

Parallelizing Information Set Generation for Game Tree Search Applications

Mark Richards, Abhishek Gupta, Osman Sarood and Laxmikant V. Kalé
University of Illinois at Urbana-Champaign
Urbana, IL 61801, USA
(mdrichar, gupta59, sarood1, kale)@illinois.edu

Abstract—Information Set Generation (ISG) 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 critical to the performance of game-playing agents. ISG represents a class of combinatorial search problems which is computationally intensive but challenging to efficiently parallelize.

In this paper, we address the parallelization 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 using datasets from real game instances in addition to benchmarks. Further, we demonstrate the effect of load balancing strategies, problem sizes and computational granularity (grainsize parameters) on performance. We achieve speedups of over 500 on 1,024 processors, far exceeding previous scalability results for game tree search applications.

I. 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 [2], hidden tiles in Scrabble [3], or hidden pieces in Kriegspiel (partially observable chess) [4]. The game tree nodes that are indistinguishable to a player since they differ only in the information that is hidden to the player by rule compose 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 [5].

Information Set Generation (ISG) is an operation that game tree search agents can use to identify nodes in the game tree that are consistent with their observations [6]. Since the operation is computationally intensive and may be repeatedly applied by a decision-making agent, its efficiency is a key factor in an agent’s overall performance.

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. Hence, 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, 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. However, the set of *possible* hands for all of the opponents is easy to conceive and enumerate.

However, for many games (such as Kriegspiel), information set generation is much more complicated. After the first move by the opponent, a player faces 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.

In this work, we show that ISG for Kriegspiel can be efficiently parallelized. The primary contributions of this work are the following:

- We describe an approach to efficient parallelization of an application - information set generation, which is not typical of high performance computing and scientific domain, yet important because of its usefulness in the field of artificial intelligence.
- We present scalability results for our parallel implementation of an information set generation algorithm for the game of Kriegspiel using problem instances from real games. We achieve results which are unprecedented in game tree search application.
- We demonstrate the effects of load balancing strategy and grainsize on the parallel performance of our implementation.

The remainder of the paper is organized as follows. In Section II, we review necessary background material, including the rules of Kriegspiel. Next, in Section III, we describe the nature of the information set generation problem and the algorithm used. In Section IV, we describe our parallel

implementation, including the CHARM++ Parallel State Space Search Engine (ParSSSE) platform. We then provide our experimental results in Section V followed by comparison with previous results and some discussion in Section VI. Related work is discussed in Section VII. We conclude with a discussion of future work.

II. BACKGROUND

Information Set Generation is the process of identifying the nodes in an imperfect information game tree that are consistent with a player's observations [6]. Here, we present our techniques and results in the context of a particular game: Kriegspiel [4]. However, the techniques and framework presented in this paper are general and can be applied to ISG in other imperfect information games.

Kriegspiel, or partially observable chess, was invented by Henry Michael Temple in 1899. Historically, the game has required three parties: two competing players and one referee. Today, players often compete over a computer network, with the role of the referee being automated. One player sees and controls only the white pieces; the other, black. The referee tracks both sets of pieces. Instead of openly declaring their moves, players secretly communicate move requests to the referee. The referee, who knows the location of all pieces, assesses the legality of the move according to the standard rules of chess.

If the move is legal, the referee executes the move 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." The player then continues, without penalty, 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.

All declarations by the referee are made to both players. In particular, 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 is blocked or would leave him in check.

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. In our implementation, the referee announces when a player is in check, when a piece has been captured, and whenever a pawn may capture an opponent's piece. In the case of a capture, the referee announces the square in which the capture occurs but not the identity of the captured piece.

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.

III. INFORMATION SET GENERATION

The concept of information sets is perhaps best explained through an example. Figures 1, 2 and 3 show a full game

1. d4 {(:)}
a5 {(:)}
2. Bg5 {(:)}
b6 {(:)}
3. Nc3 {(:Bd8)}
c5 {(:)}
4. d5 {(Tc5:Bd8)}
Na6 {(:)}
5. d6 {(:)}
f6 {(Td6:)}
fxg5 {(Td6,Tg5,Xg5:)}
7. Bc4 {(:)}
exd6 {(Td6,Xd6:)}
8. Qd5 {(:)}
Qc7 {(:)}
9. Qf7+ {(:)}
Kd8 {:g6,Kxf7,Ke7}
10. Qxf8++ {(1-0:)}

Fig. 1: Listing of a Kriegspiel game resulting in checkmate by white after 10 moves. The final, legal move for each player is shown followed by the list of moderator declarations and all moves that were attempted but disallowed before the legal move was accepted.

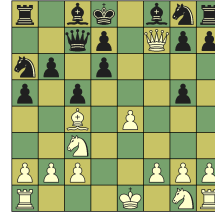


Fig. 2: Actual Position

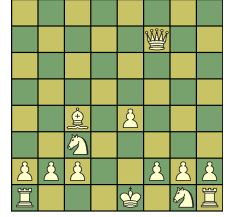


Fig. 3: White's view

listing and the associated pictorial progression of an actual Kriegspiel game [4]. We have adopted the notation used by Wolfe for Berkeley Kriegspiel [7]. Each numbered entry shows one complete turn for both White and Black. The actual move is shown first followed by a list of referee announcements. Capture announcements are prefixed with an X and give the location of the captured piece. Similarly, pawn try announcements are prefixed with T. Attempted moves that were ultimately declared illegal are shown in a list following the '·'.

Figure 2 shows the true state of the game after nine moves by each player. Figure 3 shows White's perspective of the game at that point. At move 9, Black attempts to block a potential threat along the diagonal by moving a pawn to g6, 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 e7.

```

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

```

Fig. 4: Observations received by White over the course of the game. White does not know Black’s actual moves but hears moderator announcements related to captures, pawn tries, and the number of attempted moves by Black that fail.

The listing in Figure 1 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 4. 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 shows 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 5 shows a portion of the information set for Black after both players 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.

The Information Set Generation (ISG) operation can be formulated as a combinatorial search problem. The input to the algorithm is a list of t observations for one player k (denoted $o_{1:t}^{(k)}$) in a Kriegspiel game.

The nodes in the search tree at depth n correspond to possible move sequences of n plies from the start state. (A sequence of t actions from the start state is denoted $a_{1:t}$.) There is a search space variable for each ply in the game, and

```

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

```

Fig. 5: Abbreviated output from example game. Three of 25 sequences that are consistent with Black’s observations after five moves by each player. The 25 sequences (or the states that result from them) compose Black’s information set at that point in the game. From this, Black can infer that there is definitely a white bishop at g5 and a white pawn at d6.

```

1: function DFS-INFOSET( $a_{1:t}, o_{1:T}^{(k)}, I, s, t$ )
2:   if  $t > T$  then
3:      $I \leftarrow I \cup a_{1:t}$ 
4:   else
5:     for each legal action  $a$  at  $s$  do
6:        $s' = \text{GETNEXTSTATE}(s, a)$ 
7:       if  $s'.o_{t+1}^{(k)} = o_{t+1}^{(k)}$  then
8:         DFS-INFOSET( $a_{1:t}a, o_{1:T}^{(k)}, I, s', t+1$ )

```

Fig. 6: Core of Information Set Generation Algorithm

the values that a variable can take on are all possible legal moves. 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 (i.e., search is discontinued along that path). There are three types of observations that must match in order for a state s to be included in the information set I at depth t : check status, pawn tries, and illegal moves available/attempted.

Figure 6 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 at line 7. If the search reaches the appropriate depth and passes these tests, then a solution has been found and is added to the information set (line 3).

IV. PARALLEL IMPLEMENTATION USING CHARM++ AND PARSSSE

We built our parallel ISG search engine on top of the CHARM++ run-time 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 have been ported to many shared-memory and distributed-memory platforms. CHARM++ is used by many

highly scalable scientific applications such as NAMD [10] and ChaNGa [11].

In contrast to other popular parallel programming frameworks such as MPI [12] and OPENMP [13], CHARM++ is based on object-oriented programming principles. The programmer is responsible to decompose a problem into a collection of objects that represent elements of work to be done. Typically the number of these work objects, called *chares*, far exceeds the number of processing elements (cores) available. Chares are defined so as to be able to be migrated between processors *during execution*, in order to improve the overall load balancing.

Chares communicate with each other via asynchronous message passing. They can invoke *entry methods* on other objects, which are executed according to a schedule set by the CHARM++ runtime system. The runtime is responsible for interleaving 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 chares assigned to other processors.

Some chares may inherently require more computation than others, however 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 [14], [15]. A natural decomposition is to assign different nodes to different processors for evaluation. Since 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.

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 [16]. In PARSSSE the fundamental unit of work is the processing of a tree 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 (in other words, computational granularity of a task which will be performed sequentially). 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 application performance is degraded due to the overhead of creating and scheduling tasks instead of actually doing them. PARSSSE supports 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 realtime performance analysis by the runtime system.

CHARM++ (and hence PARSSSE) provides multiple load balancing strategies and we used three of them in our application [17]. We describe these strategies next. 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 chare 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.

The second load balancing strategy is *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 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. Moreover, since victims respond by sending the highest priority work, the most important tasks are scheduled sooner.

Another strategy we evaluated is the neighbor-based load balancing which imposes a virtual topology on the processors and load exchange occur between neighbors only. In this strategy, the load balancing is initiated by overloaded processors. When a processor becomes overloaded, it sends work to its neighbors.

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

V. PERFORMANCE RESULTS

In order to understand the effect of various parameters such as grainsize and load balancing strategy, we first evaluated the scalability of our implementation using a randomly selected problem instance. Next, we show that our implementation performs equally well for instances from actual games.

A. Analysis using Random Instance

The performance results reported in this section were obtained by running our program on Blueprint with up to 1,024

processors. This system was composed of 120 POWER5+ nodes, each having 16 cores and 64GB of memory. The system was decommissioned in June 2011 and is no longer in operation.

Table I 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 sequences. 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 our implementation achieves good parallel scaling. The speedups are nearly linear for up to 32 processors, and the speedups continue to increase even up to 1,024 processors. In this particular instance, the sequential algorithm takes almost 25 minutes to run. However, when parallelized on 1,024 processors — perfectly reasonable resource usage on modern hardware — the search takes less than three seconds. This is a speedup of 576 times! In a tournament where each player is allotted an average of five minutes per move, an ISG operation that requires 25 minutes or more would be prohibitively expensive. On the other hand, if the operation can be done in under three seconds, ISG can be used as a subroutine in the decision-making process (see [18]).

1) *Effect of Load Balancing*: In our next round of experiments, we evaluated the impact of load-balancing strategies on our application’s performance. We used the random, work stealing, and neighborhood balancing strategies that are available in the general purpose search engine. These results are shown in Figure 7. Our results show that there was actually little variation in performance across the different strategies. 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 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 also related to the fact that we are doing an all-solutions search.

2) *Effect of Grainsize*: Next, we evaluate the effect of grain-size on execution time. Grainsize control is obtained in our implementation using a run-time argument called *sequential threshold*, which specifies the depth in the search tree up to which parallel tasks would be created. With this terminology, grain size = max game tree depth - sequential threshold. Figures 8(a)– 8(c) illustrate the variation in application performance with sequential threshold for three different problem sizes. 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 sequential threshold is large (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 sequential threshold is too small (grainsize is too large), it becomes too difficult to ensure that some processors are not idling with no work to do while a small number of processors are busy doing large chunks of work.

TABLE I: Speedups for a randomly selected problem instance on 1–1,024 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 sequences.

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

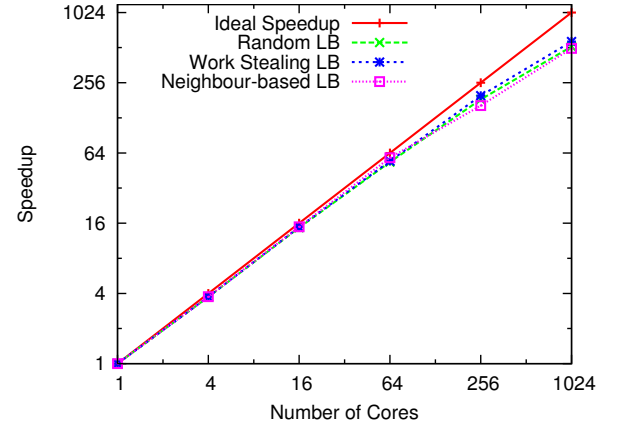


Fig. 7: Effect of load balancing strategy

In our experiments, the sequential threshold is clearly related to the size of the problem — the depth of the search tree. The optimal sequential threshold is 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 (that is, optimal grain size) 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 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, which is encouraging.

3) *Effect of Problem Size*: We next study the variation in the efficiency of the parallelization of the algorithm as size of the problem increases. Here, 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 Figure 9. In the easiest case (depth 8), there is not enough additional parallel work to justify the jump from 256 to 1,024 processors, and performance actually degrades. Aside from this anomaly, however, we see consistent speedups as we increase the number of processors. The plot on the bottom helps to illustrate the effect of problem size on parallel

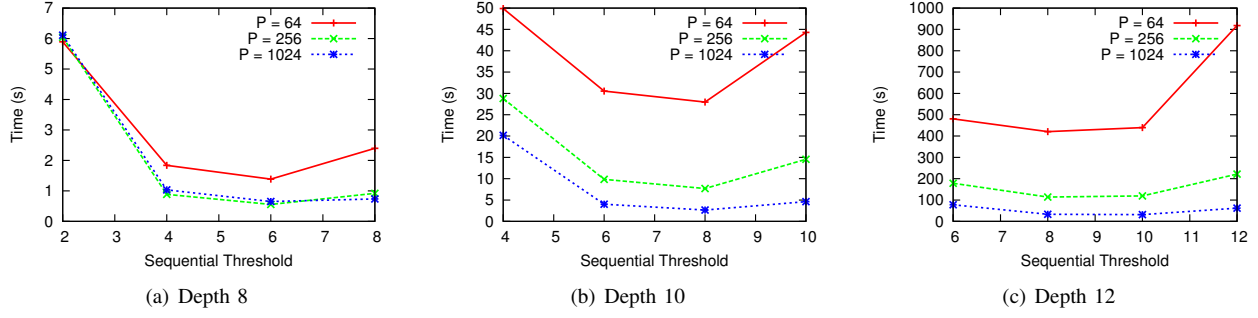


Fig. 8: Grainsize Control for different problem sizes. Sequential threshold denotes the depth in the search tree after which the remainder of the task (game tree search) is performed sequentially. Grain size = max game tree depth - sequential threshold.

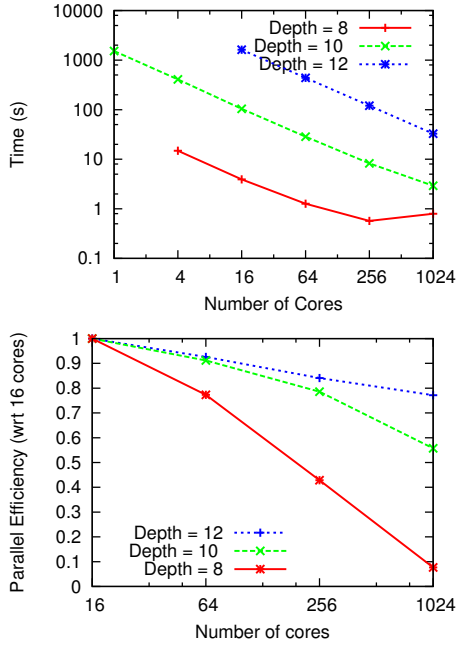


Fig. 9: (Top) Execution time for three different sizes of problem as the number of cores used increases. (Bottom) Parallel Efficiency (wrt to 16 cores) vs number of cores, to help see the effect of problem size and iso-efficiency.

efficiency (defined as Speedup/P) and helps understand the iso-efficiency of the problem. Iso-efficiency [19] quantifies how the problem size should be increased with increasing number of processors to maintain same level of parallel efficiency. As the size of the problem increases, the amount of parallelizable work available also increases, and hence parallel efficiency is higher for larger problem sizes and constant number of processors (Figure 9 bottom). As the number of processors increases, parallel efficiency decreases for constant problem size. Figure 9 suggests that in parallel information set generation, to maintain constant parallel efficiency, problem size (search depth) should be roughly increased by two when the number of processors is increased by 4 times.

TABLE II: Running times in seconds and speedups (after |), 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.

PE/G	2	4	6	8
1	177 1	178 0.99	184 0.96	493 0.36
4	71 2.48	53 3.34	48 3.63	75 2.35
16	34 5.19	19 9.27	12 13.8	16 10.7
64	24 7.25	7 23.6	3.7 47.39	4.1 43.1
256	24 7.1	4.4 40.1	1.2 145	1.0 172
1024	26 6.61	3.8 46.5	0.64 276	0.40 443

TABLE III: Running times, in seconds and speedups (after |), 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	617 1	612 1	650 0.95	3312 0.19
4	213 2.89	170 3.63	158 3.89	367 1.68
16	83 7.41	49 12.4	40 15.1	64 9.57
64	54 11.3	19 32.3	11 52.9	14.9 41.4
256	54 11.3	9.8 62.9	3.5 172	3.5 174
1024	58 10.6	5.5 111	1.3 464	1.2 536

B. Real Game Instance

We also evaluated our information set generation routine on 40 positions from actual Kriegspiel games, including human vs. human and computer vs. computer instances. Many of these were trivial to solve in a few seconds or less on a desktop computer. For these experiments, we used the Ranger supercomputer at TACC [20]. Ranger has 16-way SMP nodes containing AMD Opteron processors. The nodes are interconnected using Infiniband technology in a fully-CLOS topology providing a 1GBps point-to-point bandwidth. We used CHARM++'s net-ibverbs machine layer build and -O3 optimization for these experiments.

The running times and speedups for two of the more challenging instances are shown in Tables II–III. These tables show the results for runs on one to 1,024 cores. Each column corresponds to a different sequential threshold parameter - 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. Beyond the threshold the entire subtree is searched to completion on a single processor, with no additional parallel overhead.

These results confirm the observations made in Section V-A2. The optimal sequential threshold varies for the different numbers of cores and the pattern is consistent across problem instances. The optimal sequential threshold increases as the number of cores increases. This is expected, because as this parameter grows, the number of distinct pieces of work increases. When the number of chares per processor is high, much of the overhead of parallelization is wasted. However, as the number of cores increases, the work can be efficiently broken up into smaller pieces and spread 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 1,024 processors. We achieve a speedup of 443 in the game between humans and up to 536 in the game between computers. Figure 10 shows the plot of the running times on these two problems with the best grainsize for each core setting. These plots show that the running time continues to decrease as the number of cores increases.

VI. COMPARISON WITH PREVIOUS RESULTS, DISCUSSION

Our results compare favorably to previous work in parallel game tree search, shown in Table IV. 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 [21]. In the game of chess, Felton and Otto showed a speedup of 100 on 256 cores [22]. 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 also would have 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 ISG procedure from the perspective of the opponent. This results in many parallel applications of ISG. Furthermore, given a node in an information set, it is common for an agent to perform forward search on that node (e.g., using MCTS). All of these forward search operation can be performed independently in parallel. With these two additional levels of parallelism on top of the information set generation problem, it is likely that

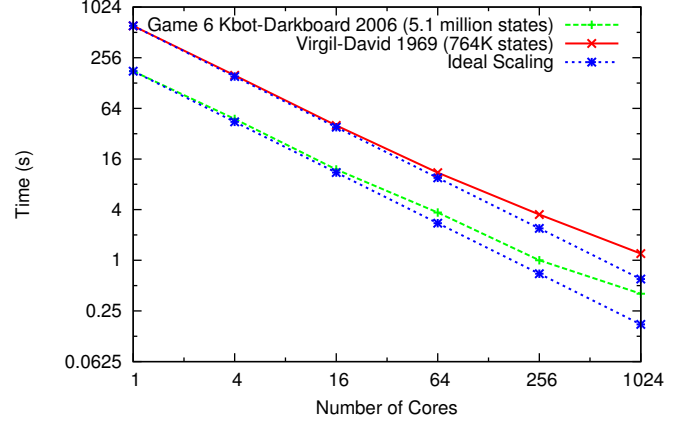


Fig. 10: Performance results for Information Set Generation for two games of Kriegspiel, position ten.

the problem of playing partially observable games could scale efficiently to millions of processors or more.

TABLE IV: Best Speedups reported for game tree applications.

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

VII. RELATED WORK

An overview of the theory of imperfect information games and other basic topics in game theory can be found in [23] and [24]. A technical explanation of the concept of information sets can be found in [25]. Koller *et al.*, showed that a Nash equilibrium can be found in time that is polynomial in the number of nodes in the game tree [26]. 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, such as Scrabble [27].

In this work, we have argued for an ISG 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 [18]. 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. The 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. Our experiments bear this out— it can indeed be expensive to produce the full information set. For the experiments shown in Table I, generating the full information set took over 25 minutes on a single processor. However, by utilizing the CHARM++/PARSSSE search engine on a parallel machine (1,024 processors), we were able to generate the full set in just 2.9 seconds.

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 [28], 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.

The effect of load balancing and computational granularity on combinatorial search problems has been studied before [16], [29]. In this work, we addressed the parallelization of an information set generation algorithm and studied the impact of these factors applied to game tree applications.

VIII. CONCLUSION AND FUTURE WORK

In this paper, we showed that information set generation for Kriegspiel can be efficiently parallelized. Through extensive experimentation, we demonstrated that the choice of a suitable load balancing strategy and optimal grain size control is crucial to achieve good scalability for parallel ISG. 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.

Since 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.

ACKNOWLEDGMENTS

Authors were supported by grants NSF-0904844, DOE DE-SC0001845. We used machine resources on the Ranger cluster (TACC), under TeraGrid allocation grant TG-ASC050039N supported by NSF. First and second author contributed equally.

REFERENCES

- [1] D. Billings, A. Davidson, J. Schaeffer, and D. Szafron, “The challenge of poker,” *Artificial Intelligence*, vol. 134, pp. 201–240, 2002.
- [2] M. L. Ginsberg, “Partition search,” in *Proceedings of AAAI-96*. AAAI Press, 1996, pp. 228–233.
- [3] M. Richards and E. Amir, “Opponent modeling in scrabble,” in *IJCAI-07: International Joint Conference on Artificial Intelligence*, Hyderabad, India, January 2007, pp. 1482–1487.
- [4] D. H. Li, *Chess Under Uncertainty*. Bethesda, MD: Premier Publishing, 1994.
- [5] M. Schofield, T. Cerexhe, and M. Thielscher, “HyperPlay: A solution to general game playing with imperfect information,” in *Proceedings of the AAAI Conference on Artificial Intelligence*. Toronto: AAAI Press, Jul. 2012.
- [6] M. Richards and E. Amir, “Information set generation in partially observable games,” in *AAAI*, Toronto, ON, 2012.
- [7] J. Wolfe and S. Russell, “Exploiting belief state structure in graph search,” in *ICAPS Workshop on Planning in Games*, 2007.
- [8] L. Kalé and S. Krishnan, “CHARM++: A Portable Concurrent Object Oriented System Based on C++,” in *Proceedings of OOPSLA’93*, A. Paepcke, Ed. ACM Press, September 1993, pp. 91–108.
- [9] L. V. Kale and G. Zheng, “Charm++ and AMPI: Adaptive Runtime Strategies via Migratable Objects,” in *Advanced Computational Infrastructures for Parallel and Distributed Applications*, M. Parashar, Ed. Wiley-Interscience, 2009, pp. 265–282.
- [10] A. Bhatel et al., “Overcoming Scaling Challenges in Biomolecular Simulations across Multiple Platforms,” in *IPDPS 2008*.
- [11] P. Jetley, F. Gioachin, C. Mendes, L. V. Kale, and T. R. Quinn, “Massively parallel cosmological simulations with ChaNGa,” in *Proceedings of IEEE International Parallel and Distributed Processing Symposium 2008*, 2008, pp. 1–12.
- [12] “MPI: A Message Passing Interface Standard,” in *M. P. I. Forum*, 1994.
- [13] L. Dagum and R. Menon, “OpenMP: An Industry-Standard API for Shared-Memory Programming,” *IEEE Computational Science & Engineering*, vol. 5, no. 1, January-March 1998.
- [14] D. E. Knuth and R. W. Moore, “An analysis of alpha-beta pruning,” *Artificial Intelligence*, vol. 6, no. 4, pp. 293–326, 1975.
- [15] G. M. Baudet, “An analysis of the full alpha-beta pruning algorithm,” in *STOC ’78: Proceedings of the tenth annual ACM symposium on Theory of computing*. New York, NY, USA: ACM, 1978, pp. 296–313.
- [16] 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.
- [17] G. Zheng, “Achieving high performance on extremely large parallel machines: performance prediction and load balancing,” Ph.D. dissertation, Department of Computer Science, University of Illinois at Urbana-Champaign, 2005.
- [18] A. Parker, D. S. Nau, and V. S. Subrahmanian, “Game-tree search with combinatorially large belief states,” in *IJCAI*, L. P. Kaelbling and A. Saffiotti, Eds. Professional Book Center, 2005, pp. 254–259.
- [19] A. Grama, A. Gupta, and V. Kumar, “Isoefficiency: Measuring the Scalability of Parallel Algorithms and Architectures,” *IEEE Parallel & Distributed Technology*, vol. 1, no. 3, August 1993.
- [20] “Ranger User Guide,” <http://services.tacc.utexas.edu/index.php/ranger-user-guide>.
- [21] C. Ferguson and R. E. Korf, “Distributed tree search and its application to alpha-beta pruning,” in *AAAI-88*, 1988, pp. 128–132.
- [22] E. W. Felten and S. W. Otto, “A highly parallel chess program,” in *Proceedings of the International Conference on Fifth Generation Computer Systems*, 1988, pp. 1001–1009.
- [23] H. W. Kuhn, *Lectures on the Theory of Games*. Princeton, NJ: Princeton University Press, 2003.
- [24] H. Kuhn, *Classics in Game Theory*. Princeton, NJ: Princeton University Press, 1997.
- [25] A. Gilpin, “Algorithms for abstracting and solving imperfect information games (proposal),” Ph.D. dissertation, Carnegie Mellon University, Pittsburgh, PA, 2009.
- [26] D. Koller, N. Megiddo, and B. von Stengel, “Fast algorithms for finding randomized strategies in game trees,” in *Symposium on the Theory of Computation*, 1994, pp. 750–759.
- [27] B. Sheppard, “World-championship-caliber Scrabble,” *Artificial Intelligence*, vol. 134, pp. 241–245, 2002.
- [28] M. Nance, A. Vogel, and E. Amir, “Reasoning about partially observable actions,” in *Proceedings of the Twenty-first Conference on Artificial Intelligence*, 2006.
- [29] E. R. Rodrigues, P. O. A. Navaux, J. Panetta, A. Fazenda, C. L. Mendes, and L. V. Kale, “A Comparative Analysis of Load Balancing Algorithms Applied to a Weather Forecast Model,” in *Proceedings of 22nd International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, Itaipava, Brazil, 2010.