Parallel Information Set Generation for Kriegspiel

Mark Richards Abhishek Gupta Osman Sarood

Abstract

Information Set Generation is the identification of the set of paths in an imperfect information game tree that are consistent with a player's observations. The ability to reason about the possible game history is crucial to the performance of game playing agents. In this work, we discuss the problem of information set generation in the context of kriegspiel (partially observable chess). We implement the algorithm on top of a general purpose combinatorial search engine and discuss its performance under different load balancing strategies and with different grainsize parameters. Our results suggest that our algorithm can be efficiently parallelized.

1. Introduction

In imperfect information games, players do not have access to full knowledge of the world. Examples of imperfect information include hidden cards in Poker [1] or Bridge [7], hidden tiles in Scrabble [17], or hidden pieces in Stratego or Kriegspiel (partially observable chess) [14]. The game tree nodes that are indistinguishable to a player because they differ only in the information that is hidden to the player by rule are called that player's *information set.* The ability to estimate the value of the possible states and to reason about the probability distribution over those states is crucial to playing imperfect information games well.

The term *belief state* is sometimes used interchangeably with information set to refer to a probability distribution over possible worlds. The latter term comes from the game theory community and is preferred here. A node in a game tree denotes not only the current state of the game, but also uniquely defines a path from the initial state or root node. Thus, a game tree node implicitly encodes not only the current state of the game but also the complete history of all decisions made by all players up to that point in the game, including the outcome of any chance moves such as dice rolls or card shuffling. Knowing one's information set means knowing all possible game histories.

For many specific games, solving the information set generation problem is trivial. For example, in a poker game, it is easy to see that the unseen cards held by a player's opponents may be any permutation of the cards not seen by that player (i.e., hole cards and any revealed community cards). After the betting rounds, it would not be reasonable to assume that each of these permutations of unseen cards is equally probable, as betting decisions made by the players up to that point would be affected by the quality of those players' cards. But the set of *possible* hands for all of the opponents is easy to conceive and enumerate.

In the game of kriegspiel, information set generation is much more complicated. The player knows the opponent's position with certainty when the game begins, but after the initial move there may be varying levels of uncertainty about the location of the opponent's pieces. Unlike poker, it is not possible to simply permute all of the opponent's possible pieces over all of the squares not occupied by the player's own pieces. A configuration of pieces for the opponent is only valid if it can be reached by a legal sequence of moves. In the general case, finding the nodes in an information set is a combinatorial search problem.

Information set generation is not the same problem as solving a game tree, although it is arguably a necessary subroutine for quality game play in imperfect information games. In this work, we describe an approach to generating information sets for the game of kriegspiel.

2. Background

In this section, we describe the game of kriegspiel, and the particular variation of it that we have implemented.

Kriegspiel was invented by Henry Michael Temple in 1899. Historically, the game has required three parties: two competing players and one referee. Each player sits with a chessboard that is partitioned off so as not be seen by the other. One player controls the white pieces; the other controls the black pieces. The referee has a board on which she keeps track of both sets of pieces. Each player has a full set of chess pieces. He may position the opponent's pieces however he wishes on his own board (if at all). The players alternate moves as in a regular chess game, but instead of announcing their moves out loud, they write the source and destination square of their desired plays on a piece of paper and pass them to the referee, so that the requested move is

not known to the other player. The referee, who knows the location of all pieces, checks the legality of the move. If the move is legal, the referee makes the move on his own board and announces that a legal move has been made and that it is the other player's turn to move. If the move is illegal because it is blocked by an opponent's piece or would place or leave the active player in check, the referee announces, "No." There is no penalty for attempting an illegal move. The player continues to attempt moves until one finally succeeds. If a player has no legal moves (because of stalemate or checkmate), the referee announces the game's result. With the advent of computers, it is convenient to play the game over a network, with a computer acting as the referee.

The referee makes other announcements besides declaring move illegal, and there are several variations of the game that differ only in the nature of these announcements. All declarations by the referee are heard by both players. Thus, for example, a player will hear and know if her opponent has attempted an illegal move, and will thus know that her opponent's pieces are configured in such a way as to allow at least one attemptable move that blocked or would leave him in check.

The referee announces when a player is in check. (And again, this announcement is heard by both players.) There is some variation in what additional information is supplied in this case. In one popular variant, the referee declares that the player is in check "by rank," "by file," "by diagonal," or "by knight." In the case of a diagonal attack, the referee may announce whether the checking piece is along the long or short diagonal (from the king's perspective). It is possible to be in check by two pieces at the same time, in which case the direction of both attacks would be declared. In our implementation, for simplicity, the referee declares only that the player is in check, without specifying the nature of the attack.

In the case of the capture, the referee announces the location of the piece that was captured, so that both players can remove the piece from the board. In some variations, the referee also announces whether the captured piece was a pawn or non-pawn. In our implementation, information about the captured piece is not explicitly provided to the player who makes the capture.

Because there is no penalty for making illegal moves, it is often to a player's advantage to attempt to make moves which are likely to be illegal, as hearing such declarations from the referee can provide important information to a player about the location of the opponent's pieces. In particular, since a diagonal move by a pawn is legal only the case where such a move would capture an opponent's piece, it could potentially be profitable to attempt all or many pawn captures on every turn. In order to speed the game up, popular variants of the game require the referee to make some kind of declaration at the beginning of a player's turn with respect to the pawn captures that are available. This may be as simple as declaring "Try" to indicate that at least one pawn capture is possible. In our implementation, the referee declares all potential pawn captures.

In the case of a pawn promotion, the player making the pawn advancement would secretly notify the referee as to which piece (queen, rook, bishop, or knight) he would like. To reduce the number of bits needed to specify a move, our implementation allows only promotion to queen.

In general, any move that is legal in a regular chess match may be at least attempted by a player in kriegspiel. And any kriegspiel move that is ultimately allowed by the referee would also be a legal move in chess.

We assume that a player may not attempt the same illegal move more than once per turn. This means that if a player hears multiple declarations from the referee that the opponent has attempted an illegal move, the player may assume that the opponent has at least that many possibilities available to him.

2.1. Special Moves

Our implementation supports the capability for pawns to move two spaces on their first turn. However, we did not implement support for castling or *en passant* pawn capture. (These moves require maintaining additional bits of information at each state that keep track of whether pieces have moved and/or whether pieces moved on the previous turn.)

2.2. Example

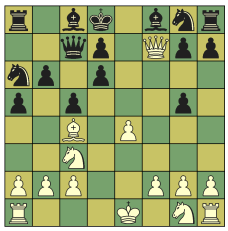

Figure 1: Actual Position

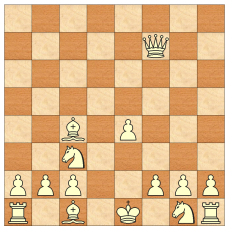

Figure 2: White's view

The concept of information sets is perhaps best explained through an example. (See also Appendix A.) Figures **??**

and **??** show a full listing and the associated pictorial progression of an actual kriegspiel game from [14]. We have adapted the notation used by Wolfe for Berkeley Kriegspiel. The moves are numbered in pairs: a move by white on one line followed by black's move on the next line. The actual move is shown first followed by a list of referee announcements. Captures announcements are prefixed with an X and give the location of the captured piece. Similarly, pawn tries announcements are prefixed with T. Attempted moves that were ultimately declared illegal are shown in a list following the ':'. For example at move 9, (Figure **??**–**??**), black attempts to block a potential threat along the diagonal by moving a pawn to f6, then attempts to escape or capture a threat at f7. When both of these attempts fail, black can infer that his king is being threatened by a protected bishop or queen at f7. The attempt to move to e7 is black's way of finding out whether the threatening piece is a queen or bishop. When this move fails, black knows it is white's queen and ultimately retreats to d8. Note that the failed moves by white on turns 3 and 4 are a result of being blocked by the pawn at e2.

The listing in Figure **??** corresponds to the referee's view of the game and would be useful for post-game analysis or commentary. During the game, players would not have access to the full information. The transcript from white's perspective is shown in Figure 3. Note that black's actual moves have been replaced with "??" and lists of black's illegal moves have been replaced with the *number* of illegal attempted moves. For example, on move 9, black attempted three illegal moves.

The listing of a sequence of moves from the perspective of one player lists all of that player's moves exactly and includes the declarations from the referee. We refer to such a list as a player's observations. Given a list of observations for a sequence of moves, an information set is the set of all possible *sequences of moves for both players*, that are consistent with those observations. In other words, any sequence of moves that would have generated the same sequence of observations for black is in black's information set. Figure 4 shows the information set for black after both player's have made five moves. From this list of possibilities, black can infer that white must have a bishop at g5 and a pawn at d6. There are 25 possible sequences of moves that are consistent with black's observations up to that point. Table 1 shows the size of the information sets for each player for each move in the game and the amount of time to find the full set on a single processor. Appendix B shows the full output of the program for the same point in the game (five moves for each player) and illustrates all of the possible positions in the belief state.

```
1. d4 {(:)}
   ?? {(:0)}
2. Bg5 {(:)}
   ?? {(:0)}
3. Nc3 {(:Bd8)}
   ??  {(:0)}
4. d5 {(:Bd8;Tc5)}
   ??  {(:0)}
5. d6 {(:)}
   ?? {(:0,Td6)}
6. e4 {(:)}
   fxg5 {(Xg5:Td6;Tg5)}
7. Bc4 {(:)}
   ?? {(Xd6:Td6)}
8. Qd5 {(:)}
   ?? {(:0)}
9. Qf7+ {(:)}
   ?? {:0}
10. Qxf8++ {(1-0:)}
```

Figure 3: Listing of a kriegspiel game resulting in checkmate by white after 10 moves.

1. Pa2:a4 Pa7:a5 2. Pd2:d4 Pb7:b6 3. Bc1:g5 Pc7:c5
4. Pd4:d5 Nb8:a6 5. Pd5:d6 Pf7:f6

1. Pd2:d3 Pa7:a5 2. Bc1:g5 Pb7:b6 3. Pd3:d4 Pc7:c5
4. Pd4:d5 Nb8:a6 5. Pd5:d6 Pf7:f6

1. Pd2:d4 Pa7:a5 2. Pa2:a4 Pb7:b6 3. Bc1:g5 Pc7:c5
4. Pd4:d5 Nb8:a6 5. Pd5:d6 Pf7:f6

Figure 4: Abbreviated output from example game. Generating information sets for black after five turns by each player. From this, black can infer that there is definitely a white bishop at g5 and a white pawn at d6.

	White		Black	
Ply	**Size**	**Time**	**Size**	**Time**
1	18	0.000	20	.000
2	18	0.000	19	.000
3	18	0.000	404	.008
4	279	0.008	401	.024
5	242	0.028	1472	.044
6	216	0.052	155	.096
7	176	0.064	293	.116
8	3406	0.092	158	.128
9	2191	0.288	118	.148
10	1423	0.508	25	.152
11	1363	0.596	798	.164
12	1416	0.744	518	.192
13	1416	0.836	13564	.304
14	3440	0.97	12394	.776
15	3428	1.25	343652	3.75
16	50521	1.82	320704	17.3
17	43192	5.91	490162	99.9
18	26128	7.94	3792	119
19	19061	9.92	14836	121

Table 1: Solution counts and running times for a sample kriegspiel game

3. The Information Set Generation Algorithm

We have formulated information set generation as a combinatorial search problem. The input to the algorithm is a list of observations for one player in a kriegspiel game. We assume that the agent has perfect recall (i.e., remembers all of its own decisions), which is a reasonable assumption for kriegspiel. (See [**?**] for a further discussion of perfect recall.)

The nodes in the search tree at depth n correspond possible to possible sequences of n plies from the start state. There is a search space variable for each ply in the game, and the values that a variable can take on are all possible legal moves. A kriegspiel game may last for on the order of tens of plies. Moves are parameterized by the piece that is relocating (both black and white have six different types of pieces: king, queen, rook, bishop, knight, pawn), the source square, and the destination square. Only a few hundred of the $64 \times 64 \times 6$ combinations represent moves that are legitimate in at least some scenarios. For example, no piece could ever move from a3 to h2[1] in one move.

Information set generation, then, is an all-solutions search through the space. Whenever a node in the search tree is found to be inconsistent with the actual observation given in the input, the subtree rooted at that node is pruned. There are types of observations that must match in order for the state to be found: check status, pawn tries, and illegal moves available/attempted.

Figure 8 shows the function at the heart of our information set generation algorithm. The actual observations can be viewed as global, read-only variables. All solutions (i.e., all nodes in the information set) will be located at the same depth in the tree: the depth that is equal to the number of plies in the observation list. Referee announcements regarding pawn tries or a player being in check are made prior to a move. The algorithm checks for the consistency of these observations in lines $5 - -6$. If the search reaches the appropriate depth and passes these tests, then a solution has been found and is reported (line 7).

Otherwise, we consider the possible moves that can be made from that state. If the player whose turn it is to move at the current level of the search tree is the same player from whose perspective we have received observations, then the next step is to make sure that every legal move that was attempted at the player at that ply is also at least attemptable at the game position that corresponds to the game tree node (lines $15 - -22$). We know exactly the legal move that was ultimately made at this depth (because we are assuming perfect recall), and we check to make sure that this move is indeed legal (pruning otherwise). If so, we apply that move and recurse (lines $24 - -32$). This corresponds to a "forced move" in the general search paradigm.

On the other hand, if the player whose turn it is to move at the current depth of the search tree is the opponent, then must consider all possible legal moves to branch on. We first check to make sure that the number of legal moves from this position is at least as large as the number of illegal moves that the opponent attempted (i.e., the number of times the referee said "No" before finally accepting the opponent's move). This is shown in lines $39 - -42$. At this point we generate a child for each move that would be accepted by the referee and recurse (lines $44 - -52$).

One of our goals was to assess the varying utilities of the different pruning criteria. We tried some different orderings of the tests and were somewhat surprised (and disappointed) to see that there was actually a fairly minimal impact on performance. This could be due to the fact that because of the tedious nature of encoding problem instances from actual games, we were not able to run experiments of a wide variety of types of positions (i.e., different styles of play).

4. Charm++

We built our search engine on top of the CHARM++ runtime system for parallel processing [8, 9]. CHARM++ is an extension to the C++ programming language that provides machine-independent infrastructure for parallel computation. The language and its accompanying runtime system

[1] We observe the convention that on an 8×8 chessboard the rows are numbered 1 to 8 and the columns are labeled a to h.

```
1: function DFS-INFOSET(a_{1:t}, o^{(k)}_{1:T}, I, s, t)
2:     if t > T then
3:         I ← I ∪ a_{1:t}
4:     else
5:         for each legal action a at s do
6:             s' = GETNEXTSTATE(s,a)
7:             if s'.o^{(k)}_{t+1} = o^{(k)}_{t+1} then
8:                 DFS-INFOSET(a_{1:t}a, o^{(k)}_{1:T}, I, s', t + 1)
```

have been ported to many shared-memory and distributed-memory platforms.

In contrast to other popular parallel programming frameworks such as MPI and OPENMP, the CHARM++ language is based on object-oriented programming principles. The programmer is responsible to decompose its problem into a collection of objects that represent elements of work to be done. Typically the number of work objects far exceeds the number of processing elements (cores) available. These objects are defined so as to be able to be migrated between processors *during execution*, in order to improve the overall load balancing.

Work objects, known as *chares*, communicate with each other via asynchronous message-passing. Objects can invoke *entry methods* on other objects, which are executed according to a schedule set by the CHARM++ runtime system. The runtime is responsible to interleave computation and communication tasks in order to maximize efficiency. To facilitate this, each processor maintains an *incoming queue* of messages targeted to chares that it currently holds. A corresponding *outgoing queue* collects messages generated by objects on the processor that need to be sent to objects assigned to other processors.

Some work tasks may inherently require more computation than others, but the amount of computation associated with each piece of work is not always known before execution. In tree search, for example, a piece of work may be defined as the computation of a minimax value on a node in the tree using alpha-beta pruning. A natural decomposition is to assign different nodes to different processors for evaluation. Because some nodes will be pruned at shallower levels of the tree than others, the amount of work performed at each subtree can vary considerably. In a traditional parallel framework, this would result in many processors idling while a small number of processors execute the larger workloads. By allowing the CHARM++ runtime system to schedule a large number of small pieces of work, better adaptive load balancing is achieved.

5. The CHARM++ Parallel State Space Search Engine

Sun *et al.* have developed the Parallel State Space Search Engine (PARSSSE) within the CHARM++ framework, which exposes an API that is useful for a variety of graph-based and tree-based search problems [21]. In PARSSE the fundamental unit of work is the processing of a node. Generally, this means identifying whether the node corresponds to a solution and/or generating descendant nodes and spawning work tasks related to them. A task is *terminal* if it spawns no children.

A *grainsize control* parameter specifies a bound on the size of a piece of work that can be passed along to the runtime system for scheduling. Once a task has been subdivided into an appropriate number of subtasks, those tasks run to completion on a single processor without preemption and without generating any additional parallel tasks. If the grainsize is too large, then the number of pieces of work to be done may be lower than the number of processors available, forcing some processors to idle. If the grainsize is too small, then the runtime system is flooded with a large number of trivial tasks, and the system gets bogged down by the overhead of creating and scheduling tasks instead of actually doing them. PARSSE supports an adaptive grainsize management, so that the threshold for sequential processing can be changed during the execution of the program, depending on the properties of the problem and/or the real-time performance analysis of the runtime system.

PARSSE currently supports two different load balancing strategies. The first is the randomized strategy. In this approach, each newly created object is assigned to a random processor. Generally, if a large enough number of chares is created, the system will achieve an approximately uniform distribution of work across processors. The approach is also appealing because of its simplicity. However, randomization can sometimes lead to inefficiencies from parallel overhead. For a program running on p processors, the probability that a piece of work is assigned to a processor other than the one that created it is $1 - 1/p$. As the number of processors gets large, this probability approaches 1, which means that the parallel overhead related to the creation and scheduling of a new task is incurred for almost every piece of new work.

A second load balancing strategy is known as *work stealing*. In this approach, all new tasks are inserted in the message queue of the processor that spawns them. Whenever a processor is idle it becomes a *thief*. A thief randomly selects another processor as its *victim* and sends a message to that processor requesting work. When the victim receives the message, it removes the work with the highest priority from its own queue and sends it to the thief. If it has no work available, it sends a negative acknowledgment message to

Procs	Time (s)	Speedup
1	1524	1
2	812	1.88
4	407	3.74
8	208	7.32
16	102	14.8
32	53.2	28.7
64	28.0	54.5
128	14.9	102
256	7.70	197
512	4.59	331
1024	2.64	576

Table 2: Speedups for a randomly selected problem instance on 1–1024 processors. The input observations were from the 10th ply of the game tree and the size of the information set in this instance is 4,495,121.

the thief, which causes the thief to look for another victim.

The advantage of the work stealing strategy is that the message-passing overhead is only incurred if at least one processor is idling. Furthermore, because victims respond by sending the highest priority work, the most important tasks are more likely to get done sooner.

In addition to these strategies, a user may define its own load balancing strategies by overriding CHARM++ task dispersal functions.

6. Performance Results

We built our search on top of the same general search engine that we used in class. The performance results reported in this section were obtained by running our program on Bleuprint with up to 1024 processors.

Table 2 shows the performance results for our algorithm on a randomly selected problem instance in which the size of the information set was about 4.5 million. (This instance is not necessarily representative of all games or states of the game, but it was not cherry-picked in any way.) The table shows that the problem is indeed highly parallizable. The speedups are nearly linear for up to 32 processors, and even on 1024 processors, we get a good speedup. In this particular instance, the sequential algorithm takes almost 25 minutes to run. But when parallelized on 1024 processors—perfectly reasonable resource usage on modern hardware—the search takes less than three seconds. This is a speedup of 576 times! It might not be reasonable to use the algorithm at all if it takes 25 or more to run for some turns. But if the cost is only a few seconds, it becomes entirely feasible.

In our next round of experiments, we set out to compare a few different load-balancing strategies for the problem. We used the random, work stealing, and neighborhood

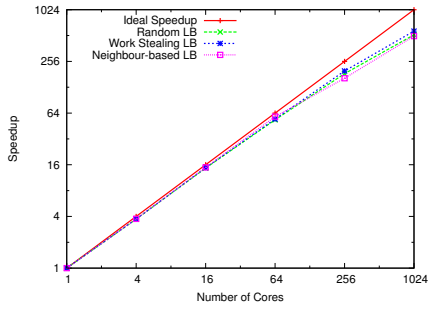

Figure 5: Experiments using different load balancing strategies.

balancing strategies that are available in the general purpose search engine. These results are shown in Figure 5. Our results show that there was actually very little variation in performance across the different strategies. We did notice that across all of our experiments, the work stealing approach appeared to be consistently—if only marginally—better than the others. Our hypothesis here is that perhaps the nodes are rarely starved for work. In other words, perhaps the very nature of the problem means that the load is naturally well-balanced and that the load-balancer rarely needs to do a significant amount of redistribution. This is probably also related to the fact that we are doing an all-solutions search.

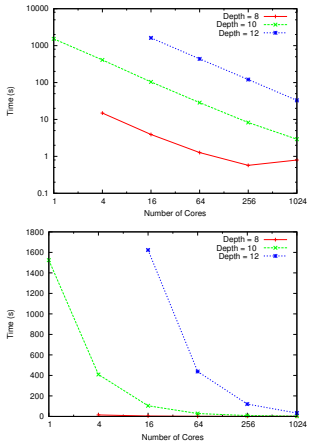

Figure 6: (Left) Log-log plot of running time for three different sizes of problem as the number of cores used increases. (Right) Same data with a linear y-axis for the running time (to help see the isoefficiency.)

We next attempted to measure the isoefficiency of the problem—how is efficiency of the parallelization of the algorithm affected by *size* of the problem? To test this we varied the number of plies in the test instance from 8 to 12. We used the work stealing load balancing strategy. The results are shown in Table 6. The data are the same in both

tables, but the one on the left uses a log-log scale. In the easiest case (depth 8), there isn't enough additional parallel work to justify the jump from 256 to 1024 processors, and performance actually degrades. Other than that, however, we see consistent speedups as we increase the number of processors. The plot on the right helps to illustrate the good isoefficiency of the problem. As the size of the problem increases, the amount of parallelizable work available also increases, and therefore the steepness of the speed curve is higher for larger problem sizes, as the number of processors increases.

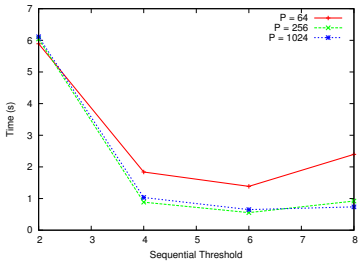

Figure 7: Grainsize control data for depth 8 problems.

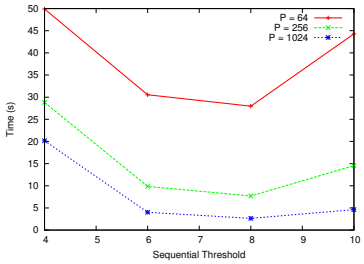

Figure 8: Grainsize control data for depth 10 problems.

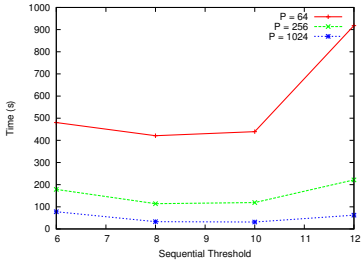

Figure 9: Grainsize control data for depth 12 problems.

Finally, we ran some experiments to measure the effect of grainsize on the problem. The results are shown in Tables 7– 9. It is clear that there is a "sweet spot" for each problem size. (At least it is clear for this particular instance. It is possible that there would be different behavior on different problem instances.) When the grainsize is too small, the overall computation is hampered by excessive parallel overhead (communication, queueing of nodes to expand, load balancing operations). On the other hand, when the grain size is too large, it becomes too difficult to ensure that all processors are busy and processors are not idling with no work to do while a small number of processors are busy doing large-sized chunks of work.

In our experiments, the optimal grainsize is clearly related to the size of the problem, the depth of the search tree. The optimal sequential threshold seems to be $2 - -4$ less than the problem depth. Since the depth of all solutions is the same for each problem instance, this means that the number of levels that should be searched sequentially is fairly consistent across problem sizes. We noted that the number of solutions found—in other words, the size of the information sets—is typically fairly large with respect to the total number of nodes expanded. (We don't have hard figures, but we estimate that it was often $30 - 40\%$). This high density of solutions limits the variability that can be present in the amount of work to be done at the nodes representing the last few levels of the tree. This means that the best grainsize is also likely to be fairly predictable.

7. New Results

In addition to the random instances, we also evaluated our information set generation routine on 40 positions from actual kriespiel games, including human vs. human and computer vs. computer. Many of these were trivial to solve in a few seconds or less on a desktop computer. The running times and speedups for two of the more challenging instances are shown in Tables 10–6. These tables show the results for runs on one to 1024 cores. Each column corresponds to a different parameter. In this case, the grainsize is the number of levels of the search tree that are searched in parallel. Up to that threshold, each node that is generated spawns a new piece of work, which the CHARM++ runtime schedules to execute on one of the processing elements according to its load balancing strategy. Above the threshold the entire subtree is searched to completion on a single processor, with no additional parallel overhead.

Note that the optimal grainsize varies for the different numbers of cores and the pattern is relatively consistent across problem instances. The optimal grainsize increases as the number of cores increases. This makes sense, because as this parameter grows, the number of distinct pieces of work increases. When the number of pieces of work per processor is high, much of the overhead of parallelization is wasted. But as the number of cores increases, the work can be efficiently broken up into smaller pieces and doled out to all the cores. With more pieces of work, it becomes easier to balance the load.

Note that both problems scale well even up to 1024 pro-

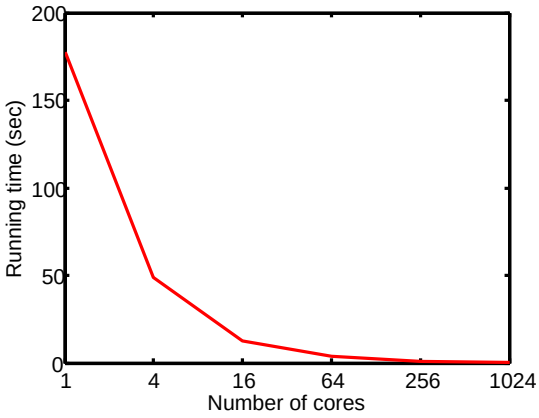

Figure 10: Kriegspiel position from Virgil-David 1969 with information set size 764,209.

PE/G	2	4	6	8
1	**177**	178	184	493
4	71	53	**48**	75
16	34	19	**12**	16
64	24	7	**3.7**	4.1
256	24	4.4	1.2	**1.0**
1024	26	3.8	0.64	**0.40**

Table 3: Running times, in seconds, to find full information set of size 764,209 at position 10 of Virgil-David 1969. Optimal grainsize (G) for each number of cores (PE) is shown in bold

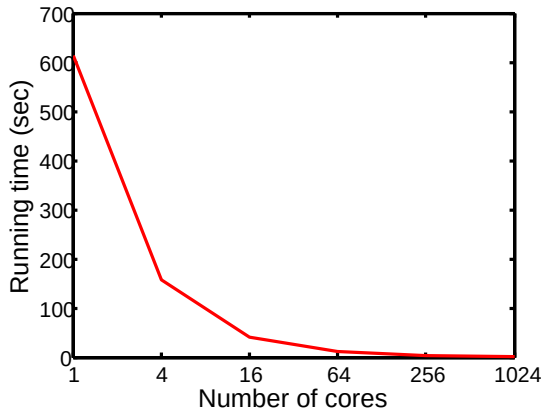

Figure 11: Kriegspiel position from Game 6 of Kbot-Darkboard at the 2006 Computer Chess championship. Information set size is 5,099,257.

PE/G	2	4	6	8
1	617	**612**	650	3312
4	213	170	**158**	367
16	83	49	**40**	64
64	54	19	**11**	14.9
256	54	9.8	3.5	**3.5**
1024	58	5.5	1.3	**1.2**

Table 4: Running times, in seconds, to find full information set of size 5,099,257 at position 10 of Game 6 Kbot-Darkboard 2006. Optimal grainsize for each number of cores is shown in bold.

PE/G	2	4	6	8
1	1	0.99	0.96	0.36
4	2.48	3.34	3.63	2.35
16	5.19	9.27	13.8	10.7
64	7.25	23.6	47.39	43.1
256	7.1	40.1	145	172
1024	6.61	46.5	276	**443**

Table 5: Speedups for Virgil-David 1969.

PE/G	2	4	6	8
1	1	1	0.95	0.19
4	2.89	3.63	3.89	1.68
16	7.41	12.4	15.1	9.57
64	11.3	32.3	52.9	41.4
256	11.3	62.9	172	174
1024	10.6	111	464	**536**

Table 6: Speedups for Kbot-Darkboard 2006.

cessors. We achieve a speedup of 443 in the game between humans and up to 536 in the game between computers. Figures 10 and 11 show the plots of the running times for the best grainsize for each core setting. These plots show that the running time continues to decrease as the number of cores increases.

Our results compare favorably to previous work in parallel game tree search, shown in Table 7. The previous best parallelization results were for parallelizing alpha-beta pruning. Ferguson and Korf achieved a speedup of 12 on 32 cores in the game of Othello [6]. In the game of chess, Felton and Otto showed a speedup of 100 on 256 cores [5]. This limited scalability is due to speculative loss. Parts of the game tree that are searched during the parallel execution would have been pruned had the algorithm run sequentially. This is a fundamental limitation in alpha-beta search.

For partially observable games, information set generation represents a different variety of tree search that does not suffer from speculative loss. All of the nodes that are visited in parallel would have also been visited in a sequential run. The key challenge for parallelizing information set generation is load balancing: some parts of the tree will be pruned at a higher level than others, resulting in a wide variation in cost for each subtree. The CHARM++/PARSSSE framework is perfectly suited to handle the load balancing issues that this problem naturally presents.

Note also that information set generation is just one subroutine needed to reason about partially observable games. Once plausible game histories have been found, an agent may simulate some sequences of moves up to previous moves made by opponents and then recursively call the information set generation procedure from the perspective of the opponent. This results in many parallel applications of information set generation. Furthermore, given a node in an informaion set, it is common for an agent to perform forward search on that node (e.g., using MCTS). With these two additional levels of parallelism on top of the information set generation problem, it is conceivable that the problem of playing partially observable games could scale efficiently to millions of processors or more.

Authors	**Game**	**Year**	**Best Speedup (cores)**
Ferguson & Korf	Othello	1988	12(32)
Felton & Otto	Chess	1988	100(256)
Richards *et al.*	Kriegspiel	2012	178(256)
Richards *et al.*	Kriegspiel	2012	**536(1024)**

Table 7: Best Speedups reported for game tree applications.

8. Related Work

An overview of the theory of imperfect information games and other basic topics in game theory can be found in [13] and [12]. A technical explanation of the concept of information sets can be found in [**?**]. Algorithms for "solving" imperfect information games—in the sense of finding a Nash equilibrium—can be found in [10] and [11]. Koller *et al.* , showed that an equilibrium can be found in time that is polynomial in the number of nodes in the game tree. And in fact, they found that it took about as much time to generate the full game tree as it did to actually solve it. Unfortunately, this is feasible only for small games. In larger games where theoretically optimal strategies cannot be computed efficiently, computer systems tend to utilize the concept of information sets or belief states in some form or another. A common strategy is to estimate the value of a move by averaging the estimated value of the associated descendant for each node in the current information set (or a random sample from the set). While this sort of "averaging over clairvoyance" *can* lead to very poor decisions, it is nevertheless a reasonable strategy in some games (e.g., Scrabble [20]).

In this work, we have argued for an information set generation approach to kriegspiel, but other authors have approached the problem differently. Parker *et al.* , consider the problem of sampling from belief states in a variety of large game trees, including kriegspiel [16]. They show that some performance gains can be achieved by only approximate sampling from the belief state. Instead of generating the full information set, or even sampling from it, they sample from possible positions for the current state (i.e., sampling from the belief state) by matching only the most recent observations. This motivation for this approach seems to be that it would not be feasible to sample from the full information set, because of the computational expense involved. And indeed, our experiments bear this out– it can indeed be expensive to produce the full information set. For the experiments shown in Table 2, generating the full information set took over 25 minutes on a single processor. However, by utilizing the Charm++ search engine on a parallel machine (1024 processors), we were able to generate the full set in just 2.9 seconds. (Note that this set had more than 4 million members.) Our results suggest that the problem is highly scalable and our approach might therefore be preferred.

Li gives an overview of kriegspiel and discusses strat-

egy from the perspective of human players [14]. He walks through the analysis of several actual games between two human players and discusses the inferences that each player can and should make at each position. As in chess, human players are able to reason about the game at an abstract level; they tend to focus on a small number of critical possibilities and to emphasize the most important pieces and squares in each position. Computer agents, lacking such sophistication, nevertheless have some advantages when it comes to brute force search strategies. Programs such Deep Blue have shown that a brute force approach to game tree search can be effective, even against humans with vastly superior abstract reasoning capabilities, provided that significant computational resources are available and wisely used [3].

Russell and Wolfe have performed some analyses on kriegspiel positions for which it possible to prove that one player has a forced win (that is, regardless of the configuration of the opponent's unseen pieces, the player has a sequence of moves that is guaranteed to produce a checkmate) [19, 22]. The authors assume knowledge of the player's belief state. They claim that under certain "aggressive" styles of play, it is uncommon for information sets to exceed 10,000 nodes in size. They throw out (i.e., do not analyze) any positions that exceed this threshold.

In our experiments, we have found positions in which the size of the information set is many tens of millions. Our goal is to develop strategies that are amenable to use in this more general setting. In other words, rather than being able to analyze positions only at the beginning or end of the game (when the size of the information set tends to be smaller), we want to be able to develop a strategy that can be used in the middle of the game as well. game).

Nance *et al.* , approach the game from the perspective of *logical filtering.* [15] The goal of logical filtering is to maintain a compact logical representation of an agent's belief state. Recall that a belief state is a probability distribution over the possible current states of the world. A belief state for kriegspiel might take the form of $(at(whitepawn, a2) \wedge atwhitepawn(b2) \ldots \wedge at(whiterook, h1)) \vee (at(whitepawn, a3) \wedge \ldots \wedge at(whiterook, h1)) \ldots$ For even moderately sizes information sets, such representations can become unwieldy in terms of storage and the cost of updating. Furthermore, reasoning with such representations often requires the use of some theorem-proving mechanism to run logical queries, such as $at(blackking, a3?)$. Nance et al. develop some strategies to maintain compact representations under sequences of actions and observations. They give a theoretical analysis of the kinds of problems that can be treated in this way. Unfortunately, kriegspiel in its full complexity is beyond the scope of their algorithm. In order to reason about the belief states in the game, they must assume knowledge of the particular kind of piece that move in each turn.

The information set generation approach that we have presented is certainly also computationally expensive. And representing a full information set (e.g., as a list of sequences of moves that encode a path from the start state to the current node) would be similarly expensive in terms of storage space. However, we have shown that our algorithm is also highly scalable. Furthermore, contrary to the approach in [15], our approach is readily adaptable to sampling algorithms. Given a sample size that matches available memory, any single sequence of moves is readily extractable without the computational expense of a theorem prover or SAT solver.

9. Future Work

First and foremost, it would be interesting to see how our information set generation algorithm would perform as a subroutine to an actual kriegspiel playing program. The utility of generating an information set is ultimately determined by how well that information improves a player's decision-making capabilities. We noted earlier that a common use of information sets is to estimate the value of a move by averaging the estimated value of the associated descendant for each node in the current information set (or a random sample from the set). This kind of strategy can be applied naively using a belief state sampling algorithm (i.e., an algorithm that samples from the distribution of possible worlds in the current state without respect to the overall history of moves). However, it has been shown that significant improvement in play can be achieved by estimating the value of future moves from the perspective of the opponent, based on the knowledge that the opponent would have had at prior positions in the move history [17]. Such a strategy would require the use of an information set generation algorithm of the kind that we have described here.[2]

Ideally, we would run our algorithm on some deeper instances from real kriegspiel games. We are not aware of a repository of such games. Unfortunately, notation for kriegspiel is not standardized and there are several minor variations in rules (i.e., with respect to the nature of referee announcements) that are unlikely to significantly to impact the computability and scalability results but which nevertheless make it difficult to write an input reader that converts the transcript of a game into a sequence of moves readable by our algorithm. An alternative would be to play some games ourselves. Currently, the manual encoding of games is not difficult but is tedious. Still, it would be nice to have some games that go on for 40 or 50 moves and have positions where the information sets are of moderate size (on the order of a few hundred thousand to a few million).

[2] This work is underway.

The information set generation algorithm that we have implemented here is the most straightforward one and is alluded to in [16] and [19]. It is analogous to the "expand-at-the-tip" strategy for the Hamiltonian Circuit problem. An alternative information set generation algorithm is described in [18] (but not implemented in parallel) and could potentially improve the search efficiency greatly by exploiting variable ordering. This would be analogous to the disconnected edge-pairs[3] heuristic for the Hamiltonian Circuit problem. Rather than search from the root node going forward, this algorithm would seek opportunities for pruning based on the propagation of logical consequences from each move both forward and backward in time. This, incidentally, is more consistent with the way human players would analyze the game. For example, if a player's proposed bishop move is rejected and the bishop is sufficiently removed from its own king, then the player may infer that one of his opponent's pieces is along that rejected path. At least one of those spaces must have been the destination of a prior move by the opponent. If a recent move by that bishop crossed the same path, then that would be hard evidence that those squares were empty at the time. And therefore, the arrival of the opponent's piece must have been between the previous (successful) bishop move and the current (unsuccessful) bishop move. To implement this algorithm, we envision the use of planning graph type data structures [2]. Planning graphs have a sequence of alternating levels of state constraints and action constraints. State constraints would be of the form $at(bishop, d4) \vee at(bishop, e6)$ or $!occupied(e4)$. Action constraints would be of the form $move(knight, a4, c5) \vee move(knight, e6)$. At a minimum, each level would require a data structure with a bit for each possible action/state. Propagating the constraints would require much of the same machinery used in STRIPS-like planning algorithms [4], and one of the many knobs that would need to be tuned would be the depth of the propagation for each constraint. (Should the consequences of a known action or non-action be propagated one move into the future and one move into the past? More? Less?) Additionally, there would be a significant and interesting tradeoff in the effectiveness of pruning and the size of the data structures that would have to be encoded in each chare. The best algorithm might be a hybrid strategy that does forward propagation from the root at the parallel level and constraint propagation at the sequential level.[4]

There are certainly several places where the efficiency of the details of each node expansion could be improved. Many of these would reduce the overall running time but would not necessarily be interesting from the standpoint of evaluating the efficiency and isoefficiency of parallelization. For example, in checking to see if a position leaves a player in check, we currently utilize existing helper functions that generate possible moves for the opponent. It would be possible to write a more specialized function that focuses only on the position of the king in question and looks only at the squares along its rank, file, diagonals, and knight-edges for threats. There are also some memory allocation issues that could be improved.

[3]This is what we referred to as "algorithm 2" in class.

[4]This work is also in progress.

10. Conclusion

We have shown that information set generation for kriegspiel can be efficiently parallelized. The opportunities for parallelism increase as the problem size increases. Our algorithm has performed well on problems with information sets with as many as 65 million nodes—much larger than any other treatment of the problem that we are aware of. Because the algorithm can be efficiently parallelized, it would be reasonable to explore using it in a full kriegspiel player. It remains to be seen whether these scalability properties hold over a wide variety of playing styles and problem instances. We suspect that fruitful research remains with respect to alternative algorithms that implement variable ordering heuristics.

References

[1] D. Billings, A. Davidson, J. Schaeffer, and D. Szafron. The challenge of poker. *Artificial Intelligence.*, 134:201–240, 2002.

[2] A. L. Blum and M. L. Furst. Fast planning through planning graph analysis. *Artificial Intelligence.*, 90:281–300, 1997.

[3] M. Campbell, A. J. H. Jr., and F. hsiung Hsu. Deep Blue. *Artificial Intelligence.*, 134:57–83, 2002.

[4] Y. Chen. *Solving Nonlinear Constrained Optimization Problems Through Constraint Partitioning*. PhD thesis, University of Illinois at Urbana-Champaign, 2005.

[5] E. W. Felten and S. W. Otto. A highly parallel chess program. In *Proceedings of the International Conference on Fifth Generation Computer Systems*, pages 1001–1009, 1988.

[6] C. Ferguson and R. E. Korf. Distributed tree search and its application to alpha-beta pruning. In *AAAI-88*, pages 128–132, 1988.

[7] M. L. Ginsberg. Partition search. In *In Proceedings of AAAI-96*, pages 228–233. AAAI Press, 1996.

[8] L. Kalé and S. Krishnan. CHARM++: A Portable Concurrent Object Oriented System Based on C++. In A. Paepcke, editor, *Proceedings of OOPSLA'93*, pages 91–108. ACM Press, September 1993.

[9] L. V. Kale and G. Zheng. Charm++ and AMPI: Adaptive Runtime Strategies via Migratable Objects. In M. Parashar, editor, *Advanced Computational Infrastructures for Parallel and Distributed Applications*, pages 265–282. Wiley-Interscience, 2009.

[10] D. Koller, N. Megiddo, and B. von Stengel. Fast algorithms for finding randomized strategies in game trees. In *Symposium on the Theory of Computation*, pages 750–759, 1994.

[11] D. Koller and A. Pfeffer. Representations and solutions for game-theoretic problems. *Artificial Intelligence*, 94(1-2):167–215, 1997.

[12] H. W. Kuhn. *Classics in Game Theory*. Princeton University Press, Princeton, NJ, 1997.

[13] H. W. Kuhn. *Lectures on the Theory of Games*. Princeton University Press, Princeton, NJ, 2003.

[14] D. H. Li. *Chess Under Uncertainty*. Premier Publishing, Bethesda, MD, 1994.

[15] M. Nance, A. Vogel, and E. Amir. Reasoning about partially observable actions. In *Proceedings of the Twenty-first Conference on Artificial Intelligence*, 2006.

[16] A. Parker, D. S. Nau, and V. S. Subrahmanian. Game-tree search with combinatorially large belief states. In L. P. Kaelbling and A. Saffiotti, editors, *IJCAI*, pages 254–259. Professional Book Center, 2005.

[17] M. Richards and E. Amir. Opponent modeling in scrabble. In *IJCAI-07: International Joint Conference on Artificial Intelligence*, pages 1482–1487, Hyderabad, India, January 2007.

[18] M. Richards and E. Amir. Information set sampling in general imperfect information positional games. In *IJCAI Workshop on General Intelligence in Game Playing Agents*, Pasadena, CA, 2009.

[19] S. Russell and J. Wolfe. Efficient belief-state and-or search, with application to kriegspiel. In *Proceedings of the 19th international joint conference on Artificial intelligence*, pages 278–285, San Francisco, CA, USA, 2005. Morgan Kaufmann Publishers Inc.

[20] B. Sheppard. World-championship-caliber Scrabble. *Artificial Intelligence.*, 134:241–245, 2002.

[21] Y. Sun, G. Zheng, P. Jetley, and L. V. Kale. An Adaptive Framework for Large-scale State Space Search. In *Proceedings of Workshop on Large-Scale Parallel Processing (LSPP) in IEEE International Parallel and Distributed Processing Symposium (IPDPS), 2011*, Anchorage, Alaska, May 2011.

[22] J. Wolfe and S. Russell. Exploiting belief state structure in graph search. In *ICAPS Workshop on Planning in Games*, 2007.