Multithreaded Pruned Tree Search in Distributed Systems

Yaoqing Gao and T. A. Marsland [*]
Computing Science Department
University of Alberta
Edmonton, Canada T6G 2H1
<gaoyq,tony>@cs.ualberta.ca

Abstract

Although efficient support for data-parallel applications is relatively well established, it remains open how well to support irregular and dynamic problems where there are no regular data structures and communication patterns. Tree search is central to solving a variety of problems in artificial intelligence and an important subset of the irregular applications where tasks are frequently created and terminated. In this paper, we introduce the design of a multithreaded distributed runtime system. Efficiency and ease of parallel programming are the two primary goals. In our system, multithreading is used to specify the asynchronous behavior in parallel game tree search, and dynamic load balancing is employed for efficient performance.

Keywords: Multithreaded computation, irregular problem, $\alpha - \beta$ search, data dependency and dynamic scheduling

1 Introduction

Workstation networks are increasingly prevalent because of their versatility and the advantage of high scalability and cost-effectiveness over parallel machines. Many distributed programming systems such as PVM, P4, MPI, Linda and Express have been built. Most of them handle only a process-based message-passing paradigm that ably supports data-parallel applications with regular data structures and communication patterns. But, unfortunately, a large proportion of real-word applications are irregular. Pruned search, an

[*]The research was supported by the Natural Sciences and Engineering Research Council of Canada.

important subset of these irregular applications, is central to solving a variety of problems in artificial intelligence. Compared with traditional data-parallel applications, search problems tend to be highly irregular and dynamic in nature and pose new challenges: (1) asynchronous behavior results in greater software complexity; (2) variable data structures such as lists are used; (3) communication patterns are irregular; (4) scheduling for load balancing and data locality are potentially difficult since work is dynamically created and terminated. Load distribution can not rely wholly on static analysis at compile-time. Thus dynamic scheduling at runtime plays an even more important role in enhancing performance. In the recent years, many parallel search algorithms have evolved, including *mandatory work first, principal-variation splitting, tree splitting, young brother waiting, delay splitting* and *multiple principal-variation splitting*. Implementing these algorithms is tedious and error-prone on existing process-based message passing systems. These reasons have motivated us to address the following two fundamental issues: (1) what kind of programming paradigms can be used to express search problems easily? (2) what mechanisms and functionalities must a runtime system provide in order to support parallel search efficiently? In this paper, we will use the multithreading technique to obtain both ease of parallel programming and efficient performance. Multiple threads can be used not only as a mechanism for tolerating unpredictable communication latency, but also for facilitating dynamic scheduling and load balancing. Multithreading is popular in single-process and shared-memory multiprocessor system. On distributed systems, however, it encounters other difficulties, due to the underlying physical distributed memories, and the need to address new issues such as thread scheduling, migration and communication in a separate addressing space.

This paper will introduce the design and implementation of a multithreaded distributed runtime system on a cluster of workstations. Section 2 first gives a brief introduction of typical parallel search algorithms and problems with their implementations on most existing systems. Then, after addressing some major issues of multithreading, we provide the details of how to obtain programmability and efficiency by multithreading. Finally we draw some conclusions by comparing what we have done to some related work.

2 Problems with Parallel Game Tree Search

Two person games between human and computer is one search problem that relies heavily on the $\alpha - \beta$ pruning technique to reduce search cost, by ignoring subtrees that cannot affect the final value of the root node. To speed up such search processes, many parallel algorithms have been proposed, and they can be classified into the following categories:

1. window partitioning (also called parallel aspiration search): the search window is divided into a number of disjoint segments. Different processors search the same tree with different disjoint windows.

2. tree splitting: selected nodes in a tree are split and assigned to different processors.

3. special hardware is used to generate moves and evaluate nodes in parallel.

This paper will focus on the tree splitting approach which is more popular and successful, compared with the others. The performance of parallel search algorithms is usually measured by search overhead, synchronization overhead and communication overhead [14]. These reflect the major obstacles to achieving perfect efficiency: starvation loss, interference loss and speculative loss. Another method can be used to quantify parallel performance by the critical path length and the total work [12]. By representing a parallel algorithm as an acyclic directed graph, the combination of the critical path length and the total work can be used to measure the average available parallelism of a program, and to determine the overhead induced by communications, load-balancing and scheduling.

Akl, Barnard and Doran [1] introduced the "*Mandatory Work First (MWF)*" algorithm. The fundamental idea of MWF is that the tree to be searched is perfectly ordered, and those nodes that must be visited (called *mandatory nodes*) in the sequential $\alpha - \beta$ algorithm are first search in parallel. By first searching the mandatory nodes, the algorithm attempts to achieve all the cutoffs that are possible in the serial case. Even simpler is a *root-tree splitting* algorithm. The nodes in the games are split and mapped onto a tree of processors. Marsland and Popowich's *Principal Variation Splitting (PVSplitting)* algorithm [14] improves things by splitting at all nodes in the first continuation. It concentrates on searching the first path along the principal variation with a full window and after a tighter bound α from the leftmost subtree is obtained, the remaining branches are split among the available processors and are searched with a minimal (or null) window. Later both Schaeffer and Hyatt et al. present *Distributed Search (DPVS)* [18] and *Enhanced Principal Variation Splitting (EPVS)* [11], respectively. Both of these algorithms improve the *PVSplitting* method by introducing dynamic work assignments. To parallelize game tree search on massively parallel machines efficiently, Feldmann et al. [6] introduce the *Young Brother Wait* (YBW) concept and a distributed control method. The search of a successor of a node in the game tree must be delayed until its leftmost brother is completely evaluated. Marsland et al.'s *Dynamic, Multiple Principal Variation Splitting (DM-PVSplit)* [13] employs a new approach to reduce re-search delay and synchronization overhead. It first exploits speculative parallelism at a set of most promising nodes, called *the PV set* in the expectation that the best successor almost always falls into this set. As before, alternatives to the PV set are assumed inferior until proven otherwise in parallel.

In addition, there are many other parallel algorithms such as ER [19], Jamboree [12], $\alpha\beta^*$[5] and ABDADA[21], which vary in the approach to reducing search and synchronization overhead. All could benefit from better efficiency in the underlying operating system.

Although search and knowledge lie in different dimensions, they are not independent: on one hand, deeper search in general results in more accurate evaluation, on the other, better search strategy depends on the quality of the evaluation function. All these in combination with the underlying system induce non-deterministic factors that effect the efficiency of a parallel algorithm. Thus, to develop a parallel game-playing program, the algorithm is often chosen through experience. But implementing these algorithms seems to be tedious and error-prone on existing process-based message passing systems. Most existing systems such as PVM and MPI focus primarily on process-based message-passing, but do not provide much support for scheduling beyond a basic and static scheduler. Since the dynamic nature of game tree search does not fit well into the data-parallel approach, developing parallel game tree search on these traditional systems leaves the following burdens to the programmer: (1) it is very difficult to specify the asynchronous behavior in game tree search, due to irregular data structures and communication modes; (2) work load distribution among these processes should be taken care of by the programmer. Since the creation of processes is time-consuming in the process-based message-passing model, a fixed number of processes are usually first created and then work loads are explicitly distributed by users.

To speed up the development of parallel game programs, while obtaining efficient performance, the programming system must provide high-level abstraction to separate a parallel algorithm from its scheduling mechanism, i.e., at the application level, users specify the parallelization points based on a parallel algorithm and leave the details of communication, synchronization and scheduling to the runtime system. Our approach is to use threads to express pieces of work corresponding to the nodes in game trees, a parallel algorithm to specify the dependency between threads and the runtime system to handle thread creation, scheduling, synchronization and scheduling. Due to the lower context switch and creation cost of threads, multithreading can be used to provide a finer granularity of concurrency and to facilitate dynamic load balancing. Moreover, better performance can be obtained through overlapping computation and communication to tolerate communication latency.

3 An Efficient Distributed System for Game Tree Search

3.1 Issues in implementing multithreading

Designing and implementing multiple threads in distributed systems involves such issues as thread implementation, thread communication and scheduling mechanisms, and thread-safety.

Threads can be supported at the system-level or by user-level library code. The User-level implementations multiplex a potentially large number of user-defined threads on top of a single kernel-implemented process. This approach can be more efficient than relying on operating system kernel support, because of the overhead of entering and leaving the kernel at each call, and since the context switch overhead of kernel threads is high. Furthermore, user-level threads are not only flexible but can also be customized to the needs of the language or user without kernel modification. So our system will employ this approach.

In distributed multithreaded systems, scheduling is done both within a process and among processes. The former is similar to the priority-based preemptive or round robin strategies in traditional single-processor systems, while the latter is more complicated. Typically there are two classes of dynamic scheduling strategies: work sharing (sender-initiated) and work stealing (receiver-initiated). With a work sharing strategy, whenever a processor generates new threads, the scheduler decides whether to migrate some of them to other processors on the basis of the current system state. Control over the maintenance of the information may be centralized in a single processor, or distributed among the processors. In the work stealing strategy, whenever processors become idle or underutilized, they attempt to steal threads from more busy processors. Our system aims at providing a flexible scheduling strategy thus enabling users to guide thread scheduling at runtime.

The thread safety problem is about whether multiple threads in the system can execute successfully without data corruption and without interfering with each other (see [15] for a more detailed discussion).

3.2 Multithreaded runtime system

In two-person, perfect-information games, there are two players that make moves alternately and both players have complete information about the current status. A computer game program contains at least move generation, search, and evaluation functions, plus data bases for storing opening and endgame knowledge.

For a game tree, nodes represent legal game states (boards); the root of the tree is the current board and a branch is a legal move. The aim of the search is to find a path from the root to the highest valued “leaf node” that can be reached, under the assumption of best play by both sides. The parallelism in game trees derives from dynamic tree splitting. In our multithreaded system, at the application level, a node can be expressed by a thread and a parallel algorithm specifies where threads can be activated in parallel and the dependency among threads. For example, for the *PVSplitting* algorithm, each node in the game is expressed by a thread. A thread will spawn one child thread for each legal move. But data dependency specified by the algorithm exists among these threads: After getting a tighter bound from the thread corresponding to the PV node, the remaining threads are ready to run, as shown in Figure 1.

Runtime support for multithreading: In a serial implementation of a recursive game tree search program, an implicit stack of activation frames is used to maintain the execution state for the depth-first search. But a parallel implementation needs a tree of activation frames so that any ready node can be expanded concurrently. Our system supports three

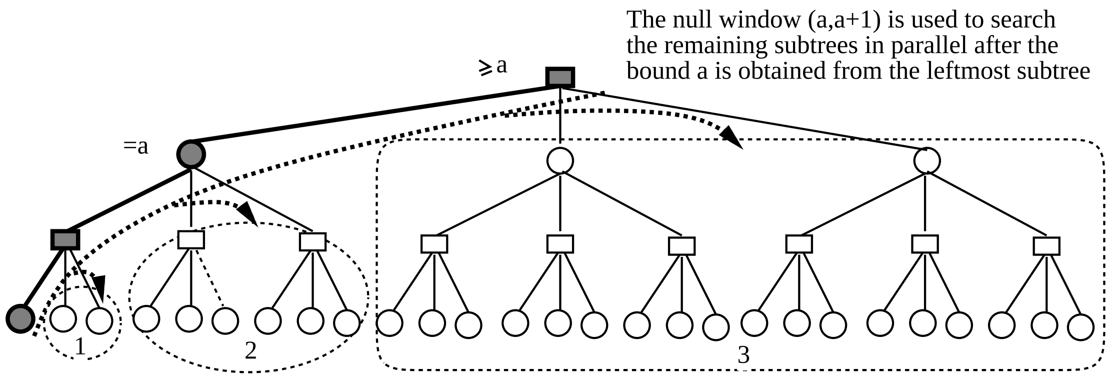
The null window (a,a+1) is used to search the remaining subtrees in parallel after the bound a is obtained from the leftmost subtree
≥a
=a
1
2
3

Figure 1: PVSplitting

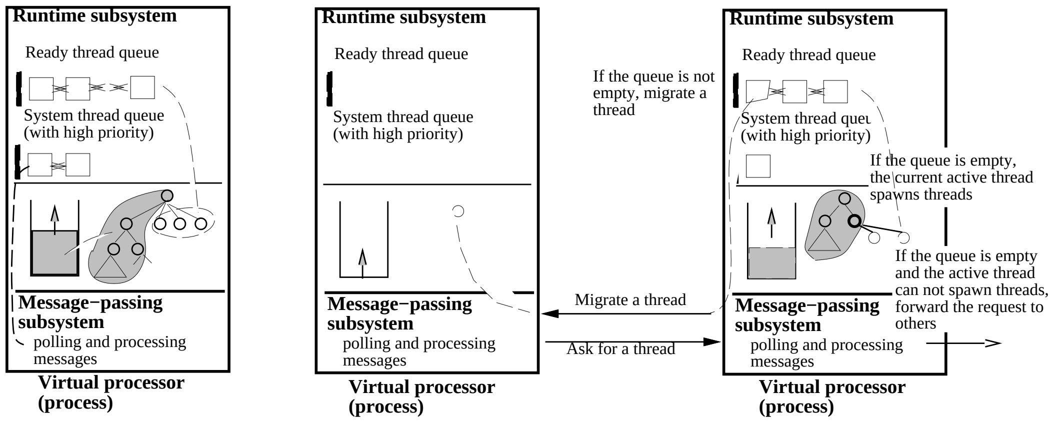
Runtime subsystem
Ready thread queue
System thread queue (with high priority)
Message−passing subsystem
polling and processing messages
Virtual processor (process)
If the queue is not empty, migrate a thread
Migrate a thread
Ask for a thread
If the queue is empty, the current active thread spawns threads
If the queue is empty and the active thread can not spawn threads, forward the request to others

Figure 2: Thread scheduling among processors

mechanisms for managing activation frames: eager, anticipatory, and lazy thread creation. By the first mechanism, an active thread will spawn all its child threads and put them into the queue. Each thread becomes ready only after all its arguments are available. This mechanism simplifies the activation frame management but has extra overhead in comparison with the latter two implementations.

To optimize accesses of activation frames, and to execute a parallel search as fast as possible compared to a serial search in a single processor, anticipatory, and lazy thread creation mechanisms are supported to create threads on demand. In each processor, a subtree in the search tree is mapped to a thread (instead of a node being a thread), which is similar to that used by Gao et al. [10]. Each processor conducts a depth-first search. Each processor maintains a ready thread queue that can be assigned to other idle processors, and a stack for activation frames during traversal down and up the game tree. Some threads must be activated on the user's pre-specified processors. Threads are prioritized and a ready thread with a highest priority will be activated first. The scheduler provides the dynamic scheduling mechanism: when the ready thread queue becomes empty, the processor can help other busy ones by sending a request message, see Algorithm 1. The processor which receives a request will first look up its ready thread queue and migrate a number of threads from the queue to the requesting processor if the queue is not empty. Otherwise it will let threads spawn their children and migrate some newly generated threads to the requesting processor. As a thread travels up and down a subtree, the anticipatory thread creation mechanism spawns ready threads once the length of the queue is less than a pre-specified threshold. But by the lazy thread creation mechanism, the currently active thread spawns its ready threads only if there is a remote request and the queue is empty. If the queue is empty and no more child threads can be spawned, the processor forwards this request to others. The child threads can be spawned at the divisible node either closest to the root of the subtree (from the bottom of the stack), or from the currently examined node (from the top of the stack), see Algorithm 2. The former may spawn more work, while the latter can result in a simple implementation and low spawning overhead, see Figure 2.

Algorithm 1:

```
 1  if the local ready thread queue is not empty
 2    then get a thread from the local ready thread queue
 3         execute the thread which corresponds to a subtree:
 4         recursively examine the subtree in a depth-first fashion
 5         if the current node is a leaf
 6           then evaluate it and backup this value
 7           else if the current node is divisible
 8                  then expand the current node in parallel only if
 9                       the local thread queue is empty:
10                       if the local ready thread queue is empty
11                         then spawn its child threads and put them into the queue
12                              the leftmost child node becomes the current node;
13                              goto 4;
14                         else sequentially expand the current node
15                              the leftmost child node becomes the current node;
16                              goto 4;
17                       fi
18                  else recursively examine the subtree in depth-first fashion:
19                       sequentially expand the current node;
20                       the leftmost child node becomes the current node;
```

```
                  goto 4;
              fi
          fi
  else
          the processor asks for a thread (thread scheduling):
          chose a processor $P_i$ based on the pre-specified order;
          send a request message to $P_i$;
fi
```

Algorithm 2:

```
if the local ready thread queue is not empty
  then get a selection of threads from the queue
          and migrate them to the request processor;
  else
          if the node currently examined is divisible
            then spawn its child threads and put them into the queue;
                  migrate one subset of threads to the request processor;
            else recursively examine the tree node in the depth-first fashion:
                  if no more threads can be spawned
                    then  chose a processor $P_i$ based on the pre-specified order;
                          send a request message to $P_i$;
                    else
                          if the current thread is divisible
                            then spawn its child threads and put them into the queue;
                                  goto 1;
                            else  goto 8;
                          fi
                  fi
          fi
fi
```

4 Related Work and Conclusion

Parallel game tree search on existing process-based message-passing systems such as PVM and MPI pose difficult programming: users need to handle scheduling issues explicitly. Parallel programming in the shared-memory paradigm is relatively easy but efficient performance may be obtained only on shared memory or massively parallel machines. Ciancarini [4] compared parallel game tree search algorithms on Linda, which defines a logically shared memory mechanism called tuples. Linda requires large volumes of data to be exchanged to and from the shared memory. This may cause cause heavy congestion over the available communication channels of a cluster of workstations and has high communication overhead.

Dynamic task migration can be broadly categorized as either at the user-level or the system-level. As discussed in Section 3.1, user-level threads and task migration simplify the implementation because there is no need to modify standard operating system kernels. Condor [17], MPVM [3] and YALB [16] are examples of such an approach. They allow parallel computations to co-exist with other applications, by global scheduling and resource management. But these systems have different goals from ours. Our system supports dynamic load balancing within a single application. TPVM [7] and Nexus [9] are two another multithreaded distributed systems but lack dynamic load balancing mechanisms.

Our system is more similar to Cilk [2], Dynamo [20] and DIB [8]. Cilk is a C-based multithreaded system and employs a work stealing strategy for load balancing. Cilk uses explicit representation throughout the computation, each node is mapped into a thread. The continuation-passing style and eager thread creation mechanism result in a simple implementation but need more frequent copying of arguments. Dynamo provides an application programming interface which separates the application and the scheduler. But, like Cilk, it only supports eager thread creation and the way tasks are spawned is as specified explicitly by users. DIB employs two representations for its activation frames to optimize accesses within a single processor: local processor stack and inter-processor tree. In addition, DIB supports a natural mechanism for fault tolerance. Our system also uses explicit and implicit activation frame representations but has different strategies for converting these two representations, including eager, anticipatory and lazy thread creation. Furthermore, users can specify different priorities of nodes and assign nodes to specific processors to guide scheduling at runtime.

This paper introduces the design and implementation of a multithreaded distributed system for parallel game search. It provides a relatively high-level and machine independent programming system that greatly simplifies the efficient parallelization of irregular applications on a cluster of workstations. Future work includes integration of multiple programming paradigms and coordination between thread and process scheduling. Multithreaded systems have demonstrated their suitability for task-parallel computations, but most existing systems have not yet obtained good efficiency for data-parallel problems. A multithreaded system involves both thread and process scheduling. So far not enough attention has been paid to coordinating these two activities. In addition, fast communication mechanisms are used to replace the heavyweight protocols of existing communication networks. Improvements in communication performance can dramatically increase the system applicability.

References

[1] Selim G. Akl, David T. Barnard, and Ralph J. Doran. Design, analysis, and implementation of a parallel tree search algorithm. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, PAMI-4(2):192–203, March 1982.

[2] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. Cilk: An efficient multithreaded runtime system. In *Proc. 6th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming*, pages 207–216, July 1995.

[3] Jeremy Casas, Dan L. Clark, Ravi Konuru, Steve W. Otto, Robert Prouty, and Jonathan Walpole. MPVM: A migration transparent version of PVM. *Computing Systems*, 8(2):171–216, 1995.

[4] P. Ciancarini. Distributed Searches: a Basis for Comparison. *Journal of the International Computer Chess Association*, 17(4):194–206, 1994.

[5] V. David. *Algorithmique parallέle sur les arbres de décision et raisonnement en temps contraint. Etude et application au Minimax*. PhD thesis, ENSAD, November 1993.

[6] Rainer Feldmann. *Game tree search on massively parallel systems*. PhD thesis, Dept. of Math. and Computer Sci., University of Paderborn, Paderborn, Germany, 1993.

[7] Adam Ferrari and V. S. Sunderam. TPVM: Distributed concurrent computing with lightweight processes. In *IEEE High Performance Distributed Computing*, pages 211–218, August 1995.

[8] Raphael Finkel and Udi Manber. DIB – a distributed implementation of backtracking. *ACM Transactions on Programming Languages and Systems*, 9(2):235–256, April 1987.

[9] Ian Foster, Carl Kesselman, and Steven Tuecke. Nexus: Runtime support for task-parallel programming languages. Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, IL, August 1994.

[10] Yaoqing Gao, Wang Dingxing, Zheng Weimin, and Shen Meiming. Intelligent scheduling AND- and OR- parallelism in the parallel logic programming system rap/lop-pim. In *Proc. of the 20th Annual Inter. conf. on Parallel Processing*, pages II–186 – II–192, August 1991.

[11] R. M. Hyatt and B. W. Suter. A parallel alpha/beta tree searching algorithm. *Parallel Computing*, 10:299–308, 1989.

[12] Bradley C. Kuszmaul. *Synchronized MIMD Computing*. PhD thesis, Dept. of EE and CS, Massachusetts Institute of Technology, 1994.

[13] T. A. Marsland, Yaoqing Gao, A. Reinefeld, and A. Yonezawa. Multiple principal variation splitting search. In *High Performance Computing Symposium, HPCS'95*, pages 292–303, Montreal, July 1995.

[14] T. Anthony Marsland and Fred Popowich. Parallel game-tree search. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, PAMI-7(4):442–452, July 1985.

[15] T.A. Marsland, Yaoqing Gao, and Francis C.M. Lau. A study of software multithreading in distributed systems. Technical Report TR 95-23, Department of Computing Science, University of Alberta, Edmonton, Canada, November 1995.

[16] S. Petri and H. Langendőrfer. Load balancing and fault tolerance in workstation clusters – migrating groups of communicating processes. *Operating Systems Review*, 29(4):25–36, October 1995.

[17] Jim Pruyne and Miron Livny. Providing resource management services to parallel applications. In *Proceedings of the Second Workshop on Environments and Tools for Parallel Scientific Computing*, May 1995.

[18] Jonathan Schaeffer. Distributed game-tree searching. *Journal of Parallel and Distributed Computing*, 6:90–114, 1989.

[19] Igor R. Steinberg and Marvin Solomon. Searching game trees in parallel. In *Proceedings of the IEEE International Conference on Parallel Processing*, pages III–9 – III–17, 1990.

[20] Erik Tarnvik. Dynamo – a portable tool for dynamic load balancing on distributed memory multicomputers. *Concurrency: Practice and Experience*, 6(8):613–639, December 1994.

[21] Jean-Christophe Weill. The ABDADA distributed minimax search algorithm. In *Proceedings 1996 ACM Computer Science Conference*, pages 131–138, Philadelphia, 1996.