saved between running shards. A finer sharding is able to use more available nodes than a coarser sharding, but also incurs a higher communication cost. Thus, the existence of multiple shardings allows for fine-tuning the shard size to the compute cluster in use.

8.2 Formalization

Definition 13. A **sharding** of a game is defined as a partition of all states into sets. These sets are thus referred to as **shards**. The **minimal shard graph** of a sharding is defined as the unique directed graph associated with a given sharding of a game, where nodes correspond to shards containing positions, and a directed edge exists between shard i and shard j if and only if there exists positions $i' \in i, j' \in j$ such that there exists a directed edge from i' to j' in the game's graph of positions. A **shard graph** of a sharding is any finite graph containing a component isomorphic to the minimal shard graph as a subset; thus, shard graphs are allowed to have extraneous edges and inaccessible shards. A sharding is said to be **acyclic** (resp. **monotonic** with respect to a given enumeration) if an easily computed shard graph (resp. with enumeration) of that sharding is acyclic (resp. monotonic).

Definition 14. A sharding's **move ratio** is the fraction of edges in the game's graph of positions that are between positions in different shards. A sharding's **size ratio** is the fraction of positions that are in the shard with the most positions.

Example. Two shardings are always possible for a game. The first sharding is one where all positions are placed in a single shard. In this case, the sharding's move ratio is 0, and the sharding's size ratio is 1. The second sharding is one where each position is assigned its own shard. This sharding has a move ratio of 1, and a size ratio near 0. These shardings are referred to as the **trivial sharding** and **complete sharding** of a game, respectively.

Under this definition, a tiering of a game is a special case of a sharding, with a move ratio of 1 and a relatively high size ratio. Further, the minimal shard graph is acyclic, and indeed linear. These properties lend themselves well to the MapReduce framework, as this makes the positions within each shard independent, and the high size ratio provides a high upper bound on the maximum possible parallelism. For the shard solver, however, we intend to solve individual shards in single nodes, rather than distribute them. As such, the shard solver prefers a sharding with a low move ratio (to minimize inter-shard communication) and a low size ratio (to minimize the maximum load on any single node). As with the tiered sharding, the sharding must admit an acyclic shard graph in order to be efficiently computed, and additionally prefers sparse shard graphs.

One useful class of shardings is the shardings associated with a given hash function of the game:

Definition 15. The hash sharding of size n of a game is defined as the sharding defined by partitioning the set of positions according to their hash. If the hash size is $k \geq n$, then a total of 2^{k-n} shards are created, where shard i contain all positions with hash in the range $[i * 2^n, (i + 1) * 2^n)$.

Example. The trivial sharding and complete sharding correspond to the hash sharding of sizes k and 0, respectively. If the game is finite-length, these shardings are guaranteed to

acyclic. The remaining hash shardings are not guaranteed to be acyclic, even if the game is finite-length.

Definition 16. A hash function is said to be shard-acyclic if all associated hash shardings are acyclic.

Hashes that are shard-acyclic are useful, as they allow fine-tuning shard size to the system. However, the existence of any acyclic hash sharding of reasonable size (around 25-28 works best) allows the shard solver to work.

Theorem 4. *Monotonic hashes are shard-acyclic.*

Proof. Assume without loss of generality that the hash is monotonic increasing. A cycle in the shard graph necessarily requires at least one directed edge from a shard of higher index to a shard of lower index, which therefore requires the existence of some move from a higher hash to a lower hash. This contradicts that the hash is monotonic increasing. Thus, monotonic hashes are shard-acyclic. □

As an aside, we note that all games of finite length necessarily admit a monotonic hash, by enumerating positions according to a topological sort of its positions. Thus, shardings of any size ratio can be found, though this generally results in a hash which is difficult to compute, and does not provide guarantees on move ratio.

We also note that the two extreme shardings correspond roughly to previously discussed solving approaches; the MapReduce model effectively uses the trivial sharding under a breadth-first ordering, while the single-threaded solver uses the complete sharding. We can thus interpret intermediate shard sizes as being a middle-ground between the two approaches, therefore taking advantage of both the low communication cost of the single-threaded solver and the work distributions of the tier-based solver.

8.3 Application to Connect 4

As hinted in the motivation section, the Connect 4 hash described earlier fulfills many properties that allow the shard solver to be efficient. To wit:

- The hash is easy to compute
- The hash is monotonic, and therefore shard-acyclic
- All associated hash shardings have relatively low size ratios, as a shard with size n has at most 2^n positions. For reasonable sizes, a single shard can be solved by one thread in about 1-2 seconds using the stack solver.
- All associated hash shardings have relatively low move ratios. A quick estimate would put the move ratio as approximately the number of columns which are used to distinguish shards, which is thus approximately linear with $k - n$.

- The resulting shard graph is sparse. Each shard has at most two children shards per column used to distinguish shards. Thus, for (6,7,4) Connect 4 with a shard size of 28, each shard has at most six children shards, and is the child of at most three parent shards.

The `Game.h` file requires a function that determines the child shards of the shard with a given shard ID and size. This is ideally fast to compute, as the full shard graph is precomputed on all nodes before starting to solve any shard.

8.4 Shard-Based Distributed Computing Solver

The shard-based solver is divided into two components: A large pool of worker nodes, which are assigned individual shards to compute, and a single "manager" node, which assigns work, controls which shards get computed, and outputs progress information to the user. The worker nodes primarily run in `solver.c`, while the manager node primarily runs in `maindriver.c`.

The Manager Node

As with the MapReduce solver, solving each shard is divided into two stages; a **discovery** phase which determines the list of positions in the shard, and a **solve** phase which determines the value of each position. The primary difference from the tier solver is that discovery and solving must now handle multiple input shards and output shards. In the shard graph, each directed edge from shard i to shard j lists a set of valid positions in shard j that must be solved before shard i can be solved. The set of positions in shard j is thus contained in the descendants of all positions listed in at least one incoming edge in the shard graph; any position not found this way is necessarily unreachable from the start state. Thus:

- Discovery of shard i outputs a list of all children of positions in shard i that belong in other shards, given lists of positions from its parent shards.
- Solving of shard i creates a play-state game tree of the shard, given the positions from its parent shards and the play-state game trees of its children shards.

Discovering a shard requires that all its parent shards have been discovered, and solving a shard requires that all its children shards have been solved and that the current shard has already been discovered (in order to guarantee that all parent shards have been discovered, and for implementations that compute a list of all positions in the shard during discovery). The manager node is thus responsible for ensuring that discovery and solving are called only when possible.

In order to accomplish this, the `shardgraph` struct is defined in `solver.h`:

```
typedef struct shardgraph {
    uint64_t shardid;
    int childrencount;
    int parentcount;
    int parentsdiscovered;
    int childrensolved;
    struct shardgraph** childrenshards;
    struct shardgraph** parentshards;

    //For use in shard work queue. Owned by maindriver.c
    struct shardgraph* nextinqueue;
    int discovered; //0 == untouched, 1 == discovered, 2 == solved
} shardgraph;
```

This acts mostly as a standard graph implementation with adjacency lists, but additionally includes a list of parent shards, counters for the number of parents discovered and children solved, and a state variable that tracks if the shard is untouched, discovered, or solved. In addition, the `shardgraph` implements a queue structure, which is used by the main driver to keep track of all shards that can currently be worked on.

The following pseudocode details how the single-threaded main driver works, given functions to discover and solve shards:

```
Define Solve():
    Let Shardgraph = computeshardgraph()
    Initialize Queue with starting shard
    While Queue is not empty:
        Pop shard from queue
        If shard is untouched:
            Discover(shard)
            Mark shard as discovered
            For each child shard of shard:
                Increment parents discovered of child shard
                If parents discovered == parent count, add child shard to Queue
            If current shard no children shards:
                Add shard to Queue
        Else If shard is discovered:
            Solve(shard)
            Mark shard as solved
            For each parent shard of shard:
                Increment children solved of parent shard
                If children solved == children count, add child shard to Queue
```

This guarantees that discovery and solving happen in an acceptable order.

For the OpenMP solver, a similar process is run on each thread. A lock is used to ensure that at most one thread accesses the shard graph at a time to determine if a new shard can be worked on; if no work is currently available, the thread simply sleeps for 1 second in order to wait for more work to be available, or terminates if the starting shard has been solved.

The MPI solver is configured such that only the manager runs the shard queue. Once a worker finishes setting up, it sends a message to the manager asking for more work (by sending the number -1). In this case, the manager checks if work is available, and if so, responds to that message with the shard to be worked on, and if the shard should be discovered, or solved. If no work is available, the manager responds with a sleep request, in which case the worker waits 1 second before checking again for work, or sends a termination request if the solve has been completed, in which case the worker exits and terminates. Once a worker completes its shard, it sends back the shard ID to the manager. The manager then computes which shards can be worked on next, and assigns more work to the worker if work exists.

Worker Nodes

Worker nodes start by initializing their constants, as well as the full shard graph; this is used to track the children and parent shards of a working shard. After this, they run discovery or solves of individual shards, per requests from the manager node. Both discovery and solve phases are fairly similar to the stack-based solver, with the added change that each shard receives a list of starting positions, instead of a single starting position. When working on a shard, separate hashtables are used for the shard and each child shard, and a number of files are opened for each parent shard.

All communication between shards is done via the file system; in the working folder, a new file is created for each edge in the shard graph (named "`transfer-<parent shard ID>-<child shard ID>-<position type>`") and for solved shards (named "`solved-<shard ID>`"). The transfer itself has separate segments for nonprimitive positions and primitive positions, since primitives rely on knowing the most recent move under `Game.h`'s defined interface. The following pseudocode describes the discovery and solve code:

```
Define Discovery(shard):
    Initialize solverdata for current shard, and one solverdata per child shard
    For each parent shard:
        Open corresponding nonprimitive transfer file
        For each position in transfer file:
            Insert position to Stack
            While Stack is not empty:
                Pop pos from Stack
                If pos in current shard solverdata:
                    Continue
                Add pos to current shard solverdata
```

```
                Let arr = List of children positions from pos
                For child in arr:
                    If child in current shard:
                        Add child to Stack
                    Else:
                        Save primitive value to child shard's
                            solverdata
    For each child of shard:
        Save nonprimitive, loss, and tie positions in
            child's solverdata to separate files

Define Solve(shard):
    Initialize solverdata for current shard, and one
        playerdata per child shard (from file)
    For each parent shard:
        Open corresponding primitive transfer files
        Save all primitives to solverdata
        Open corresponding nonprimitive transfer file
        For each position in transfer file:
            Insert position to Stack
            While Stack is not empty:
                Let pos = Top element of Stack
                    If pos in solverdata:
                        Remove pos from stack
                        Continue
                Let arr = List of children positions from pos
                For child in arr:
                    If child in current shard and not in solverdata:
                        If child is primitive:
                            Save primitive value to solverdata
                        Else:
                            Put child on Stack
                If all children solved:
                    Let k = minimal value among all children
                    Let val = value of pos, given k
                    Save (pos,val) to solverdata
                    Remove pos from stack
    Save solverdata as playerdata to file
```

The starting shard is, as with the stack-based solver, initialized with the start position of the game.

The current `solver.c` additionally adds one optimization that minimizes the size of the

transfer files: when saving nonprimitive positions to a file during discovery, it marks any children of that position as not needing to be saved, since they will be expanded during discovery of the child shard anyway. These children are further expanded, thus removing any descendants of a given position from needing to be saved. Overall, this significantly reduces the number of positions that need to be transferred (by about a factor of six). The current form of this optimization takes advantage of the linear pass through the solver needed to save all positions, and thus only works because Connect 4's hash is monotonic. It is possible to implement this optimization for games with nonmonotonic hashes, but this would require a separate loop to fully remove all descendants. Regardless, this does cause a slight runtime increase, so large games with extremely low move ratios may benefit from using `solvermpiinitial.c` instead.

Chapter 9

Results

Three main metrics are useful when determining the efficiency of the shard solver:

- The size of player results, both before and after gzip compression
- The strong-scaling and weak-scaling efficiency of the solver.
- The total amount of communication necessary for the solver to run

This culminated in an efficient solve of (6,7,4) Connect 4, and the compression and saving of a final player database within the Gamescrafters' 16 TiB drive.

Over all tests, 8841 service units were consumed, equal to about 3% of Gamescrafters' yearly allotment.

9.1 Player Database Compression

Tests for this were done locally with (5,5,4) Connect 4, as runtime performance was not of significant concern, and `gzip` tends to run slowly for large files. After computing results, `gzip` was used with default parameters for lossless compression. (5,5,4) Connect 4 uses a 30-bit hash to store 69,763,700 positions [21], so an uncompressed database would use 1 GiB using Approach 1, and 266 MiB using Approach 2.

Table 9.1 shows the database size, bytes per valid position, and compression ratio (relative to Approach 1) obtained from running these tests on the four memory modules. The Optimized memory module (which used Approach 3) yielded a 24% better compression ratio when used with `gzip`, a modest improvement over the naive approach. Approaches 4 and 5 (run with the More Optimized memory module and Fast Retrieval Memory module, respectively) yielded around a 10x better compression ratio before `gzip` compression, and around a 2x better compression ratio after `gzip`. Thus, the tree structures used in Approaches 4 and 5 significantly improved compression ratios of player databases.

Testing was also done on how effective the Fast Retrieval solver was at compressing various game sizes. Due to the nature of the tree structure used in Approach 5, database

	Naive	Optimized	More Optimized	Fast Retrieval
Database Size (Raw)	1 GiB	1 GiB	113 MiB	106 MiB
Bytes/Position (Raw)	15.4	15.4	1.70	1.59
Compression Ratio (Raw)	1	1	9.06	9.66
Database Size (gzip)	41.2 MiB	33.3 MiB	24.2 MiB	18.7 MiB
Bytes/Position (gzip)	0.62	0.50	0.36	0.28
Compression Ratio (gzip)	1	1.24	1.70	2.20

Table 9.1: Memory use of (5,5,4) Connect 4 database, before and after running gzip with default parameters

	(5,5,4)	(6,5,4)	(5,6,4)	(6,6,4)	(6,7,4)
Positions	69,763,700	1,044,334,437	2,818,972,642	69,173,028,785	4,531,985,219,092
Database Size (Raw)	106 MiB	1.4 GiB	3.6 GiB	78 GiB	4.3 TiB
Bytes/Position (Raw)	1.59	1.44	1.37	1.21	1.09
Database Size (gzip)	18.7 MiB	236 MiB	630 MiB	12 GiB	557 GiB
Bytes/Position (gzip)	0.28	0.24	0.23	0.19	0.13

Table 9.2: Memory use of Connect 4 database over multiple game sizes, before and after running gzip with default parameters

size is nearly independent of shard size; thus, only one test was done per unique game size. This is shown in Table 9.2. Compression rates were maintained, and indeed slightly improved, for larger game sizes.

9.2 Scaling Efficiency of the Shard Solver

Scaling tests were done on the Savio cluster, using the `savio3` partition. The partition specs [1] are restated here for convenience:

- 112 32-core nodes with a 2.1 GHz Intel Xeon Skylake 6130 CPU model and 96 GB RAM
- 72 40-core nodes with a 2.1 GHz Intel Xeon Skylake 6230 CPU model and 96 GB RAM

All tests (except those with only one core) were run using the `maindrivermpi.c` and `memoryfastretrieval.c` modules.

Strong Scaling

Strong scaling tests were done on (6,6,4) Connect 4, with shard sizes ranging from 28 to 25. Since the MPI solver uses a dedicated manager core, few-core jobs incur a significant

	28	27	26	25
Shards Evaluated	16129	32258	64516	129032
Runtime (10 cores, seconds)	14443.52	16018.8	17345.5	18769.35
Runtime (100 cores, seconds)	2046.57	1826.81	2016.83	2267.5
Runtime (960 cores, seconds)	393.94	318.27	270.01	264.47
Efficiency (10 cores)	479000	432000	399000	369000
Efficiency (100 cores)	338000	379000	343000	226000
Efficiency (960 cores)	183000	226000	267000	272000

Table 9.3: Strong Scaling Efficiency of Shard Solver, run on (6,6,4) Connect 4, by shard size

runtime penalty relative to many-core jobs. Thus, tests were done using:

- 10 cores, as this provides a useful "low-core" system while mitigating the effect of the manager core
- 100 cores, as a medium-sized system
- 960 cores, in order to fit within the 1000-core limit and to ensure full utilization of all nodes (960 is a multiple of both 32 and 40).

Table 9.3 and Figure 9.1 show the runtime and efficiency (measured in positions solved per second per core) of the shard solver. In general, a larger shard size yielded lower overhead costs (due to fewer shard loads and saves), while a smaller shard size yielded better strong scaling (due to the greater number of shards). A shard size of 28 yielded an efficiency of around $800000 * x^{-0.2}$ (where x is the number of cores), while a shard size of 25 yielded an efficiency of $430000 * x^{-0.07}$.

Weak Scaling

While Connect 4 does provide some incremental game sizes, the overall set of viable problem sizes is relatively limited; only the (5,5,4), (5,6,4), (6,5,4), and (6,6,4) games are sufficiently large to provide reasonable data, and sufficiently small to test repeatedly. Since the (5,5,4) game is almost exactly 1000 times smaller than the (6,6,4) game, weak scaling tests were unable to encompass both games; thus, the (5,5,4) game was also omitted. For all tests, a constant shard size of 25 was chosen, to avoid inefficiencies due to idle cores.

Table 9.4 and Figure 9.2 show the weak scaling efficiency of the shard solver. Weak scaling was approximately proportional to $x^{-0.21}$, which, while lower than strong scaling efficiency, is somewhat expected. Some runtime components of the shard solver are linear in the total hash range rather that the total number of positions; since smaller games have a higher density in their hash range, larger games would incur an efficiency penalty.

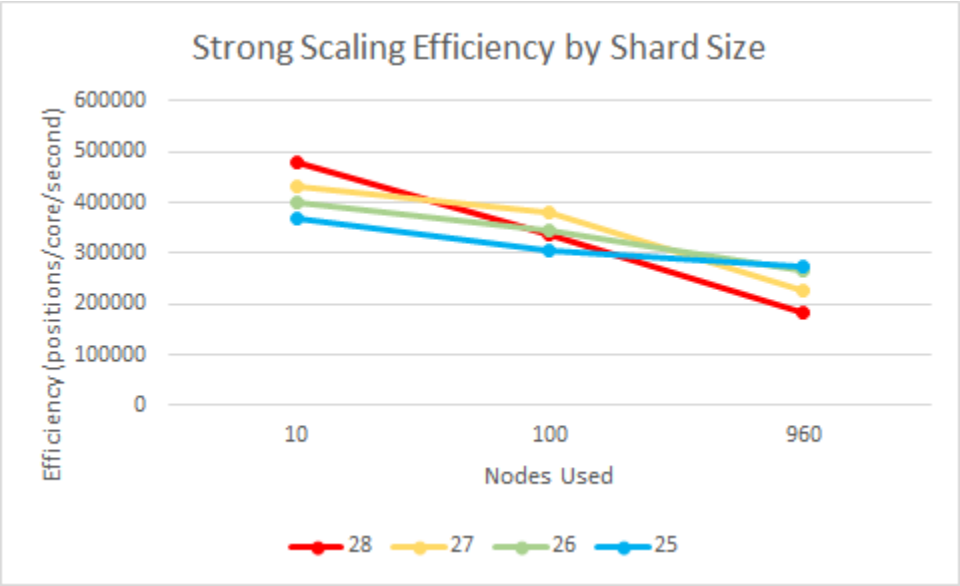

Figure 9.1: Strong Scaling Efficiency of Shard Solver, for shard sizes of 28, 27, 26, and 25. Shard size of 28 is approximately $800000 * x^{-0.2}$, while a shard size of 25 yields $430000 * x^{-0.07}$

	(6,5,4) Connect 4	**(5,6,4) Connect 4**	**(6,6,4) Connect 4**
Problem Size (Billions of Positions)	1.04	2.81	69.2
Cores Used	16	40	960
Runtime (seconds)	107.25	131.95	264.47
Efficiency	672000	546000	272000

Table 9.4: Weak Scaling Efficiency of Shard Solver

9.3 Communication Overhead of the Shard Solver

Each scaling test produced a folder containing all solved shards and all transfer files; this allowed a check on the total disk space used for communication. Under a MapReduce system, each position is saved to disk exactly once, yielding a total transfer size of 8 bytes per position.

Tables 9.5 and 9.6 show these results. As expected, games with a high number of shards require more positions to be transferred. The ratio between the number of bits used to identify shards and the total number of bits in a hash serves as a good approximation for the fraction of moves that cross from one shard to another; the data here indicates that this is approximately correlated with the transfer ratio, with larger, sparser games requiring less data transfer.

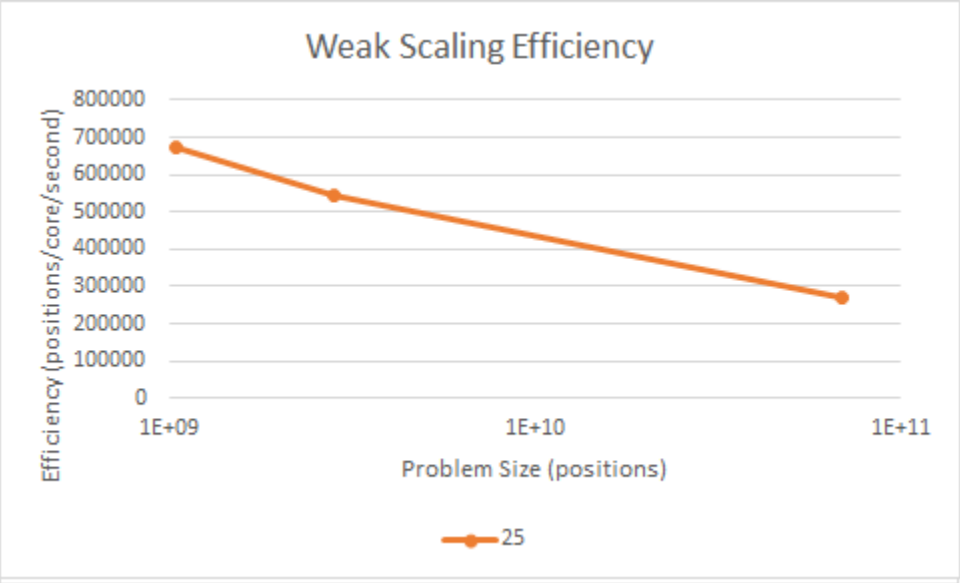

Figure 9.2: Weak Scaling Efficiency of Shard Solver. Approximately $52500000 * x^{-0.21}$

	(6,5,4)	**(5,6,4)**	**(6,6,4)**	**(6,7,4)**
Shards	1016	2016	129032	2048383
Fraction of hash bits used to identify shards	0.29	0.31	0.40	0.42
Transfer Amount	654 MiB	1.3 GiB	65 GiB	3.4 TiB
Bytes Transferred/Position	0.26	0.50	1.01	0.82

Table 9.5: Amount of data transferred by game size (shard size of 25). For (6,7,4), a shard size of 28 was used instead.

	28	**27**	**26**	**25**
Shards	16129	32258	64516	129032
Fraction of hash bits used to identify shards	0.33	0.36	0.38	0.40
Transfer Amount	32 GiB	44 GiB	53 GiB	65 GiB
Bytes Transferred/Position	0.50	0.68	0.82	1.01

Table 9.6: Amount of data transferred by shard size ((6,6,4) Connect 4)

9.4 Solving (6,7,4) Connect 4

The final (6,7,4) Connect 4 solve was done using the Fast Retrieval memory module and 960 cores, with a shard size of 28 (0.42 of the hash bits are used to identify shards). As noted earlier, this game has 4.5 trillion positions, so a MapReduce solver would use 36 TB of disk space, and naively store 41.5 TB for the final player database.

The solve completed in 20484.77 seconds (230455 positions solved per core per second), or 5 hours and 41 minutes. In total, 3.4 TiB were used in disk space for transfers (0.82 bytes per position). The final player database was 4.5 TiB uncompressed (1.09 bytes per position), and 557 GiB after gzip compression (0.13 bytes per position). These results are consistent with tests at smaller sizes; thus, we can expect this solver to exhibit similar performance for larger games.

Chapter 10

Future Work

The solver at present state does indeed perform better than previous solvers. However, there are still a number of small optimizations that can be done to improve the efficiency of the Connect 4 solver. This section will discuss likely avenues of improvement in the solver and Connect 4 code, as well as other games which are likely to be efficiently solved by this solver.

10.1 Improvements to Current Algorithms

The following sections detail potential improvements to the shard solver, which could feasibly allow for further speedups. Within each subsection, they will be roughly ordered by the expected difficulty of the modification, with early improvements likely useful for onboarding new members, and later improvements likely encompassing a full-semester project.

Benchmarking

Before any improvements are made on the individual components of the solver, it would be prudent to run a few benchmark tests to determine which components are currently taking the most runtime. Benchmark testing would also serve as a useful way to onboard new members, to gain familiarity with the codebase before significantly modifying it. Once benchmark tests have been run and the slowest component determined, it would be possible to accurately assess which of the following optimizations to prioritize, and the expected maximum improvement.

Connect 4

The current Connect 4 code currently attempts to balance runtime for game operations (ex. finding a position's children), and unhashing values. However, depending on the exact number of these operations that get done in a solve, it may be preferable to have faster game operations (at a cost of slower unhashing) or faster unhashing (at a cost of slower game operations).

Of these, it is simpler to implement faster unhashing, simply by removing the most significant bit of the game struct. This makes the `game` struct exactly equal to its `gamehash`, trivializing unhashing. In this case, it is useful to instead encode the current player in the move instead (such as by using the MSB of the move to encode the current turn), thus reducing the needed player computation to a single function. This could also theoretically synergize well with the GPU optimization described later, as the similarities to tiered solving allow the program to assume large blocks of same-player moves.

To implement faster game operations, the general approach would be to add additional components to the `game` struct, in addition to the hash. Currently, the `game` struct consists of the hash, plus a single bit used to track the current player. Tromp's Connect 4 code additionally includes a bitmask that tracks the current set of cells which are nonempty; in this bitmask, a bit is set to 1 if that bit position corresponds to a red or white cell, and 0 if that bit position is either empty, or not associated with a cell. Using this, Tromp lists a different method of computing primitives, which is capable of determining if a position is a win in 8 bitwise operations. Keeping track of this in the `game` struct would approximately double memory use for games (though hashes will stay the same size, and currently only a few `game` structs remain unhashed at a time), and double runtime for unhashing. However, this would significantly reduce the runtime needed to compute if a position is a primitive position. Further, we can also store the bit positions of the lowest empty cell of each column, to speed up computing moves.

Beyond this, it would be feasible to convert the current code to in-line assembly. Most of the Connect 4 code is already written with bitwise operations, and relatively few branches. However, manually writing assembly would allow slight reductions in the number of branch operations, thus saving time (at the expense of portability).

Memory Modules

The currently available memory modules use large amounts of RAM to allow for fast random accesses. This limits the maximum shard size to around 28 bits. Further, the current solver relies on a linear pass through the memory component, reducing the efficiency on games with low hash density. As such, larger shards would require a more memory-efficient system, and games with low hash density would prefer a memory module implementing the successor and predecessor operations.

The former can be resolved by implementing a standard hash table. This could be achieved relatively easily, though this would incur a significant cost in access time. For the latter, a **van Emde Boas tree** or **Y-fast trie** [24] would theoretically allow solver runtime to be reduced from $O(n)$ to $O(k \log(n))$ (where n is the size of each shard and k is the number of positions in the shard). However, implementing this would require significant further development (as these data structures are relatively obscure), constant factors may limit their effectiveness, and these structures would lose the thread-safe behavior that would permit some solver improvements.

For saving positions to files, it is useful to note that the current memory saver saves pointers only in multiples of bytes. In the current best memory system, the majority of the data consists of 2-bit pointers stored as full bytes. Modifying the memory module to save by bits rather than by bytes could save significant space, and bring the pre-zipped size closer to the zipped size.

Solvers

As noted, the current solver uses an optimization designed to reduce transfer file sizes by removing any positions that are descendants of already-transferred positions. This is done through a linear sweep of the memory module, which thus relies on the hash being monotonic. Thus, any game whose hash is nonmonotonic-increasing would be less efficient with the current solver. This can be fixed by doing a sweep of the memory module in a loop before starting to save data. Since this loop would likely take longer to run than the current code, it would likely be best to fork the current code and make a special version for monotonic hashes. Doing this would also allow the monotonic solver to use this hash property during discovery, thus allowing for a faster solve on monotonic hashes.

For larger games, it may also be useful to delete transfer files once they are no longer needed. The current solver keeps all transfer files, partially to allow for a full record of all needed transfers. Additionally, the current solver ends up creating all solved shards in a single folder. When dealing with millions of shards, this ends up overloading the Savio cluster, preventing easy transfers from the cluster to other locations (even running `ls` to list all files ended up timing out the cluster). For Connect 4, data was manually transferred to separate folders, but modifying the solver to save shards in different folders would help with this significantly. In addition, the current solver restarts a solve from scratch each run. For games which take extremely long times (the Savio cluster has a 3 day limit on jobs), it would be necessary to restart the solve using already-computed shards.

The current solver uses the stack solver to solve individual shards. This is relatively fast, but can be improved by parallelizing a shard solve. Notably, the current solver treats cores on the same node as separate processes; if instead each node received a multithreaded program, that would allow for larger shards to be run per node, and thus reduce memory transfer costs.

This would also allow for specialization of nodes. In particular, the Savio cluster has access to several GPU nodes, which would be useable to solve single shards. A tiered approach would allow for an efficient use of a GPU core (as with most MapReducible programs), and would likely yield significant improvements for those cores. The main driver itself is designed in a way that would permit different types of nodes to be used, and would also automatically assign more work to faster cores. As a downside, both this and the multithreaded code would require a memory module that is threadsafe. Under the current solver, the array-based memory module is threadsafe (since no value is ever overwritten, and a missing value simply gets recomputed), but upgrading to a van Emde Boas tree would prevent this from working. Thus, a fully optimal solver might use a GPU solver (with array-based memory modules) on

some nodes, and the stack-based solver (with a van Emde Boas-based memory module) on others.

Players

Currently, the solver outputs a large database containing the states of all valid positions. These can be loaded as `playerdata` structs, but otherwise, this project has not yet developed an actual interactive player that would allow direct gameplay with Connect 4. The Gamescrafters group has already developed a frontend UI that would work well with Connect 4, so writing a program to connect the two interfaces should be feasible.

10.2 Extension and Expected Efficacy when Solving Other Impartial Games

The solver as provided was successful in solving (6,7,4) Connect 4 in under 6 hours with 960 cores. Based on results, we can estimate an upper limit of games with around 50- to 53-bit hash lengths for a reasonable solve (1-10 TiB compressed database size, and within the 72-hour job limit and 1000-core limit of a single Savio cluster job). In comparison, the current single-threaded solver is limited to around 40-bit hashes.

The following games are thus likely to be newly solvable by the shard solver. While many of these games can be solved more efficiently by specialized solvers, the shard solver allows for faster iteration of new games, thus reducing the development overhead necessary to experiment with small rules variations (such as **miserè** variants)

Domineering

The game Domineering [7] is a game played on a checkerboard. Players take turns placing nonoverlapping dominoes (horizontal dominoes for Player 1, vertical dominoes for Player 2), with the first player unable to move losing. It is fairly easy to define an efficient monotonic hash for this game (use one bit per square, with a 1 signifying that the spot is covered by a domino). It would likely be feasible to strongly solve Domineering 7x7, Domineering 5x10, or smaller.

It is useful to note that Domineering is a **decomposable** game; thus, the value of a position is less useful than determining a position's associated **surreal number** [17]. It should be feasible to modify the current solver to accommodate this metric, and to define children as sums of positions, such that the solver database need only store connected positions.

Quantified Boolean Formula

The Boolean Satisfiability problem (or SAT) is a classic decision problem to determine if there exist variable assignments that fulfill a boolean expression; for example, the follow-

ing is a SAT question which is satisfiable: $\exists a \exists b \exists c (a \vee b) \wedge (\neg a \vee \neg c) \wedge (\neg b \vee c)$. As the prototypical NP-complete problem, SAT is commonly studied in the context of puzzles and computational theory.

A similar problem is received by allowing for universal quantifiers as well as existential quantifiers. Thus, the following is satisfiable: $\forall a \exists b \exists c (a \vee b) \wedge (\neg a \vee \neg c) \wedge (\neg b \vee c)$, while the following is not satisfiable: $\exists a \forall b \exists c (a \vee b) \wedge (\neg a \vee \neg c) \wedge (\neg b \vee c)$. This is known as the Quantified Boolean Formula problem, or QBF [19]. QBFs can be expressed as acyclic games, by allowing Player 1 to choose values for existential quantifiers, and Player 2 to choose values for universal quantifiers. Then, a player 1 win indicates satisfiability, while a player 2 win indicates unsatisfiability. The resulting game is easily hashed by defining the start position as $0b000...1$ and each move as left shifting the previous position and adding a 0 or 1 (depending on if a variable is set to true or false). Further, the generalized QBF problem is known to be PSPACE-complete (that is, any problem solveable in polynomial space can be solved in polynomial time given a QBF oracle), thus gaining significant practical application. The current solver is likely able to solve QBF problems with up to 49 variables. Since many games are in PSPACE, this also allows for efficient solves of any PSPACE game that can be expressed as a QBF of 49 or fewer variables.

Connect 4 Generalizations

Two common generalizations exist for Connect 4: Removing the restriction of play (that is, allowing for moves in non-bottom positions), and changing the win condition. The former leads to games like Tic-Tac-Toe and Gomoku [23]; the primary challenge here is to construct an efficient way to translate between binary gameplay and a ternary hash. For the latter, games such as "TOOT and OTTO" [20] (where both players are allowed to play a "T" or an "O" per turn, and each player winning when a "TOOT" or "OTTO" is made, respectively) should be feasible. The likely maximum for the former generalization would be around 25-36 squares, while the latter would allow for similar sizes (ex. 6x7 or 5x8 "TOOT and OTTO"). Both generalizations can be applied at once, creating, for example, Hex.

Othello

Othello is one of the more challenging games to solve in a shard fashion. The game Othello [9] is played on a checkerboard, with a start position consisting of four pieces in the center in a checkerboard pattern. Players take turns placing pieces (black pieces for Player 1, white pieces for Player 2). Any pieces that get "sandwiched" by the newly placed piece and another already-placed piece of the same color along an orthogonal or diagonal get "captured" and change into the other color. Moves must capture at least one piece, and the game is won by having more pieces than your opponent at a position where neither side has a valid move.

Notably, this game is relatively tricky to find a space-efficient monotonic hash, since pieces can change from white to black and vice versa, multiple times over the course of a

game. We present instead a hash that admits a monotonic sharding for reasonable shard sizes.

Definition 17. The following hash function is defined on Othello games with an $2n \times 2n$ ($n \geq 2$) board, as a ternary string:

The top four trits are assigned to store the value of the four corners, in arbitrary order. These are set to 0 for empty cells, 1 for black cells, and 2 for white cells. The next eight trits are assigned to store the value of the 8 cells orthogonally adjacent to the corners. These are set according to the following rules:

- If the adjacent corner is empty, store as before (0 for empty, 1 for black, 2 for white).
- If the adjacent corner is nonempty, store instead 0 if empty, 1 if the cell is the opposite color of its adjacent corner, and 2 if the cell is the same color of its adjacent corner.

The next eight trits are assigned the next edge cells orthogonally adjacent to the previous eight cells, and a similar rule is used to set their values. This continues for the remaining cells on the edge. The remaining cells (non-edge cells) are stored in row-major order, using 0 for empty cells, 1 for black cells, and 2 for white cells.

Example. In figure 10.1, the top-right corner is white, and is therefore labelled 2. The piece directly left is the same color as this corner, so it is labelled 2 as well. The piece directly below the corner is the opposite color, so it is labelled 1. The piece two spaces below the top-right corner is the opposite color of the piece directly below the corner, so it is labelled 1 as well. The same happens for the other corners, with a piece after an empty cell "resetting" to using 1 for black and 2 for white.

Theorem 5. *The hash defined above is shard-monotonic for shard sizes of length at least* $(2(n-1))^2$ *trits.*

Proof. The key property used here is that cells in the corner of Othello are guaranteed never to change from white to black or vice versa, since they are impossible to sandwich. Further, edge cells that are orthogonally adjacent to the corners can only change color if either:

- The corner is taken by a piece. In this case, the hash is guaranteed to increase, since all corners are stored in a more significant position than its orthogonally-adjacent positions
- The corner is already held by a piece, and the current cell is the opposite color of the corner piece. In this case, the trit corresponding to that cell increases, per its definition.

Inductively, this can be extended to the remaining pieces on the edge. Thus, the hash value of the edge is monotonic, so a shard size at least the size of the internal space permits a monotonic sharding. □

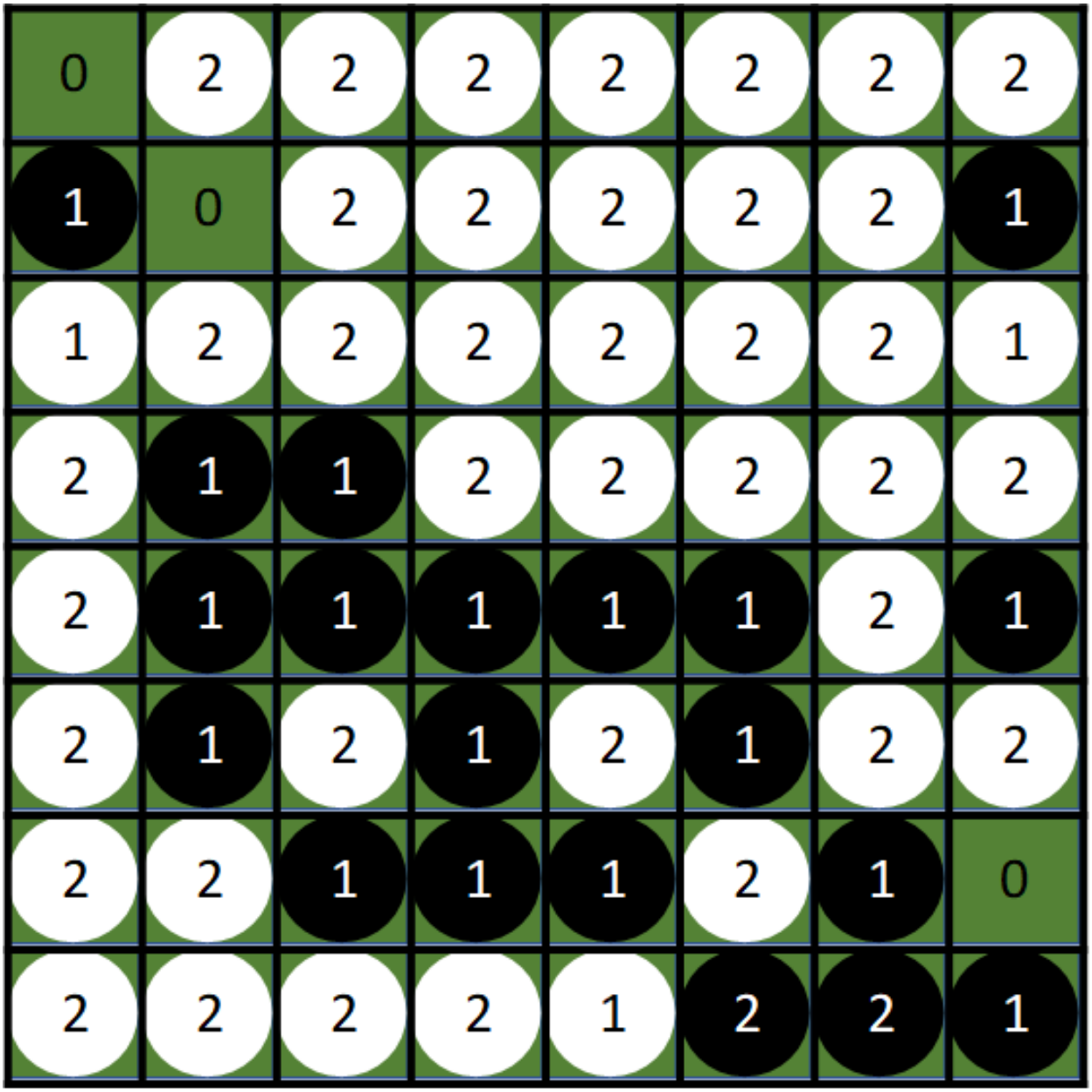

Figure 10.1: Example Othello position. The trit associated with each piece is 1 for black and 2 for white, except on the edge of the board. For pieces on the edge, the trit's value depends on the color of its adjacent pieces, moving towards the corner.

It is useful to note that this approach is not extensible to the center. The cells diagonally adjacent from the corners would be the next candidate to be stored, but a piece in that spot is able to change in response to an inner piece matching with any of the three adjacent edge cells. Thus, this hash is not monotonic, and cannot easily be extended to a monotonic hash.

Small improvements can be made to the hash by noting that the center pieces are guaranteed to be filled; thus, their values can be stored as bits instead of trits. Even then, Othello 6x6 requires a hash length of 55 bits and a shard size greater than 26 bits. This is slightly beyond the current feasible level (taking about two weeks to solve with the current solver), but will likely reach feasibility with a few of the above optimizations implemented. Solving this game would thus serve as a useful target for further development of the shard solver.

Bibliography

[1] URL: https://docs-research-it.berkeley.edu/services/high-performance-computing/user-guide/.

[2] URL: https://docs-research-it.berkeley.edu/services/high-performance-computing/user-guide/data/transferring-data/.

[3] *7 Billion Humans*. 2018. URL: http://tomorrowcorporation.com/7billionhumans.

[4] James Allen. *The Complete Book of Connect 4*. Sterling Publishing, 2010.

[5] Victor Allis. "A knowledge-based approach to connect-four. The game is solved: White wins". In: *Master's thesis, Vrije Universiteit*. 1988, p. 4.

[6] Zach Barth. *TIS-100*. 2015. URL: https://www.zachtronics.com/tis-100/.

[7] D.M. Breuker, J.W.H.M. Uiterwijk, and H.J. van den Herik. "Solving 8×8 Domineering". In: *Theoretical Computer Science* 230.1 (2000), pp. 195–206. ISSN: 0304-3975. DOI: https://doi.org/10.1016/S0304-3975(99)00082-1. URL: https://www.sciencedirect.com/science/article/pii/S0304397599000821.

[8] Jeffrey Dean and Sanjay Ghemawat. "MapReduce: Simplified Data Processing on Large Clusters". In: *Commun. ACM* 51.1 (Jan. 2008), pp. 107–113. ISSN: 0001-0782. DOI: 10.1145/1327452.1327492. URL: https://doi.org/10.1145/1327452.1327492.

[9] Joel Feingstein. *Perfect play in 6x6 Othello from two alternative starting positions*. URL: https://web.archive.org/web/20091101013931/http://www.feinst.demon.co.uk/Othello/6x6sol.html.

[10] Edgar Gabriel et al. "Open MPI: Goals, Concept, and Design of a Next Generation MPI Implementation". In: *Proceedings, 11th European PVM/MPI Users' Group Meeting*. Budapest, Hungary, Sept. 2004, pp. 97–104.

[11] *GamesCrafters*. URL: https://nyc.cs.berkeley.edu/wiki/Projects.

[12] *Gamesman Classic*. URL: https://github.com/GamesCrafters/GamesmanClassic.

[13] *Gamesman Java*. URL: https://github.com/GamesCrafters/GamesmanJava.

[14] Ralph Gasser. "Solving Nine Men's Morris". In: *Games of No Chance* 29 (1996), pp. 101–113.

[15] Maurice Herlihy and Nir Shavit. *The Art of Multiprocessor Programming, Revised Reprint*. Elsevier Science, 2013, p. 14.

[16] Michael Klemm and Jim Cownie. *High Performance Parallel Runtimes: Design and Implementation*. De Gruyter Oldenbourg, 2021. ISBN: 9783110632729. DOI: doi:10.1515/9783110632729. URL: https://doi.org/10.1515/9783110632729.

[17] Donald Ervin Knuth. *Surreal numbers : how two ex-students turned on to pure mathematics and found total happiness : a mathematical novelette*. Addison-Wesley Pub. Co., 1974.

[18] NVIDIA, Péter Vingelmann, and Frank H.P. Fitzek. *CUDA, release: 10.2.89*. 2020. URL: https://developer.nvidia.com/cuda-toolkit.

[19] Markus Rabe. *CADET*. URL: https://github.com/MarkusRabe/cadet.

[20] *Toot and Otto*. URL: https://www.ocf.berkeley.edu/~gamers/games.php?game=tootandotto.

[21] John Tromp. *John's Connect Four Playground*. URL: http://tromp.github.io/c4/c4.html.

[22] John Tromp. *The Fhourstones Benchmark (version 3.1)*. URL: http://tromp.github.io/c4/fhour.html.

[23] Chu-Hsuan Hsueh Wei-Yuan Hsu Chu-Ling Ko and I-Chen Wu. "Solving 7,7,5-game and 8,8,5-game". In: *ICGA Journal* 40.3 (2018), pp. 246–257. DOI: 10.3233/ICG-180061.

[24] Dan E. Willard. "Log-logarithmic worst-case range queries are possible in space (N)". In: *Information Processing Letters* 17.2 (1983), pp. 81–84. ISSN: 0020-0190. DOI: https://doi.org/10.1016/0020-0190(83)90075-3. URL: https://www.sciencedirect.com/science/article/pii/0020019083900753.

Appendix A

Code

Due to code reuse, many files under the same interface are nearly identical, and thus their inclusion in this appendix would be of limited utility. As such, several files (mostly early versions of the memory module) have been omitted from this document. The full code base is available at:
`https://github.com/GamesCrafters/GamesmanClassic/tree/master/Fa21ParallelSolver`
The files presented here are listed in the order they are described in Chapter 5.

A.1 solversinglethreaded.c

```
#include "Game.h"
#include "memory.h"


int main(int argc, char** argv)
{
    if(argc != 2)
    {
        printf("Usage: %s <filename>", argv[0]);
        return 1;
    }
      initialize_constants();
    game pos = getStartingPositions();
    solverdata* primitives = initializesolverdata(hashLength());
    game* fringe = calloc(sizeof(game), getMaxMoves()*getMaxDepth());
    if(primitives == NULL || fringe == NULL) {
        printf("Memory allocation error\n");
        return 1;
    }
```

```
int index = 1;
fringe[0] = getStartingPositions();
game g;
game newg;
gamehash h;
char primitive;
char minprimitive;
char moves[getMaxMoves()];
int movecount;
int i;
int oldindex;
gamehash minindex = maxHash();
gamehash maxindex = 0;
int* positionsfound = calloc(sizeof(int), 256);
printf("Beginning main loop\n");
fflush(stdout);
while(index)
{
    minprimitive = 255;
    oldindex = index;
    g = fringe[index-1];
    h = getHash(g);
    if(solverread(primitives, h)== 0)
    {
    movecount = generateMoves((char*)&moves, g);
    for(i=0;i<movecount;i++)
    {
        newg = doMove(g, moves[i]);
        h = getHash(newg);
        primitive = solverread(primitives, h);
        if(!primitive)
        {
            primitive = isPrimitive(newg, moves[i]);
            if(primitive != (char) NOT_PRIMITIVE) {
                solverinsert(primitives,h,primitive);
                if(h < minindex) {minindex = h;}
                if(h > maxindex) {maxindex = h;}
                //printf("Position 0x%08x determined primitive\n",
                    newg);
                positionsfound[primitive]++;
                minprimitive = minprimitive <= primitive
                    ? minprimitive : primitive;
```

```
                }
                else {
                    fringe[index] = newg;
                    index++;
                }
            }
            else
            {
                minprimitive = minprimitive <= primitive
                    ? minprimitive : primitive;
            }
        }
        if(index == oldindex)
        {
            if(minprimitive & 128)
            {
                if(minprimitive & 64) minprimitive = 257-minprimitive;
                else minprimitive = minprimitive + 1;
            }
            else minprimitive = 255-minprimitive;
            h = getHash(g);
            solverinsert(primitives,h,minprimitive);
            if(h < minindex) minindex = h;
            if(h > maxindex) maxindex = h;
            //printf("Position 0x%08x determined fully\n", g);
            positionsfound[minprimitive]++;
            index--;
        }
        }
        else { index--;}
    }
    printf("Done computing, listing statistics\n");
    fflush(stdout);
    int totalpositioncount = 0;
    for(int i = 1; i < 64; i++)
    {
        if(positionsfound[i])
            printf("Loss in %d: %d\n", i-1, positionsfound[i]);
        totalpositioncount+=positionsfound[i];
    }
    for(int i = 128; i < 192; i++)
    {
```

```
        if(positionsfound[i])
            printf("Tie in %d: %d\n", i-128, positionsfound[i]);
        totalpositioncount+=positionsfound[i];
    }
    for(int i = 254; i >= 192; i--)
    {
        if(positionsfound[i])
            printf("Win in %d: %d\n", 255-i, positionsfound[i]);
        totalpositioncount+=positionsfound[i];
    }
    printf("In total, %d positions found\n", totalpositioncount);
    printf("%d primitive positions found\n",
        positionsfound[LOSS]+positionsfound[TIE]);
    printf("Starting position has value %d\n",
        solverread(primitives, getStartingPositions()));
    printf("%llx, %llx\n", minindex, maxindex);
    printf("Starting output to file %s\n", argv[1]);
    fflush(stdout);
    FILE* file = fopen(argv[1], "wb");
    solversave(primitives, file);
    fclose(file);
    printf("Initializing player data\n");
    fflush(stdout);
    playerdata* p = initializeplayerdata(hashLength(), argv[1]);
    printf("Done initializing\n");
    fflush(stdout);
    if(verifyPlayerData(primitives, p)) {
    \tprintf("Success\n");
    }
    else {
        printf("Failure\n");
    }
    freesolver(primitives);
    printf("Done outputting\n");
    fflush(stdout);
    free(fringe);
    free(positionsfound);
    return 0;
}
```

A.2 maindriversinglethreaded.c

```
#include "Game.h"
#include "memory.h"
#include "solver.h"

#define shardsize 28

//Sends a message to all children/parents that the shard is done computing,
//and adds workable shards to the work queue
//issolved is 0 if during discovery, and 1 if during solving.
static shardgraph* addshardstoqueue(shardgraph* bottomshard,
    shardgraph* completedshard, int issolved) {
    if(issolved) {
        for(int i = 0 ; i < completedshard->parentcount;i++) {
            shardgraph* parentshard = completedshard->parentshards[i];
            parentshard->childrensolved++;
            if(parentshard->childrensolved == parentshard->childrencount)
            {
                //printf("Added shard %d to queue for solving\n",
                //  parentshard->shardid);
                bottomshard->nextinqueue = parentshard;
                bottomshard = parentshard;
            }
        }
    }
    else {
        for(int i = 0 ; i < completedshard->childrencount;i++) {
            shardgraph* childshard = completedshard->childrenshards[i];
            childshard->parentsdiscovered++;
            if(childshard->parentsdiscovered == childshard->parentcount)
            {
                //printf("Added shard %d to queue for discovery\n",
                //  childshard->shardid);
                bottomshard->nextinqueue = childshard;
                bottomshard = childshard;
            }
        }
        if(completedshard->childrencount == 0) {
            //printf("Added shard %d to queue for solving\n",
            //  completedshard->shardid);
            bottomshard->nextinqueue = completedshard;
```

```
            bottomshard = completedshard;
        }
    }
    return bottomshard;
}


int main(int argc, char** argv)
{
    if(argc != 2)
    {
        printf("Usage: %s <foldername>", argv[0]);
        return 1;
    }
      initialize_constants();

      shardgraph* shardList;
    int validshards = initializeshardlist(&shardList, shardsize);
    printf("Shard graph computed: %d shards will be computed\n",
        validshards);
    fflush(stdout);


    int shardsdiscovered = 0;
    int shardssolved = 0;
    shardgraph* topshard = getstartingshard(shardList, shardsize);
    shardgraph* bottomshard = topshard;
    //pointers to front and back of work queue
    char* workingfolder = argv[1];


    printf("Discovering shard %d/%d with shard id %d\n", shardsdiscovered,
        validshards, (topshard)->shardid);
    fflush(stdout);
    discoverfragment(workingfolder, topshard, shardsize, true);
    //Initialize work queue and compute first shard
    shardsdiscovered++;
    bottomshard = addshardstoqueue(bottomshard, bottomshard, 0);
    topshard->discovered++;
    shardgraph* oldtopshard = topshard;
    topshard = topshard->nextinqueue;
    oldtopshard->nextinqueue = NULL;
```

```
    //Compute remaining shards
    while(topshard!= NULL) {
        oldtopshard = topshard;
        if(oldtopshard->discovered) {
            printf("Solving shard %d/%d with shard id %d\n", shardssolved,
                validshards, oldtopshard->shardid);
            fflush(stdout);
            //if(shardssolved>=251)
            solvefragment(workingfolder, oldtopshard, shardsize,
                oldtopshard == getstartingshard(shardList, shardsize));
            fflush(stdout);
            shardssolved++;
        }
        else {
            printf("Discovering shard %d/%d with shard id %d\n",
                shardsdiscovered, validshards, oldtopshard->shardid);
            fflush(stdout);
            discoverfragment(workingfolder, oldtopshard, shardsize, false);
            shardsdiscovered++;
        }
        bottomshard = addshardstoqueue(bottomshard, oldtopshard,
            oldtopshard->discovered);
        topshard = topshard->nextinqueue;
        oldtopshard->nextinqueue = NULL;
        oldtopshard->discovered++;

    }
    printf("Done computing\n");
    freeshardlist(shardList, shardsize); //Clean up
}
```

A.3 maindriveropenmp.c

```
#include "Game.h"
#include "memory.h"
#include "solver.h"
#include <omp.h>
#include <unistd.h>
#include <time.h>
```

```
#define shardsize 26

//Sends a message to all children/parents that the shard is done computing,
//and adds workable shards to the work queue
//issolved is 0 if during discovery, and 1 if during solving.
static void addshardstoqueue(shardgraph** topshard,
    shardgraph** bottomshard, shardgraph* completedshard, int issolved) {
    if (issolved) {
        for(int i = 0 ; i < completedshard->parentcount;i++) {
            shardgraph* parentshard = completedshard->parentshards[i];
            parentshard->childrensolved++;
            if (parentshard->childrensolved ==
                parentshard->childrencount) {
                // printf("Added shard %d to queue for solving\n",
                //    parentshard->shardid);
                // fflush(stdout);
                if (*topshard == NULL) {
                    *topshard = parentshard;
                    *bottomshard = parentshard;
                } else {
                    (*bottomshard)->nextinqueue = parentshard;
                    *bottomshard = parentshard;
                }
            }
        }
    } else {
        for(int i = 0 ; i < completedshard->childrencount;i++) {
            shardgraph* childshard = completedshard->childrenshards[i];
            childshard->parentsdiscovered++;
            if (childshard->parentsdiscovered == childshard->parentcount) {
                // printf("Added shard %d to queue for discovery\n",
                //    childshard->shardid);
                // fflush(stdout);
                if (*topshard == NULL) {
                    *topshard = childshard;
                    *bottomshard = childshard;
                } else {
                    (*bottomshard)->nextinqueue = childshard;
                    *bottomshard = childshard;
                }
            }
```

```
        }
        if (completedshard->childrencount == 0) {
            // printf("Added shard %d to queue for solving\n",
            //    completedshard->shardid);
            // fflush(stdout);
            if (*topshard == NULL) {
                *topshard = completedshard;
                *bottomshard = completedshard;
            } else {
                (*bottomshard)->nextinqueue = completedshard;
                *bottomshard = completedshard;
            }
        }
    }
}


int main(int argc, char** argv) {
    if (argc != 2) {
        printf("Usage: %s <foldername>", argv[0]);
        return 1;
    }

    clock_t start, end;
    double cpu_time_used;

    start = clock();

      initialize_constants();

    shardgraph* shardList;
    int validshards = initializeshardlist(&shardList, shardsize);
    printf("Shard graph computed: %d shards will be computed\n",
        validshards);
    fflush(stdout);

    int shardsdiscovered = 0;
    int shardssolved = 0;
    shardgraph* topshard = getstartingshard(shardList, shardsize);
    shardgraph* bottomshard = topshard;
    //pointers to front and back of work queue
    char* workingfolder = argv[1];
```

```
printf("Discovering shard %d/%d with shard id %d\n", shardsdiscovered,
    validshards, (topshard)->shardid);
fflush(stdout);
discoverfragment(workingfolder, topshard, shardsize, true);
//Initialize work queue and compute first shard
shardsdiscovered++;
addshardstoqueue(&topshard, &bottomshard, bottomshard, 0);
topshard->discovered++;
shardgraph* oldtopshard = topshard;
topshard = topshard->nextinqueue;
oldtopshard->nextinqueue = NULL;

// Lock information
omp_lock_t shardgraphLock;
omp_init_lock(&shardgraphLock);
omp_lock_t terminalLock;
omp_init_lock(&terminalLock);
bool isDone = false;

//Compute remaining shards
#pragma omp parallel private(oldtopshard)
while(!isDone) {
    if (topshard == NULL) {
        // printf("Thread %d found no shard, going to sleep\n",
            omp_get_thread_num());
        // fflush(stdout);
        sleep(1);
        continue;
    }
    omp_set_lock(&shardgraphLock);
    if (topshard == NULL) {
        omp_unset_lock(&shardgraphLock);
        continue;
    }
    oldtopshard = topshard;
    topshard = topshard->nextinqueue;
    oldtopshard->nextinqueue = NULL;
    omp_unset_lock(&shardgraphLock);
    if (oldtopshard->discovered) {
        omp_set_lock(&terminalLock);
        shardssolved++;
```

```
            printf("Solving shard %d/%d with shard id %llu by thread %d\n",
                shardssolved, validshards, oldtopshard->shardid,
                omp_get_thread_num());
            fflush(stdout);
            omp_unset_lock(&terminalLock);
            solvefragment(workingfolder, oldtopshard, shardsize,
                oldtopshard == getstartingshard(shardList, shardsize));
            isDone = oldtopshard == getstartingshard(shardList, shardsize);
        } else {
            omp_set_lock(&terminalLock);
            shardsdiscovered++;
            printf("Discovering shard %d/%d with shard id %llu by thread
                %d\n", shardsdiscovered, validshards, oldtopshard->shardid,
                omp_get_thread_num());
            fflush(stdout);
            omp_unset_lock(&terminalLock);
            discoverfragment(workingfolder, oldtopshard, shardsize, false);
        }
        omp_set_lock(&shardgraphLock);
        addshardstoqueue(&topshard, &bottomshard, oldtopshard,
            oldtopshard->discovered);
        oldtopshard->discovered++;
        omp_unset_lock(&shardgraphLock);
    }
    end = clock();
    cpu_time_used = ((double) (end - start)) / CLOCKS_PER_SEC;
    printf("Done computing\n");
    printf("Total time taken: %f seconds\n", cpu_time_used);
    fflush(stdout);
    freeshardlist(shardList, shardsize); //Clean up
}
```

A.4 maindrivermpi.c

```
#include "Game.h"
#include "memory.h"
#include "solver.h"
#include <mpi.h>
#include <unistd.h>
#include <time.h>
```

```
#define shardsize 28
//General form of this program: Process 0 will act as the main driver,
//request other processes to compute various shards, and output progress
//data to the terminal. All other child processes wait for work to be
//assigned from process 0.
//Communication protocol from driver tp child:
/*
-1: Not enough work currently available. Sleep for 1 second to wait for
more work.
-2: All work done. Terminate process.
0b00xxxx... : Discover shard 0bxxxx...
0b01xxxx... : Solve shard 0bxxxx...

Communication protocol from child to driver:

-1: No work done. Checking in for more work.
0b00xxxx... : Shard 0bxxxx done computing.
*/
#define NOT_ENOUGH_WORK (-1)
#define TERMINATE (-2)
#define send_discovery_request(a) (a)
#define send_solve_request(a) ((a) | 1ULL<<62)
#define getshardID(a) ((a) & 0x3FFFFFFFFFFFFFFFULL)
#define issolve(a) (((a) & 1ULL<<62)>0)

//Sends a message to all children/parents that the shard is done computing,
//and adds workable shards to the work queue
//issolved is 0 if during discovery, and 1 if during solving.
static void addshardstoqueue(shardgraph** topshard,
    shardgraph** bottomshard, shardgraph* completedshard, int issolved) {
    if (issolved) {
        for(int i = 0 ; i < completedshard->parentcount;i++) {
            shardgraph* parentshard = completedshard->parentshards[i];
            parentshard->childrensolved++;
            if (parentshard->childrensolved ==
                parentshard->childrencount) {
                // printf("Added shard %d to queue for solving\n",
                    parentshard->shardid);
                // fflush(stdout);
                if (*topshard == NULL) {
                    *topshard = parentshard;
```

```
                    *bottomshard = parentshard;
                } else {
                    (*bottomshard)->nextinqueue = parentshard;
                    *bottomshard = parentshard;
                }
            }
        }
    } else {
        for(int i = 0 ; i < completedshard->childrencount;i++) {
            shardgraph* childshard = completedshard->childrenshards[i];
            childshard->parentsdiscovered++;
            if (childshard->parentsdiscovered == childshard->parentcount) {
                // printf("Added shard %d to queue for discovery\n",
                    childshard->shardid);
                // fflush(stdout);
                if (*topshard == NULL) {
                    *topshard = childshard;
                    *bottomshard = childshard;
                } else {
                    (*bottomshard)->nextinqueue = childshard;
                    *bottomshard = childshard;
                }
            }
        }
        if (completedshard->childrencount == 0) {
            // printf("Added shard %d to queue for solving\n",
                completedshard->shardid);
            // fflush(stdout);
            if (*topshard == NULL) {
                *topshard = completedshard;
                *bottomshard = completedshard;
            } else {
                (*bottomshard)->nextinqueue = completedshard;
                *bottomshard = completedshard;
            }
        }
    }
}


int main(int argc, char** argv) {
    if (argc != 2) {
```

```
    printf("Usage: %s <foldername>\n", argv[0]);
    return 1;
}
MPI_Init(&argc, &argv); //Initialize the MPI environment (for multiple
//node work). All code between here and finalize gets run by all nodes.
int processID, clusterSize;
MPI_Comm_size(MPI_COMM_WORLD, &clusterSize);
MPI_Comm_rank(MPI_COMM_WORLD, &processID);

if(clusterSize<=1) {
    printf("Not enough nodes for MPI; use standard code\n");
    MPI_Finalize();
    return 1;
}

clock_t start, end;
double cpu_time_used; //Variables used for timekeeping. Technically
//only used by process 0, so can be optimized a bit here.
start = clock();

  initialize_constants(); //These constants are used in solving
  //positions, and so should be initialized from all processes.
char* workingfolder = argv[1];

shardgraph* shardList;
int validshards = initializeshardlist(&shardList, shardsize); //All
//processes should initialize the shard list; this shouldn't take much
//time, and isn't really worth it to parallelize.
if(processID == 0) { //Only process 0 should send messages to stdout.
    printf("Shard graph computed: %d shards will be computed\n",
        validshards);
    fflush(stdout);

    int shardsdiscovered = 0;
    int shardssolved = 0;
    shardgraph* topshard = getstartingshard(shardList, shardsize);
    shardgraph* bottomshard = topshard;
    //pointers to front and back of work queue

    printf("Discovering shard %d/%d with shard id %d\n",
        shardsdiscovered, validshards, (topshard)->shardid);
    fflush(stdout);
```

```
//First shard can't be parallelized, and process 0 needs to finish
//shard computing anyway, so let process 0 run discovery on
//starting fragment.
shardsdiscovered++;
discoverfragment(workingfolder, topshard, shardsize, true);
//Initialize work queue and compute first shard

addshardstoqueue(&topshard, &bottomshard, bottomshard, 0);
topshard->discovered++;
shardgraph* oldtopshard = topshard;
topshard = topshard->nextinqueue;
oldtopshard->nextinqueue = NULL;


while(true) {
    MPI_Status status;
    uint64_t shardcompleted;
    MPI_Recv(&shardcompleted, 1, MPI_UINT64_T, MPI_ANY_SOURCE, 0,
        MPI_COMM_WORLD,&status);
        //Receive a request from one child process for work
    if(shardcompleted!= -1) {
        //Some shard was completed. Update the work queue
        //Note: Assumes that shardList[shardcompleted] has
        //shardid of shardcompleted
        addshardstoqueue(&topshard, &bottomshard,
            shardList+shardcompleted,
            (shardList[shardcompleted]).discovered);
        (shardList[shardcompleted]).discovered++;
        if(shardList+shardcompleted == getstartingshard(shardList,
            shardsize)) {
            //The starting shard was worked on. This indicates that
            //it was solved (since discovery happened earlier), and
            //as such, the solve is complete. Begin termination.
            uint64_t response = TERMINATE;
            MPI_Send(&response, 1, MPI_UINT64_T, status.MPI_SOURCE,
                0, MPI_COMM_WORLD);
            //Send termination message. We will send termination
            //messages to remaining processes after the while loop.
            break;
        }
    }
    if (topshard == NULL) {
```

```
            // printf("Process %d has no work, going to sleep\n",
                status.MPI_SOURCE);
            // fflush(stdout);
            uint64_t response = NOT_ENOUGH_WORK;
            MPI_Send(&response, 1, MPI_UINT64_T, status.MPI_SOURCE, 0,
                MPI_COMM_WORLD);
            //If there's currently no work to do, send a waiting
            //message
            continue;
        }
        oldtopshard = topshard;
        topshard = topshard->nextinqueue;
        oldtopshard->nextinqueue = NULL;
        if (oldtopshard->discovered) {
            shardssolved++;
            printf("Solving shard %d/%d with shard id %llu by process
                %d\n", shardssolved, validshards, oldtopshard->shardid,
                status.MPI_SOURCE);
            fflush(stdout);
            uint64_t response =
                send_solve_request(oldtopshard->shardid);
            MPI_Send(&response, 1, MPI_UINT64_T, status.MPI_SOURCE, 0,
                MPI_COMM_WORLD);
        } else {
            shardsdiscovered++;
            printf("Discovering shard %d/%d with shard id %llu by
                process %d\n", shardsdiscovered, validshards,
                oldtopshard->shardid, status.MPI_SOURCE);
            fflush(stdout);
            uint64_t response =
                send_discovery_request(oldtopshard->shardid);
            MPI_Send(&response, 1, MPI_UINT64_T, status.MPI_SOURCE, 0,
                MPI_COMM_WORLD);
        }
    }
    printf("Done computing. Sending termination messages to remaining
        processes\n");
    int processesterminated = 1;
    //One process was terminated in the main loop
    while(processesterminated < (clusterSize - 1)) {
    //Process 0 doesn't need a termination message
        MPI_Status status;
```

```
            uint64_t shardcompleted;
            MPI_Recv(&shardcompleted, 1, MPI_UINT64_T, MPI_ANY_SOURCE, 0,
                MPI_COMM_WORLD,&status);
            uint64_t response = TERMINATE;
            MPI_Send(&response, 1, MPI_UINT64_T, status.MPI_SOURCE, 0,
                MPI_COMM_WORLD);
            processesterminated++;
        }
        end = clock();
        cpu_time_used = ((double) (end - start)) / CLOCKS_PER_SEC;
        printf("Total time taken: %f seconds\n", cpu_time_used);
        fflush(stdout);
    } else {
        uint64_t senddata = -1;
        //Done setting up shard graph. Send to main that this process is
        //ready to work.
        MPI_Send(&senddata, 1, MPI_UINT64_T, 0, 0, MPI_COMM_WORLD);
        while(true) {
            uint64_t parentmessage;
            MPI_Recv(&parentmessage, 1, MPI_UINT64_T, 0, 0,
                MPI_COMM_WORLD,MPI_STATUS_IGNORE);
            if(parentmessage == NOT_ENOUGH_WORK) {
                sleep(1);
                senddata = -1;
                MPI_Send(&senddata, 1, MPI_UINT64_T, 0, 0, MPI_COMM_WORLD);
            }
            else if(parentmessage == TERMINATE) {
                break;
            }
            else {
                uint64_t targetshardID = getshardID(parentmessage);
                shardgraph* targetshard = shardList+targetshardID;
                if(issolve(parentmessage)) {
                    solvefragment(workingfolder, targetshard, shardsize,
                        targetshard == getstartingshard(shardList,
                        shardsize));
                }
                else {
                    discoverfragment(workingfolder, targetshard, shardsize,
                        false);
                }
                MPI_Send(&targetshardID, 1, MPI_UINT64_T, 0, 0,
```

```
                    MPI_COMM_WORLD);
            }
        }
    }

    freeshardlist(shardList, shardsize); //Clean up
    MPI_Finalize();
}
```

A.5 solver.h

```
#include <stdio.h>
#include <stdbool.h>
#include <stdlib.h>
#include <stdint.h>
#include <string.h>
#include "Game.h"
#include "memory.h"

typedef struct shardgraph {
    uint64_t shardid;
    int childrencount;
    int parentcount;
    int parentsdiscovered;
    int childrensolved;
    struct shardgraph** childrenshards;
    struct shardgraph** parentshards;

    //For use in shard work queue. Owned by maindriver.c
    struct shardgraph* nextinqueue;
    int discovered;
} shardgraph;

void discoverfragment(char* workingfolder, shardgraph* targetshard,
    char fragmentsize, bool isstartingfragment);

void solvefragment(char* workingfolder, shardgraph* targetshard,
    char fragmentsize, bool isstartingfragment);

//Shard graph functions
```

```
shardgraph* getstartingshard(shardgraph* shardlist, int shardsize);

int initializeshardlist(shardgraph** shardlistptr, uint32_t shardsize);

void freeshardlist(shardgraph* shardlist, uint32_t shardsize);
```

A.6 solver.c

```
#include "solver.h"


void discoverfragment(char* workingfolder, shardgraph* targetshard,
    char fragmentsize, bool isstartingfragment) {
    uint64_t currentshardid = targetshard->shardid;

    // Gives list of children shards that we send discovery children
    // to that we send to
    int childrenshardcount = targetshard->childrencount;
    solverdata** childrenshards = malloc(sizeof(solverdata*) *
        childrenshardcount);

    game g;
    game newg;
    gamehash h;
    gamehash minindex = maxHash();
    gamehash maxindex = 0;
    char primitive;

    int index = 0;

    // Next Tier: Use CUDA Malloc when moving to GPU for "moves"
    // and "fringe"
    solverdata* localpositions = initializesolverdata(fragmentsize);
    for(int i = 0; i < childrenshardcount; i++) {
        childrenshards[i] = initializesolverdata(fragmentsize);
        if(childrenshards[i] == NULL) {
            printf("Memory allocation error\n");
            fflush(stdout);
            return;
        }
```

```
}
char moves[getMaxMoves()];
game* fringe = calloc(sizeof(game), getMaxMoves() * getMaxDepth());
if (localpositions == NULL || fringe == NULL) {
    printf("Memory allocation error\n");
    fflush(stdout);
    return;
}

// Add incoming Discovery states from parent
// Multiple top node in shard
//Set up the file name
int filenamemaxlength =
    strlen("/transfer-100000000-100000000-x-primitive")+1;
char* filename = malloc(sizeof(char)*
    (filenamemaxlength + strlen(workingfolder)));
if(filename == NULL) {
    printf("Memory allocation error\n");
    fflush(stdout);
    return;
}
strncpy(filename, workingfolder, strlen(workingfolder));
char* filenamewriteaddr = filename+strlen(workingfolder);
const int SHARDOFFSETMASK = (1ULL << fragmentsize) - 1;
if(isstartingfragment) {
    fringe[0] = getStartingPositions();
    index = 1;

    int movecount, k, oldindex, newpositionshard;
    while (index) {
        oldindex = index;
        g = fringe[index - 1];
        index--; //For discovery, there's no need to keep the current
        // position on the fringe, since we don't need to go back to it.
        h = getHash(g);
        if (solverread(localpositions, h&(SHARDOFFSETMASK)) == 0) {
        //If we don't find this position in our solver, expand it for
        //discovery
            solverinsert(localpositions, h&(SHARDOFFSETMASK), 1);
            //Insert 1 here to that position to signify that it has
            //been expanded. Do this now to minimize the time other
            //threads can access this and try to duplicate work.
```

```
                movecount = generateMoves((char*) &moves, g);
                for (k = 0; k < movecount; k++) {
                //Iterate through all children of the current position
                    newg = doMove(g, moves[k]);
                    h = getHash(newg);
                    newpositionshard = h >> fragmentsize;
                    if(newpositionshard == currentshardid && !(solverread(
                        localpositions, h&SHARDOFFSETMASK))) {
                        //If we have a position in our current shard that
                        hasn't been expanded, add it to the fringe
                        if(isPrimitive(newg, moves[k]) == NOT_PRIMITIVE)
                            fringe[index++] = newg;
                        else
                            solverinsert(localpositions,
                                h&(SHARDOFFSETMASK), 2);
                    }
                    else if(newpositionshard != currentshardid) {
                    // If the child is not in the current shard, insert
                    // it into the appropriate child shard
                        for(int l = 0; l < targetshard->childrencount;l++) {
                            if(newpositionshard ==
                                targetshard->childrenshards[l]->shardid) {
                                if(!solverread(childrenshards[l],
                                    h&SHARDOFFSETMASK))
                                    solverinsert(childrenshards[l],
                                        h&SHARDOFFSETMASK,
                                        isPrimitive(newg, moves[k]));
                                break;
                            }
                        }
                    }
                }
            }
        }
    else {
    //Find all children in childrenshards
        for (int i = 0; i < targetshard->parentcount; i++) {
            //Open the file and read the first word to determine how many
            children are to be read
            snprintf(filenamewriteaddr,filenamemaxlength,
```

```
    "/transfer-%llu-%llu", targetshard->parentshards[i]->
    shardid, targetshard->shardid);
FILE* parentfile = fopen(filename, "rb");
if(parentfile == NULL) {
    printf("File Open Error: %s\n", filename);
    fflush(stdout);
    return;
}
int positioncountfromparent;
fread(&positioncountfromparent, sizeof(int), 1, parentfile);
//printf("%d positions will be checked from parent %llu\n",
positioncountfromparent, targetshard->parentshards[i]->shardid);
for(int j = 0; j < positioncountfromparent; j++) {
    //For each position in the file, add it to the fringe and
    //run graph traversal from there
    uint32_t newpositionoffset;
    fread(&newpositionoffset, sizeof(uint32_t), 1, parentfile);
    if(solverread(localpositions, newpositionoffset)) continue;
    //If the position already is in the solver, we've already
    //traversed from there, so ignore the position.
    gamehash newpositionhash = (((uint64_t) targetshard->
        shardid) << fragmentsize) + newpositionoffset;
        //Otherwise, set up the game corresponding to
        //the saved offset.
    fringe[0] = hashToPosition(newpositionhash);
    index = 1;

    int movecount, k, oldindex, newpositionshard;
    while (index) {
        oldindex = index;
        g = fringe[index - 1];
        index--; //For discovery, there's no need to keep the
        //current position on the fringe, since we don't need
        //to go back to it.
        h = getHash(g);
        if (solverread(localpositions, h&(SHARDOFFSETMASK))
            == 0) {
        //If we don't find this position in our solver,
        //expand it for discovery
            solverinsert(localpositions, h&(SHARDOFFSETMASK),
            1);
            //Insert 1 here to that position to signify that
```

```
                //it has been expanded. Do this now to minimize
                //the time other threads can access this and try
                //to duplicate work.

                movecount = generateMoves((char*) &moves, g);
                for (k = 0; k < movecount; k++) { //Iterate through
                //all children of the current position
                    newg = doMove(g, moves[k]);
                    h = getHash(newg);
                    newpositionshard = h >> fragmentsize;
                    if(newpositionshard == currentshardid &&
                    !(solverread(localpositions,
                    h&SHARDOFFSETMASK))) {
                    //If we have a position in our current shard
                    //that hasn't been expanded, add it to the
                    //fringe
                        if(isPrimitive(newg, moves[k]) ==
                            NOT_PRIMITIVE)
                            fringe[index++] = newg;
                        else
                            solverinsert(localpositions, h&
                                (SHARDOFFSETMASK), 2);
                    }
                    else if(newpositionshard != currentshardid) {
                    // If the child is not in the current shard,
                    // insert it into the appropriate child shard
                        for(int l = 0;
                            l < targetshard->childrencount;l++) {
                            if(newpositionshard == targetshard->
                                childrenshards[l]->shardid) {
                                if(!solverread(childrenshards[l],
                                    h&SHARDOFFSETMASK))
                                    solverinsert(childrenshards[l],
                                        h&SHARDOFFSETMASK,
                                        isPrimitive(newg,
                                        moves[k]));
                                break;
                            }
                        }
                    }
                }
            }
```

```
                }
            }

            fclose(parentfile); //Clean up
        }
    }
    //Save all children to appropriate files
    for(int i = 0; i < childrenshardcount; i++) {
        snprintf(filenamewriteaddr,filenamemaxlength, "/transfer-%llu-%llu",
            currentshardid, targetshard->childrenshards[i]->shardid);
        FILE* childnonprimitivefile = fopen(filename, "wb");
        //Open non-primitive file
        snprintf(filenamewriteaddr,filenamemaxlength,
            "/transfer-%llu-%llu-l-primitive", currentshardid, targetshard->
            childrenshards[i]->shardid);
        FILE* childlossfile = fopen(filename, "wb"); //Open loss file
        snprintf(filenamewriteaddr,filenamemaxlength,
            "/transfer-%llu-%llu-t-primitive", currentshardid, targetshard->
            childrenshards[i]->shardid);
        FILE* childtiefile = fopen(filename, "wb"); //Open tie file
        int nppositionsfound = 0, losspositionsfound = 0,
            tiepositionsfound = 0;
        fwrite(&nppositionsfound, sizeof(int), 1, childnonprimitivefile);
        // Store a dummy space of 4 bytes to later save
        // the number of positions found
        fwrite(&nppositionsfound, sizeof(int), 1, childlossfile);
        // Store a dummy space of 4 bytes to later save
        // the number of positions found
        fwrite(&nppositionsfound, sizeof(int), 1, childtiefile);
        // Store a dummy space of 4 bytes to later save
        // the number of positions found
        for(uint32_t j = 0; j < 1<<fragmentsize; j++) {
        //Could probably be made more efficient, but I think this
        //setup is more easily parallelizable
            switch(solverread(childrenshards[i], j)) {
            //If the position was found in discovery, save it to the file
            //and increment the number of positions found
                case LOSS: {
                    fwrite(&j, sizeof(uint32_t), 1, childlossfile);
                    losspositionsfound++;
                    break;
                }
```

```
case TIE: {
    fwrite(&j, sizeof(uint32_t), 1, childtiefile);
    tiepositionsfound++;
    break;
}
case NOT_PRIMITIVE: {
    fwrite(&j, sizeof(uint32_t), 1, childnonprimitivefile);
    nppositionsfound++;
    //No break here; move to the code in unsavednonprimitive
}
case UNSAVED_NONPRIMITIVE: {
    //All children of this position don't need to be sent,
    //since they'll be computed by the parent anyway
    //Note that this removes all possible positions only
    //because the hash is strictly increasing.
    //That is, the children of a position always have
    //greater hash values than their parent
    //When moving to Othello (or the GPU shard solver), this
    //code should be done in a separate loop before
    //saving starts.
    g = hashToPosition((((uint64_t) targetshard->
        childrenshards[i]->shardid) << fragmentsize) + j);
    int movecount = generateMoves((char*) &moves, g);
    for (int k = 0; k < movecount; k++) {
    //Iterate through all children of the current position
        newg = doMove(g, moves[k]);
        h = getHash(newg);
        int newpositionshard = h >> fragmentsize;
        if(newpositionshard == targetshard->
            childrenshards[i]->shardid) {
            if(isPrimitive(newg, moves[k]) == NOT_PRIMITIVE)
            {
                solverinsert(childrenshards[i], h&
                    SHARDOFFSETMASK, UNSAVED_NONPRIMITIVE);
            }
            else
                solverinsert(childrenshards[i],
                    h&SHARDOFFSETMASK, UNSAVED_PRIMITIVE);
        }
    }
    break;
}
```

```
            }
        }
        //printf("%d positions found\n", positionsfound);
        fseek(childnonprimitivefile, 0, SEEK_SET);
        fwrite(&nppositionsfound, sizeof(int), 1, childnonprimitivefile);
        fseek(childlossfile, 0, SEEK_SET);
        fwrite(&losspositionsfound, sizeof(int), 1, childlossfile);
        fseek(childtiefile, 0, SEEK_SET);
        fwrite(&tiepositionsfound, sizeof(int), 1, childtiefile);
        fclose(childnonprimitivefile);
        fclose(childlossfile);
        fclose(childtiefile);
    }
    //Clean up
    free(filename);
    freesolver(localpositions);
    for(int i = 0; i < childrenshardcount; i++) {
        freesolver(childrenshards[i]);
    }
    free(childrenshards);
    free(fringe);
}

void solvefragment(char* workingfolder, shardgraph* targetshard,
    char fragmentsize, bool isstartingfragment)
{
    uint64_t currentshardid = targetshard->shardid;

    // Gives list of children shards that we send discovery children to that
    // we send to
    int childrenshardcount = targetshard->childrencount;

    playerdata** childrenshards = malloc(sizeof(playerdata*) *
        childrenshardcount);

    game g;
    game newg;
    gamehash h;
    gamehash minindex = maxHash();
    gamehash maxindex = 0;
    char primitive;
```

```
int index = 0;

// Next Tier: Use CUDA Malloc when moving to GPU for "moves"
// and "fringe"
solverdata* localpositions = initializesolverdata(fragmentsize);
int solvedshardfilenamemaxlength
= strlen("/solved-100000000")+1;
char* solvedshardfilename = malloc(sizeof(char)*
    (solvedshardfilenamemaxlength + strlen(workingfolder)));
strncpy(solvedshardfilename, workingfolder, strlen(workingfolder));
char* solvedshardfilenamewriteaddr = solvedshardfilename+
    strlen(workingfolder);
for(int i = 0; i < childrenshardcount; i++) {
    snprintf(solvedshardfilenamewriteaddr,solvedshardfilenamemaxlength,
        "/solved-%d", targetshard->childrenshards[i]->shardid);
    childrenshards[i] = initializeplayerdata(fragmentsize,
        solvedshardfilename);
    /*printf("Shard %d player loaded\n", targetshard->
        childrenshards[i]->shardid);
    fflush(stdout);*/
    if(childrenshards[i] == NULL) {
        printf("Memory allocation error\n");
        return;
    }
}
char moves[getMaxMoves()];
game* fringe = calloc(sizeof(game), getMaxMoves() * getMaxDepth());
if (localpositions == NULL || fringe == NULL) {
    printf("Memory allocation error\n");
    return;
}

// Add incoming Discovery states from parent
// Multiple top node in shard
//Set up the file name
int filenamemaxlength =
    strlen("/transfer-100000000-100000000-x-primitive")+1;
char* filename = malloc(sizeof(char)*(filenamemaxlength +
    strlen(workingfolder)));
strncpy(filename, workingfolder, strlen(workingfolder));
char* filenamewriteaddr = filename+strlen(workingfolder);
const int SHARDOFFSETMASK = (1ULL << fragmentsize) - 1;
```

```
if(isstartingfragment) {
    fringe[0] = getStartingPositions();
    index = 1;


    int movecount, k, oldindex, newpositionshard;
    char minprimitive;

    while (index) {
        /*if(itcount <100) {
            printf("Done with iteration %d with j=%d and miss=%d on
                position %llx with index %d\n", itcount, j, misscount,
                fringe[index-1], index);
            fflush(stdout);
        }
        itcount++;*/
        minprimitive = 255;
        oldindex = index;
        g = fringe[index-1];
        h = getHash(g);
        if(solverread(localpositions, h&(SHARDOFFSETMASK)) == 0)
        //If we don't find this position in our solver, expand it for
        //discovery
        {
            //misscount++;
        movecount = generateMoves((char*)&moves, g);
        for(k = 0; k < movecount; k++)
        //Iterate through all children of the current position
        {
            newg = doMove(g, moves[k]);
            h = getHash(newg);
            newpositionshard = h >> fragmentsize;
            if(newpositionshard != currentshardid) {
            //If the position isn't in our current shard, we've already
            //solved it. Read the corresponding data
                for(int l = 0; l < targetshard->childrencount;l++) {
                    if(newpositionshard ==
                        targetshard->childrenshards[l]->shardid) {
                        primitive = playerread(childrenshards[l],
                            h&SHARDOFFSETMASK);
                        /*if(primitive == 0)
```

```
                {
                    printf("Error in primitive value: position
                        %llx\n", h);
                    printf("Current fringe state: ");
                    for(int m = 0; m < index;m++)
                        printf("%llx ", fringe[m]);
                    printf("\n");
                    exit(1);
                }*/

                    break;
                }
                /*if(l == targetshard->childrencount - 1) {
                    printf("Shard not found in children: %d, %llx,
                        %llx->%llx\n", newpositionshard, h, g,
                        newg);
                    printf("Current fringe state: ");
                    for(int m = 0; m < index;m++) printf("%llx ",
                        fringe[m]);
                    printf("\n");
                    exit(1);
                }*/
            }
        }
        else primitive = solverread(localpositions,
            h&SHARDOFFSETMASK);
        //Otherwise, check its value in the current shard
        if(!primitive) //Since playerread guarantees nonzero
        //values, only goes through here if it's a local position
        {
            primitive = isPrimitive(newg, moves[k]);
            if(primitive != (char) NOT_PRIMITIVE) {
                solverinsert(localpositions,h&SHARDOFFSETMASK,
                    primitive);
                minprimitive = minprimitive <= primitive ?
                    minprimitive : primitive;
            }
            else {
                fringe[index] = newg;
                index++;
            }
        }
```

```
                else
                {
                    minprimitive = minprimitive <= primitive ?
                        minprimitive : primitive;
                }
            }
            if(index == oldindex)
            {
                if(minprimitive & 128)
                {
                    if(minprimitive & 64) minprimitive = 257-minprimitive;
                    else minprimitive = minprimitive + 1;
                }
                else minprimitive = 255-minprimitive;
                h = getHash(g);
                solverinsert(localpositions,h&SHARDOFFSETMASK,minprimitive);
                index--;
            }
            }
            else { index--;}
        }
    }
    else {
        //Find all children in childrenshards
        for (int i = 0; i < targetshard->parentcount; i++) {
            //Open the file and read the first word to determine how many
            //children are to be read
            int positioncountfromparent;

            //Insert the loss positions found from the parent shard
            snprintf(filenamewriteaddr,filenamemaxlength,
                "/transfer-%llu-%llu-l-primitive",
                targetshard->parentshards[i]->shardid,
                targetshard->shardid);
            FILE* childlossfile = fopen(filename, "rb"); //Open loss file
            fread(&positioncountfromparent, sizeof(int), 1, childlossfile);
            for(int j = 0; j < positioncountfromparent; j++) {
                uint32_t newpositionoffset;
                fread(&newpositionoffset, sizeof(uint32_t), 1,
                    childlossfile);
                solverinsert(localpositions, newpositionoffset, LOSS);
                //Insert is going to be faster than a read and check, so
```

```
    //might as well just insert directly.
}
fclose(childlossfile);

//Insert the tie positions found from the parent shard
snprintf(filenamewriteaddr,filenamemaxlength,
    "/transfer-%llu-%llu-t-primitive",
    targetshard->parentshards[i]->shardid,
    targetshard->shardid);
FILE* childtiefile = fopen(filename, "rb"); //Open tie file
fread(&positioncountfromparent, sizeof(int), 1, childtiefile);
for(int j = 0; j < positioncountfromparent; j++) {
    uint32_t newpositionoffset;
    fread(&newpositionoffset, sizeof(uint32_t), 1,
        childtiefile);
    solverinsert(localpositions, newpositionoffset, TIE);
}
fclose(childtiefile);
//Begin computing any nonprimitives found from the parent shard
snprintf(filenamewriteaddr,filenamemaxlength,
    "/transfer-%llu-%llu",
    targetshard->parentshards[i]->shardid,
    targetshard->shardid);
FILE* parentfile = fopen(filename, "rb");
fread(&positioncountfromparent, sizeof(int), 1, parentfile);
//int itcount = 0, misscount = 0;
for(int j = 0; j < positioncountfromparent; j++) {
    //For each position in the file, add it to the fringe and
    //run graph traversal from there
    uint32_t newpositionoffset;
    fread(&newpositionoffset, sizeof(uint32_t), 1, parentfile);
    if(solverread(localpositions, newpositionoffset)) continue;
    //If the position already is in the solver, we've already
    //traversed from there, so ignore the position.
    gamehash newpositionhash = (((uint64_t)
    targetshard->shardid) << fragmentsize) + newpositionoffset;
    //Otherwise, set up the game corresponding to the saved
    //offset.
    fringe[0] = hashToPosition(newpositionhash);
    index = 1;
```

```
int movecount, k, oldindex, newpositionshard;
char minprimitive;

while (index) {
    /*if(itcount <100) {
        printf("Done with iteration %d with j=%d and
            miss=%d on position %llx with index %d\n",
            itcount, j, misscount, fringe[index-1], index);
        fflush(stdout);
    }
    itcount++;*/
    minprimitive = 255;
    oldindex = index;
    g = fringe[index-1];
    h = getHash(g);
    if(solverread(localpositions,
        h&(SHARDOFFSETMASK)) == 0) //If we don't find this
        //position in our solver, expand it for discovery
    {
        //misscount++;
    movecount = generateMoves((char*)&moves, g);
    for(k = 0; k < movecount; k++)
    //Iterate through all children of the current position
    {
        newg = doMove(g, moves[k]);
        h = getHash(newg);
        newpositionshard = h >> fragmentsize;
        if(newpositionshard != currentshardid) { //If the
        //position isn't in our current shard, we've
        //already solved it. Read the corresponding data
            for(int l = 0; l < targetshard->childrencount;
                l++) {
                if(newpositionshard == targetshard->
                    childrenshards[l]->shardid) {
                    primitive = playerread(
                        childrenshards[l],
                        h&SHARDOFFSETMASK);
                    /*if(primitive == 0)
                    {
                        printf("Error in primitive value:
                            position %llx\n", h);
                        printf("Current fringe state: ");
```

```
                    for(int m = 0; m < index;m++)
                        printf("%llx ", fringe[m]);
                    printf("\n");
                    exit(1);
                }*/

                break;
            }
            /*if(l == targetshard->childrencount - 1) {
                printf("Shard not found in children:
                    %d, %llx, %llx->%llx\n",
                    newpositionshard, h, g, newg);
                printf("Current fringe state: ");
                for(int m = 0; m < index;m++)
                    printf("%llx ", fringe[m]);
                printf("\n");
                exit(1);
            }*/
        }
    }
    else primitive = solverread(localpositions,
        h&SHARDOFFSETMASK);
        //Otherwise, check its value in the
        //current shard
    if(!primitive) //Since playerread guarantees
    //nonzero values, only goes through here if
    //it's a local position
    {
        primitive = isPrimitive(newg, moves[k]);
        if(primitive != (char) NOT_PRIMITIVE) {
            solverinsert(localpositions,h&
                SHARDOFFSETMASK,primitive);
            minprimitive = minprimitive <= primitive ?
                minprimitive : primitive;
        }
        else {
            fringe[index] = newg;
            index++;
        }
    }
    else
    {
```

```
                        minprimitive = minprimitive <= primitive ?
                            minprimitive : primitive;
                    }
                }
                if(index == oldindex)
                {
                    if(minprimitive & 128)
                    {
                        if(minprimitive & 64) minprimitive =
                            257-minprimitive;
                        else minprimitive = minprimitive + 1;
                    }
                    else minprimitive = 255-minprimitive;
                    h = getHash(g);
                    solverinsert(localpositions,h&SHARDOFFSETMASK,
                        minprimitive);
                    index--;
                }
                }
                else { index--;}
            }
        }

        fclose(parentfile); //Clean up
    }
}


//Save shard
snprintf(solvedshardfilenamewriteaddr,solvedshardfilenamemaxlength,
    "/solved-%llu", currentshardid);
FILE* childfile = fopen(solvedshardfilename, "wb");
solversave(localpositions, childfile);
fclose(childfile);

//Clean up
free(filename);
free(solvedshardfilename);
freesolver(localpositions);
for(int i = 0; i < childrenshardcount; i++) {
    freeplayer(childrenshards[i]);
}
```

```
    free(childrenshards);
    free(fringe);
    return;
}



static int initializeshard(shardgraph* shardlist, char* shardinitialized,
    uint32_t shardsize, uint32_t startingshard) {
    if(!shardinitialized[startingshard]) {
        //printf("Initializing shard %d\n", startingshard);
        //fflush(stdout);
        shardinitialized[startingshard] = 1;
        uint64_t* childrenshards;
        int childrencount = getchildrenshards(&childrenshards, shardsize,
            startingshard);

        shardlist[startingshard].shardid = startingshard;
        shardlist[startingshard].childrencount = childrencount;
        //printf("Shard %d has %d children\n", startingshard,
            childrencount);
        shardlist[startingshard].childrenshards = calloc(childrencount,
            sizeof(shardgraph*));
        int subshardsadded = 1;
        for(int i = 0; i < childrencount; i++) {
            subshardsadded+= initializeshard(shardlist, shardinitialized,
                shardsize, childrenshards[i]);
            shardlist[startingshard].childrenshards[i] =
                shardlist+childrenshards[i];
            shardlist[childrenshards[i]].parentcount++;
        }
        free(childrenshards);
        return subshardsadded;
    }
    else return 0;
}
static void initializeparentshard(shardgraph* shardlist,
    shardgraph* startingshard) {
    if(!startingshard->parentshards) {
        startingshard->parentshards = calloc(startingshard->parentcount,
            sizeof(shardgraph*));
        startingshard->parentcount = 0;
```

```
        for(int i = 0; i < startingshard->childrencount; i++)
        {
            initializeparentshard(shardlist,
                startingshard->childrenshards[i]);
            (startingshard->childrenshards[i])->
                parentshards[(startingshard->childrenshards[i])->
                parentcount] = startingshard;
            ((startingshard->childrenshards[i])->parentcount)++;
        }
    }
}

shardgraph* getstartingshard(shardgraph* shardlist, int shardsize)
{
    return shardlist+(getHash(getStartingPositions()) >> shardsize);
}
//Initializes all relevant shards. Returns the number of shards
//actually created.
int initializeshardlist(shardgraph** shardlistptr, uint32_t shardsize) {
    uint32_t shardcount = 1 << (hashLength() - shardsize);
    shardgraph* shardlist = calloc(shardcount, sizeof(shardgraph));
    *shardlistptr = shardlist;
    char* shardinitialized = calloc(shardcount, sizeof(char));
    uint32_t startingshard = getHash(getStartingPositions()) >> shardsize;
    int validshards = initializeshard(shardlist, shardinitialized,
    shardsize, startingshard);
    initializeparentshard(shardlist, shardlist+startingshard);
    free(shardinitialized);
    return validshards;
}
void freeshardlist(shardgraph* shardlist, uint32_t shardsize) {
    uint32_t shardcount = 1 << (hashLength() - shardsize);
    for(int i = 0; i < shardcount; i++)
        if(shardlist[i].childrenshards != NULL) {
            free(shardlist[i].childrenshards);
            free(shardlist[i].parentshards);
        }
    free(shardlist);
}
```

A.7 Game.h

```
#include <stdio.h>
#include <stdbool.h>
#include <stdlib.h>
#include <stdint.h>

#define LOSS 1
#define TIE 128
#define WIN 255
#define NOT_PRIMITIVE 127
#define UNSAVED_PRIMITIVE 125
#define UNSAVED_NONPRIMITIVE 126

/*
The solver stores the win-loss record in the following format:
0: RESERVED (Currently in use in solver to indicate a position that
is not valid)
1: Loss in 0 moves
2: Loss in 1 move
3: Loss in 2 moves
...
63: Loss in 62 moves
64: Draw
65-124: UNUSED
125: RESERVED (Currently in use in solver to indicate a valid primitive that
doesn't need to be sent to a child shard)
126: RESERVED (Currently in use in solver to indicate a valid nonprimitive
that doesn't need to be sent to a child shard)
127: RESERVED (Currently in use in solver to indicate a non-primitive
position)
128: Tie in 0 moves
129: Tie in 1 move
...
191: Tie in 63 moves
192: Win in 63 moves
...
254: Win in 1 move
255: Win in 0 moves

Games with depth greater than 63 are not supported
*/
```

```
typedef uint64_t game;
typedef uint64_t gamehash;


void initialize_constants();

game getStartingPositions();

/** Returns the result of moving at the given position
    * Bit 63 is flipped to change current player
    * Move is assumed to be the bit position of the empty cell.
    * As such, we add 1<<move for a yellow, and 1<<(move+1) for red
    * 0b0001011 + 1<<move = 0b0010011 = ---YYRR
    * 0b0001011 + 1<<(move+1) = 0b0011011 = ---RYRR*/
game doMove(game position, char move);

/** Returns the list of viable moves.
Assumes retval has length at least MaxMoves*/
int generateMoves(char* retval, game position);
/** Returns if the given position is primitive, assuming that the most
recent move is as stated*/
char isPrimitive(game position, char mostrecentmove);


//int getSize();

int getMaxDepth();

int getMaxMoves();

gamehash getHash(game position);

gamehash maxHash();

int hashLength();

game hashToPosition(gamehash hash);

/*Returns a list of shard ids which the given parent shard has as children.
Ideally, this should be fast enough that the shard graph can be determined
by one thread. Return value is the number of children shards.*/
```

```
int getchildrenshards(uint64_t** childrenshards, char shardsize,
    uint64_t parentshard);
```

A.8 Connect4.c

```
#include "Game.h"

#define COLUMNCOUNT 5
#define ROWCOUNT 5
#define CONNECT 4

static uint64_t DOWNDIAGWIN;
static uint64_t HORIZONTALWIN;
static uint64_t VERTICALWIN;
static uint64_t UPDIAGWIN;
static uint64_t INITIALPOSITION;

static bool isawin(game position, game pieces);

void initialize_constants() {
    uint64_t l = 0;
    for(int i = 0; i < COLUMNCOUNT; i++) l |= 1ULL<<((ROWCOUNT+1)*i);
    uint64_t down=0, left=0, updiag=0, downdiag = 0;
    for(int i = 0; i < CONNECT;i++)
    {
        down = (down << 1)|1;
        left = (left << (ROWCOUNT+1))|1;
        downdiag = (downdiag << ROWCOUNT)|1;
        updiag = (updiag << (ROWCOUNT+2))|1;
    }
    DOWNDIAGWIN = downdiag;
     HORIZONTALWIN = left;
    VERTICALWIN = down;
    UPDIAGWIN = updiag;
    INITIALPOSITION = l;
}

game getStartingPositions() {
    return INITIALPOSITION;
}
```

```
/** Returns the result of moving at the given position
    * Bit 63 is flipped to change current player
    * Move is assumed to be the bit position of the empty cell.
    * As such, we add 1<<move for a yellow, and 1<<(move+1) for red
    * 0b0001011 + 1<<move = 0b0010011 = ---YYRR
    * 0b0001011 + 1<<(move+1) = 0b0011011 = ---RYRR*/
game doMove(game position, char move) {
    return (position ^ 0x8000000000000000L)+(1ULL<<(move+(position >> 63)));
}

/** Returns the list of viable moves.
Assumes retval has length at least COLUMNCOUNT+1*/
int generateMoves(char* retval, game position) {
    int k = 0;
    for(int i = 0; i < COLUMNCOUNT; i++)
    {
        char start = (char)((ROWCOUNT+1)*(i+1) - 1);
        while((position & (1ULL<<start))==0)
            start--; // this is what sigbit does
        if(start != (char)(ROWCOUNT+1)*(i+1)-1)
        {
            *retval = start;
            retval++;
            k++;
        }
    }
    *retval = -1;
    return k;
}

/** Returns if the given position is primitive, assuming that the most
recent move is as stated*/
char isPrimitive(game position, char mostrecentmove) {
    game origpos = position;
    if((position & 0x8000000000000000L) == 0)
    { // Check wins of 1s
        for(int i = 0; i < COLUMNCOUNT; i++)
        {
            char start = (char)((ROWCOUNT+1)*(i+1) - 1);
            while((position & (1ULL<<start))==0) start--;
            position ^= (1ULL<<start);
        }
```

```
}
else
{ //Check wins of 0s
    for(int i = 0; i < COLUMNCOUNT; i++)
    {
        char start = (char)((ROWCOUNT+1)*(i+1) - 1);
        char start2 = (char)(start+1);
        while((position & (1ULL<<start))==0) start--;
        position |= (1ULL<<(start2))-(1ULL << start);
    }
    position = ~position;
}
position &= (1ULL<<(COLUMNCOUNT*(ROWCOUNT+1)))-1;
//System.out.printf("%016X %n", position);
//At this point, the position should contain 1s only on the places
that match the most recent move.
int x = mostrecentmove/(ROWCOUNT+1), y=mostrecentmove%(ROWCOUNT+1);
//Vertical check
if((y >= CONNECT-1)&&
isawin(position, VERTICALWIN<<(mostrecentmove-(CONNECT-1))))
    return LOSS;
//Horizontal and diagonal checks
for(int i = 0; i < CONNECT; i++) {
    if (x >= i && (x + ((CONNECT - 1) - i)) < COLUMNCOUNT) {
        //Horizontal check
        if (isawin(position, HORIZONTALWIN <<
            (mostrecentmove - i * (ROWCOUNT + 1)))) return LOSS;
        if (y >= i && (y + ((CONNECT - 1) - i)) < ROWCOUNT) {
            //Up Diagonal check
            if (isawin(position, UPDIAGWIN <<
                (mostrecentmove - i * (ROWCOUNT + 2)))) return LOSS;
        }
        if (y + i < ROWCOUNT && (y - ((CONNECT - 1) - i)) >= 0) {
            //Down Diagonal check
            if (isawin(position, DOWNDIAGWIN <<
                (mostrecentmove - i * (ROWCOUNT)))) return LOSS;
        }
    }
}
if((origpos & (INITIALPOSITION << ROWCOUNT)) ==
    INITIALPOSITION << ROWCOUNT) {return TIE;}
return NOT_PRIMITIVE;
```

```
}

static bool isawin(game position, game pieces)
{
    //System.out.printf("%016X %b %n", pieces, (position&pieces) == pieces);
    return (position&pieces) == pieces;
}

int getSize() {
    return COLUMNCOUNT*ROWCOUNT;
}

int getMaxDepth() {
    return getSize();
}

int getMaxMoves() {
    return COLUMNCOUNT+1;
}

gamehash getHash(game position) {/*
    game newpos= position & 0x7FFFFFFFFFFFFFFL;
    game oppositepos = 0;
    for(int i = 0; i < COLUMNCOUNT; i++)
    {
        uint64_t val = (newpos>>(i*(ROWCOUNT+1)))&((1ULL<<(ROWCOUNT+1))-1);
        oppositepos |= val<<((ROWCOUNT+1)*(COLUMNCOUNT-i-1));
    }
    return oppositepos < newpos ? oppositepos : newpos;*/
    return position & 0x7FFFFFFFFFFFFFFL;
}
uint64_t maxHash()
{
    return 1ULL<<((ROWCOUNT+1)*COLUMNCOUNT);
}
int hashLength()
{
    return ((ROWCOUNT+1)*COLUMNCOUNT);
}
game hashToPosition(gamehash hash) {
    char emptyspots = 0;
    for(int j = 0; j < COLUMNCOUNT; j++) {
```

```
        for (int i = ROWCOUNT; i >= 0; i--) {
            if((hash&(1ULL<<(j*(ROWCOUNT+1)+i))) == 0) emptyspots++;
            else break;
        }
    }
    return hash | ((gamehash) ((getSize()-emptyspots)%2)) << 63;
}

/*static void recurse(char* primitives, int* hashsetsize, game position,
    int layer, char* movestring) {
    if(layer < 5) {printf("%s\n", movestring); fflush(stdout);}
    char moves[COLUMNCOUNT+1];
    generateMoves(moves, position);
    for(int i = 0; i < COLUMNCOUNT;i++)
    {
        if(moves[i] == -1) break;
        game newpos = doMove(position, moves[i]);
        if(isPrimitive(newpos, moves[i]) != (char)NOT_PRIMITIVE)
        {
            gamehash poshash = getHash(newpos);
            if(!((primitives[poshash>>3])&(1<<(poshash&7))))
            {
                primitives[poshash>>3]|=1<<(poshash&7);
                (*hashsetsize)++;
            }
        }
        else
        {
            movestring[layer] = 48+i;
            recurse(primitives,hashsetsize, newpos, layer+1, movestring);
        }
    }
    movestring[layer] = 0;
}
static int testC4() {
    initialize_constants();
    game pos = getStartingPositions();
    char* primitives = calloc(sizeof(char),(1ULL<<
        ((COLUMNCOUNT*(ROWCOUNT+1))-3)));
    int hashsetsize = 0;
    char* movestring = calloc(sizeof(char), getSize()+1);
    recurse(primitives, &hashsetsize, pos, 0, movestring);
```

```
    printf("Number of primitives: %d", hashsetsize);
    free(primitives);
    free(movestring);
}*/


static inline uint64_t makeColumnMove(uint64_t shard, char move,
    char mode) {
    return shard + (1<<(move + mode));
}

int getchildrenshards(uint64_t** childrenshards, char shardsize,
    uint64_t parentshard) {
    // Column size is not 7
    int id_size = hashLength() - shardsize; // size in bits
    int full_cols = (id_size / (ROWCOUNT+1));
    int remainders = id_size % (ROWCOUNT+1);
    int length = 0;

    // Count the number of non-full 7-sized columns
    for (int i = 0; i < full_cols; i++) {
        if (!(parentshard&(1<<(id_size-1-((ROWCOUNT+1)*i))))) {
            length+=2;
        }
    }
    if(remainders) {
        // Count the number of non-full <7-sized columns
        int filter = (1<<remainders) - 1;
        if (!(parentshard&(1<<(remainders-1)))) {
            length += (parentshard & filter) ? 2 : 1;
        }
    }
    uint64_t* children = malloc(length * sizeof(uint64_t));
    int allocatedchildren = 0;
    int columnremainingbits = ROWCOUNT+1;
    for(int i = id_size - 1; i >= 0;) {
        if(parentshard&(1<<i)) {
            //printf("%d %d\n", i, columnremainingbits);
            if(columnremainingbits != ROWCOUNT+1) {
                children[allocatedchildren] = parentshard+(1<<i);
                children[allocatedchildren+1] = parentshard+(1<<(i+1));
                allocatedchildren+=2;
```

```
            }
            i-=columnremainingbits;
            columnremainingbits = ROWCOUNT+1;
        }
        else{
            columnremainingbits--;
            i--;
        }
    }
    if(allocatedchildren < length) {
        children[allocatedchildren] = parentshard+1;
        allocatedchildren++;
    }
    /*printf("Children found for shard %d: ", parentshard);
    for(int i = 0; i < allocatedchildren; i++) printf("%d ", children[i]);
    printf("\n");
    if(allocatedchildren != length) {
        printf("Error: incorrect children found: %d instead of %d\n",
            allocatedchildren, length);
    }*/
    *childrenshards = children; // no free
    return length;
}

/* KEENE'S PYTHON CODE CAUSE I WAS LAZY TO DEBUG IN C
ROWCOUNT = 6
COLUMNCOUNT = 7
def makeColumnMove(shard, move, mode):
return shard + (1<<(move + mode));
def hashLength():
    return ((ROWCOUNT+1)*COLUMNCOUNT);
def sigbit(shard, col, shardsize):
id_size = hashLength() - shardsize
remainders = id_size % (ROWCOUNT+1)
if (col == 0 and remainders == 0):
    return shard > 0
elif (col == 0):
    return shard & (1 << remainders - 1) > 0
return shard & (1 << (col * (ROWCOUNT+1) + remainders - 1)) > 0

def getchildrenshards(childrenshards, shardsize, parentshard):
id_size = hashLength() - shardsize
```

```
full_cols = (int)(id_size / (ROWCOUNT+1))
remainders = id_size % (ROWCOUNT+1)
length = 0
for i in range(1, full_cols+1):
    pp = parentshard
    if not sigbit(pp, i, shardsize):
        length+=2
filter = ((1<<remainders)-1)
if (not sigbit(parentshard, 0, shardsize) and parentshard & filter > 0):
    length+=2
elif (parentshard & filter <= 0 and not sigbit(parentshard, 0, shardsize)):
    length+=1
children = [0 for _ in range(length)]
i = id_size - 1;
added = 0
full_c_passed = 0
while (i >= 0 and added < length):
    if (full_c_passed >= full_cols and parentshard & filter <= 0):
            children[added] = parentshard+1
            added += 1
            break
    if ((parentshard>>i) & 1 > 0):
        if (((i+1)-remainders) % (ROWCOUNT + 1) == 0):
            i = ((i-2)//(ROWCOUNT+1)) * (ROWCOUNT+1) - 1 + remainders

            full_c_passed += 1
        else:
            children[added] = makeColumnMove(parentshard, i, 0)
            added += 1
            children[added] = makeColumnMove(parentshard, i, 1)
            added += 1
            full_c_passed += 1
            i = ((i-1)//(ROWCOUNT+1)) * (ROWCOUNT+1) - 1 + remainders
    else:
        i-=1
return children, length
def print_b(num):
print("{0:b}".format(num))
# print_b(1)
# print_b(makeColumnMove(1, 0, 1))
# print_b(makeColumnMove(1, 0, 0))
c, l = getchildrenshards([], 33, 0b1100011110001101)
```

```
print([bin(k) for k in c])
# print(c)
# print(l)
# 4 --> 2
print("YO",((4-1)//(6+1)) * (6+1) + 2)

*/
```

A.9 memory.h

```
#include <stdio.h>
#include <stdbool.h>
#include <stdlib.h>
#include <stdint.h>

/*Struct defined for solving a game*/
typedef struct solverdata solverdata;
/*Struct defined for playing a game*/
typedef struct playerdata playerdata;

/*Initializes the data structure for solver, which acts as a dictionary with
keys as keylen-bit integers and value of one (nonzero) byte each.
Returns NULL if error, and assumes keylen <= 64*/
solverdata* initializesolverdata(int keylen);

/*Inserts the given (key,val) pair to a solver. It is assumed that any
insert either occurs only once
or occurs with the same value for a given key, and assumes that val != 0*/
void solverinsert(solverdata* data, uint64_t key, unsigned char val);


/*Reads a data value at the given key, returning 0 if the value has not
been inserted*/
unsigned char solverread(solverdata* data, uint64_t key);


/*Writes to the given file (assumed to be opened and readable) the current
solver state. Is not required to leave the solverdata object unmodified.
The output file is designed to be used with the playerdata object,
but only guarantees values for stored keys; any key not set is set to
```

```
a random value*/
void solversave(solverdata* data, FILE* fp);

/*Frees a solver*/
void freesolver(solverdata* data);

/*Initializes the data structure for a player, read from a given filename*/
playerdata* initializeplayerdata(int keylen, char* filename);

/*Reads a data value at the given key. Returns a random value if the key
had not received a defined value in the corresponding solver.*/
unsigned char playerread(playerdata* data, uint64_t key);

/*Frees a player*/
void freeplayer(playerdata* data);

bool verifyPlayerData(solverdata* sd, playerdata* pd);
```

A.10 memoryfastretrieval.c

```
#include "memory.h"
#include <string.h>


/*Struct defined for solving a game*/
struct solverdata
{
    unsigned char size;
    unsigned char data[];
};

/*Struct defined for playing a game*/
struct playerdata {
    unsigned char size;
    unsigned char data[];
};

/*Initializes the data structure for solver, which acts as a dictionary with
keys as keylen-bit integers and value of one (nonzero) byte each.
Returns NULL if error, and assumes keylen <= 64*/
solverdata* initializesolverdata(int keylen)
```

```
{
    solverdata* s = (solverdata*) calloc(1l + (1l << keylen), 1);
    if(!s) {return NULL;}
    s->size = keylen;
    return s;
}

/*Inserts the given (key,val) pair to a solver. It is assumed that any
insert either occurs only once or occurs with the same value for a given
key, and assumes that val != 0*/
void solverinsert(solverdata* data, uint64_t key, unsigned char val)
{
    data->data[key] = val;
}

/*Reads a data value at the given key, returning 0 if the value has not
been inserted*/
unsigned char solverread(solverdata* data, uint64_t key)
{
    return data->data[key];
}


/* Writes to output the shard of (2^SIZE) length. Returns -n if n is the
   unique nonzero value in the shard, 0 if the shard is empty (contains
   all zeros), or the number of bytes written otherwise.

   Format of this file structure:
   Each block of size 2^n begins with a pointer.
   If pointer == 0, then the next byte contains the unique value stored in
   the given block.
   If pointer == 1, then left and right blocks are identical, and what
   follows is the structure for a block of size 2^(n-1).
   Otherwise, the left and right blocks are different. Then the pointer
   lists the length of the left block. What follows is the left block,
   then the right block. Since all blocks contain at least one pointer
   of length at least one byte, and at least one byte of data, subblocks
   are at least 2 bytes long.
*/
static int64_t solversavefragment(int size, unsigned char* data,
    unsigned char* output) {
    /* Base case: data is 1 byte long. Return that inverse of that
```

```
    unique nonzero value. */
    if (size == 0) {
        return -*data;
    }
    int64_t leftlength = solversavefragment(size - 1, data, output + 1l);
    if (leftlength > 0) {
        /* Left tree contains multiple values. */
        int64_t rightlength = solversavefragment(size - 1, data +
            (1l << (size - 1)), output + 1l + leftlength);
        if (rightlength > 0) {
            /* Right tree contains multiple values. */
            output[0] = 2;
            return leftlength + rightlength + 1l;
        } else if (rightlength == 0) {
            /* Right tree contains all zeros, treat right tree as identical
            to the left tree. */
            output[0] = 1;
            return leftlength + 1l;
        } else {
            /* Right tree contains a unique nonzero value. */
            output[0] = 2;
            output[1l + leftlength] = 0;
            *(output + leftlength + 2l) = -rightlength;
            return leftlength + 3l;
        }
    } else if (leftlength == 0) {
        /* Left tree contains all zeros. */
        int64_t rightlength = solversavefragment(size - 1, data +
            (1l << (size - 1)), output + 1l);
        if (rightlength <= 0) {
            /* Right tree contains a unique value (possibly zero). */
            return rightlength;
        }
        /* Right tree contains multiple values, treat left tree as identical
            to the right tree. */
        output[0] = 1;
        return rightlength + 1l;
    } else {
        /* Left tree contains a unique nonzero value. */
        int64_t rightlength = solversavefragment(size - 1, data +
            (1l << (size - 1)), output + 3l);
        if (rightlength > 0) {
```

```
            /* Right tree contains multiple values. */
            output[0] = 2;
            output[1] = 0;
            output[2] = -leftlength;
            return rightlength + 3l;
        } else if (rightlength == 0) {
            /* Right tree contains all zeros. */
            return leftlength;
        } else {
            /* Right tree contains a unique nonzero value. */
            if (rightlength == leftlength) {
                return leftlength;
            }
            output[0] = 2;
            output[1] = 0;
            output[2] = -leftlength;
            output[3] = 0;
            output[4] = -rightlength;
            return 5l;
        }
    }
}

/*Writes to the given file (assumed to be opened and readable) the current
solver state.
Is not required to leave the solverdata object unmodified.
The output file is designed to be used with the playerdata object,
but only guarantees values for stored keys; any key not set is set to a
random value*/
void solversave(solverdata* data, FILE* fp)
{
    /* Why is this safe? */
    /*In any cases we care about, we'll get significant memory improvements
        anyway.*/
    unsigned char* result = calloc(1l << (data->size-2),
        sizeof(unsigned char));
    if (result == NULL) {
        printf("Memory allocation error\n");
        return;
    }
    int length = solversavefragment(data->size, data->data, result);
    fwrite(&(data->size), sizeof(unsigned char), 1, fp);
```

```
    if(length <= 0) {
        printf("Compression complete. New length: %d bytes\n", 2);
        result[0] = 0;
        result[1] = -length;
        fwrite(result, sizeof(unsigned char), 2, fp);
    } else {
        printf("Compression complete. New length: %d bytes\n", length);
        fwrite(result, sizeof(unsigned char), length, fp);
    }
    free(result);
}

/*Frees a solver*/
void freesolver(solverdata* data) {
    free(data);
}

static void initializesegment(char* data, FILE* file, int size) {
    int64_t ptr = 0;
    fread(&ptr, 1, 1, file);
    //printf("%llx %d\n", ptr, size);
    //fflush(stdout);
    if (ptr == 0) {
        char c;
        fread(&c, 1, 1, file);
        memset(data, c, 1l<<size);
    }
    else if(ptr == 1) {
        initializesegment(data, file, size-1);
        memcpy(data+(1l<<(size-1)), data, (1l<<(size-1)));
    }
    else {
        initializesegment(data, file, size-1);
        initializesegment(data+(1l<<(size-1)), file, size-1);
    }
}

/*Initializes the data structure for a player, read from a given filename*/
playerdata* initializeplayerdata(int keylen, char* filename)
{
    //printf("Here\n");
    //fflush(stdout);
```

```
    FILE* file = fopen(filename, "rb");
    if (!file) {
        return NULL;
    }
    playerdata* s = (playerdata*) calloc(1l + (1l << keylen), 1);
    //printf("%p\n", s);
    if(!s) {return NULL;}
    s->size = keylen;
    char size;
    fread(&size, sizeof(unsigned char), 1, file);
    //printf("%d %d\n", keylen, size);
    //fflush(stdout);
    initializesegment(s->data, file, size);
    fclose(file);
    return s;
}


/* Reads a data value at the given key. Returns a garbage value
   if the key had not received a defined value in the corresponding
   solver. */
unsigned char playerread(playerdata* data, uint64_t key) {
    return data->data[key];
}

/*Frees a player*/
void freeplayer(playerdata* data)
{
    free(data);
}

bool verifyPlayerData(solverdata* sd, playerdata* pd) {
    int size = (int)sd->size;
    for (uint64_t i = 0; i < (1ul << size); ++i) {
        if (i % 1000000 == 0) {
            printf("verifying %lld\n", i);
            fflush(stdout);
        }
        unsigned char solverVal = solverread(sd, i);
        if (solverVal != 0) {
            unsigned char playerVal = playerread(pd, i);
            if (solverVal != playerVal) {
```

```
                printf("inconsistent value at key %llx: solver value %d
                    and player value %d\n", i, solverVal, playerVal);
                fflush(stdout);
                return false;
            }
        }
    }
    return true;
}
```

A.11 maketestmemory.sh

```
#!/bin/bash

echo "Starting compilation."
mkdir -p build
gcc -c -funsigned-char Connect4.c -o build/Game.o
gcc -c -funsigned-char memoryfastretrieval.c -o build/memory.o
gcc -c -funsigned-char solversinglethreaded.c -o build/solver.o
gcc -o build/connect4memtest.exe build/solver.o build/Game.o build/memory.o
echo "Compilation complete."
```

A.12 makesinglethreaded.sh

```
#!/bin/bash

echo "Starting compilation."
mkdir -p build
gcc -c -funsigned-char solver.c -o build/solver.o
gcc -c -funsigned-char Connect4.c -o build/Game.o
gcc -c -funsigned-char memoryfastretrieval.c -o build/memory.o
gcc -c -funsigned-char maindriversinglethreaded.c -o build/maindriver.o
gcc -o build/connect4singlethreaded.exe build/maindriver.o \
    build/solver.o build/Game.o build/memory.o
echo "Compilation complete."
```

A.13 makeopenmp.sh

```
#!/bin/bash
```

```
echo "Starting compilation."
mkdir -p build
gcc -c -funsigned-char -fopenmp solver.c -o build/solver.o
gcc -c -funsigned-char -fopenmp Connect4.c -o build/Game.o
gcc -c -funsigned-char -fopenmp memoryfastretrieval.c -o build/memory.o
gcc -c -funsigned-char -fopenmp maindriveropenmp.c -o build/maindriver.o
gcc -fopenmp -o build/connect4openmp.exe build/maindriver.o \
    build/solver.o build/Game.o build/memory.o
echo "Compilation complete."
```

A.14 makempi.sh

```
#!/bin/bash

echo "Starting compilation."
mkdir -p build
mpicc -c -funsigned-char solver.c -o build/solver.o
mpicc -c -funsigned-char Connect4.c -o build/Game.o
mpicc -c -funsigned-char memoryfastretrieval.c -o build/memory.o
mpicc -c -funsigned-char maindrivermpi.c -o build/maindriver.o
mpicc -o build/connect4mpi.exe build/maindriver.o \
    build/solver.o build/Game.o build/memory.o
echo "Compilation complete."
```

A.15 mpi-run.sh

```
#! /bin/bash
#SBATCH --job-name=connect4
#SBATCH --account=jyokota
#SBATCH --partition=savio3
#SBATCH --ntasks=960
#SBATCH --time=24:00:00

mpirun -n 960 ./build/connect4mpi.exe workingfolder
```

Appendix B

Example Runs

B.1 Running Memory Tests

In order to run the memory module test, ensure that a `build` folder has been created in the main directory. To configure the memory test, set the following:

- In `maketestmemory.sh`, change `memoryfastretrieval.c` to the C file containing the memory module you wish to test. This memory module should ideally have both solver and player working, though a memory module with only solver implemented can successfully run half of the provided tests.
- In `Connect4.c` (or any other game you wish to test), ensure that the three parameters are set to reasonable values. (5,5,4) is generally a good test of runtime for a single-threaded system, but uses 2 GiB of RAM when running `playerdata` validation; thus, smaller games may be necessary to avoid memory allocation errors.

Once configuration is done, run `maketestmemory.sh`. This should yield the following output:

```
$ ./maketestmemory.sh
Starting compilation.
Compilation complete.
```

This creates an executable `build/connect4memtest.exe`. The intended command-line interface for this executable is:

```
./connect4memtest.exe <filename>
```

where `<filename>` is an over-writable file location; this file is used to save and retrieve the `playerdata`. This runs the following tests:

- The game is fully solved using the `solverdata`, and a compilation of results are outputted. Errors in the `solverdata` would likely manifest here as incorrect results.

- The `solverdata` is compressed and saved to `<filename>`.
- `<filename>` is loaded back as a `playerdata`, and the results between the `playerdata` and `solverdata` are compared. Errors in the `playerdata` or saving function would likely manifest here.

The following output is the result of running this test using (5,4,4) Connect 4:

```
$ ./connect4memtest.exe a.out
Beginning main loop
Done computing, listing statistics
Loss in 0: 320734
Loss in 2: 66275
Loss in 4: 49479
Loss in 6: 30707
Loss in 8: 12877
Loss in 10: 3408
Loss in 12: 679
Loss in 14: 72
Loss in 16: 2
Tie in 0: 37080
Tie in 1: 74372
Tie in 2: 100364
Tie in 3: 105958
Tie in 4: 103555
Tie in 5: 90510
Tie in 6: 67610
Tie in 7: 51280
Tie in 8: 33775
Tie in 9: 22194
Tie in 10: 12232
Tie in 11: 7102
Tie in 12: 3472
Tie in 13: 1694
Tie in 14: 744
Tie in 15: 288
Tie in 16: 132
Tie in 17: 38
Tie in 18: 16
Tie in 19: 4
Tie in 20: 1
Win in 1: 348191
Win in 3: 58690
```

```
Win in 5: 54248
Win in 7: 32117
Win in 9: 12346
Win in 11: 3321
Win in 13: 608
Win in 15: 78
Win in 17: 2
In total, 1706255 positions found
357814 primitive positions found
Starting position has value 148
41041, fbda08
Starting output to file a.out
Compression complete. New length: 3385069 bytes
Initializing player data
Done initializing
verifying 0
verifying 1000000
verifying 2000000
verifying 3000000
verifying 4000000
verifying 5000000
verifying 6000000
verifying 7000000
verifying 8000000
verifying 9000000
verifying 10000000
verifying 11000000
verifying 12000000
verifying 13000000
verifying 14000000
verifying 15000000
verifying 16000000
Success
Done outputting
```

B.2 Running the Single-Threaded Shard Solver Locally

In order to run the single-threaded solver, ensure that a `build` folder has been created in the main directory. To configure the solver, set the following:

- In `makesinglethreaded.sh`, set the desired memory module and game.
- In `Connect4.c` (or any other game you wish to test), ensure that the three parameters are set to reasonable values. (5,5,4) is generally a reasonable solve on a single-threaded machine.
- In `maindriversinglethreaded.c`, set the shard size. A shard size of 25-28 tends to yield reasonable results.

Once configuration is done, run `makesinglethreaded.sh`. This should yield the following output:

```
$ ./makesinglethreaded.sh
Starting compilation.
Compilation complete.
```

This creates an executable `build/connect4singlethreaded.exe`. The intended command-line interface for this executable is:

```
./connect4memtest.exe <folder>
```

where `<folder>` is an initially-empty folder used to save intermediate results.

Each shard outputs a progress message on starting discovery and solving of a new shard. The following output is the result of running this test using (5,5,4) Connect 4 with a shard size of 28:

```
$ ./connect4singlethreaded.exe workingfolder/
Shard graph computed: 4 shards will be computed
Discovering shard 0/4 with shard id 0
Discovering shard 1/4 with shard id 1
Discovering shard 2/4 with shard id 2
Discovering shard 3/4 with shard id 3
Solving shard 0/4 with shard id 2
Compression complete. New length: 24282934 bytes
Solving shard 1/4 with shard id 3
Compression complete. New length: 22572215 bytes
Solving shard 2/4 with shard id 1
Compression complete. New length: 27555394 bytes
Solving shard 3/4 with shard id 0
Compression complete. New length: 37224709 bytes
Done computing
```

B.3 Running the MPI Solver

The MPI solver has similar syntax to the single-threaded solver (example in Appendix B.2), with the following key differences on the Savio cluster:

- The working folder should be placed in the `/global/scratch/users/<username>/` directory instead of the home directory; otherwise, the working folder will be memory-limited to a few GiB.
- After compiling, `mpi-run.sh` should be configured to set the desired Savio partition, node count, and working folder (per the specifications in Chapter 3.5).

A job can be run using the command `sbatch mpi-run.sh`. This produces the following output, and creates a job with the given ID (in this case, 12759105):

```
[jyokota@ln001 build]$ sbatch mpi-run.sh
Submitted batch job 12759105
```

Once a job begins, output gets piped into a corresponding `slurm-<Job ID>.out` file. As with the single-threaded solver, this provides progress info, which can be viewed continuously using the command `tail -f slurm-<Job ID>.out`.

The following is an abridged output from running (6,5,4) Connect 4 on 16 cores:

```
[jyokota@ln001 build]$ cat slurm-12705162.out
Starting
Shard graph computed: 1016 shards will be computed
Discovering shard 0/1016 with shard id 8
Discovering shard 2/1016 with shard id 16 by process 15
Discovering shard 3/1016 with shard id 24 by process 1
Discovering shard 4/1016 with shard id 9 by process 2
Discovering shard 5/1016 with shard id 40 by process 1
Discovering shard 6/1016 with shard id 56 by process 15
Discovering shard 7/1016 with shard id 32 by process 2
Discovering shard 8/1016 with shard id 48 by process 12
...
Compression complete. New length: 4501776 bytes
Compression complete. New length: 5507267 bytes
Compression complete. New length: 5977030 bytes
Solving shard 1016/1016 with shard id 8 by process 8
Compression complete. New length: 6308761 bytes
Done computing. Sending termination messages to remaining processes
Total time taken: 107.250000 seconds
```