# The Performance of a Highly Unstructured Parallel Algorithm on the KSR1*

Tom Nurkkala
nurkkala@cs.umn.edu

Vipin Kumar
kumar@cs.umn.edu

Department of Computer Science, University of Minnesota, Minneapolis, MN 55455

## Abstract

*This paper examines the performance on the Kendall Square Research KSR1 multicomputer of a highly unstructured algorithm for natural language parsing. It describes a Tree Adjoining Grammar parsing algorithm that exhibits near linear speedup and very high efficiency for grammars of even moderate size. The work reported demonstrates the utility of shared-address-space parallel architectures for algorithms that require shared data structures. Finally, the paper presents practical guidelines for the efficient use of the KSR1.*

## 1 Introduction

Unstructured parallel algorithms exhibit highly irregular and unpredictable computational requirements and communication patterns. Naive implementations of such algorithms on message-passing architectures perform poorly because they can lead to work load imbalance, processor idling, duplicate work, and high communication overhead. Shared-address-space machines, like the KSR1, appear to offer substantial performance advantages for unstructured algorithms, because communication latencies on these machines are much smaller than those on message passing architectures. This lower latency allows shared-address-space machines to exploit fine-grain parallelism more effectively than message-passing multicomputers.

Natural language parsing using Tree Adjoining Grammar (TAG) is one such unstructured algorithm. TAG is a powerful grammatical formalism for large-scale natural language processing [3, 9]. However, the computational complexity of parsing algorithms for TAG is high. For an $n$-word input sentence, TAG parsers based on the CYK parsing model [1, 3] exhibit tightly-bounded asymptotic complexity of $\Theta(n^6)$. Earley-style TAG parsers [2, 8] also have an upper-bound complexity of $O(n^6)$.

Given this computational complexity, the principle constraint on natural language research using TAG is the amount of time necessary to parse sentences. The best-known *sequential* TAG parser, developed at the University of Pennsylvania's Institute for Research in Cognitive Science (IRCS), delivers only 80–100 parsed sentences *per day*, from an ever-growing corpus of over 30,000 sentences. Computational restrictions also prevent experiments on sentences longer than about 15 words; the linguistic constructs of longer sentences simply cannot be explored in detail due to the runtime of the parser.

Thus, TAG parsing is a good candidate for parallelization. However, TAG parsers demonstrate highly unstructured behavior. They exhibit highly unbalanced computation, resulting in poor efficiency due to processor idling, load imbalance, and communication overhead.

Our earlier work considered CYK-style TAG parsing on a message-passing architecture [7]. In this paper, we examine an Earley-style TAG parser on the shared-address-space KSR1. The research reflected in this paper:

1. explores the performance characteristics of a highly unstructured algorithm—natural language parsing using Tree Adjoining Grammar.

2. demonstrates the utility of shared-address-space architectures for unstructured problems that make extensive use of shared data structures, particularly *queues* and *hash tables*.

3. evaluates the performance of the KSR1, highlighting how its shared-address-space architecture differs from a true shared-memory machine.

*This work was supported by IST/BMDO through Army Research Office grant number DAA H04-93-G-0080, the Army High Performance Computing Research Center at the University of Minnesota, National Science Foundation grant number IRI-9216941, and Army Research Office grant number DA/DAAH04-93-G-0131

## 2 Tree Adjoining Grammar

A Tree Adjoining Grammar consists of two finite, disjoint sets of terminal and non-terminal symbols, and two sets of trees, ***initial trees*** and ***auxiliary trees***, collectively referred to as ***elementary trees*** (Figure 1). The nodes of elementary trees are generically referred to as ***grammar nodes***. One non-terminal symbol is distinguished as the ***start symbol*** of the grammar.

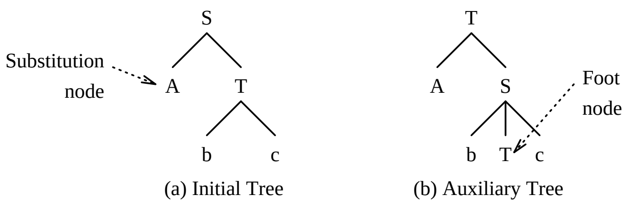


Figure 1: Types of TAG trees

In each elementary tree, interior nodes are labeled by non-terminals, and leaf nodes are labeled by either terminals or non-terminals. We show non-terminals as upper-case letters or strings, and terminals as lower-case letters or strings. Leaf nodes labeled by non-terminals are called ***substitution nodes***. In an auxiliary tree, exactly one of its leaf nodes (the ***foot node***) must be labeled by the same non-terminal that labels its root. In natural language processing applications, initial trees represent basic categories such as noun phrases, verb phrases, and sentences. Auxiliary trees encode modifiers, such as adjectives, adverbs, and relative clauses.

Elementary trees are composed by two operations: ***substitution***, which uses initial trees and ***adjunction***, which uses auxiliary trees. The resulting tree is a ***derived tree***, which can likewise be composed with other trees. Substitution (Figure 2) replaces a substitution node with an initial tree whose root has the same label as the substitution node.

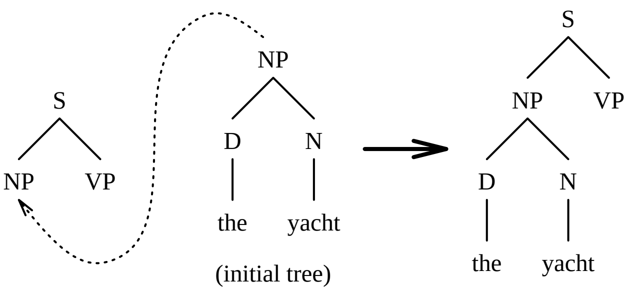


Figure 2: Substitution

Adjunction (Figure 3) inserts an auxiliary tree inside another tree. Adjunction takes place at an internal node (the ***adjunction site***) whose label is identical to that of the auxiliary tree's root node. Its effect is to create a new tree in which the subtree rooted at the adjunction site is replaced by the auxiliary tree, and the excised subtree is reattached at the foot node of the auxiliary tree.

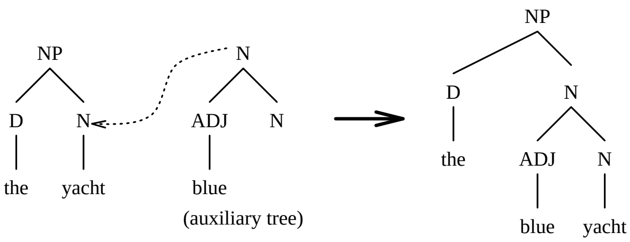


Figure 3: Adjunction

Given a TAG and an input string, a TAG parsing algorithm applies a series of substitution and adjunction operations in an attempt to compose a derived tree whose root is labeled by the start symbol, and whose leaves consist of exactly the input string. If such a tree can be composed, the TAG ***accepts*** the input string; otherwise, it ***rejects*** it.

## 3 Serial Algorithm

Our algorithm is an Earley-style TAG parser, based on the work of Schabes [8]. Such parsers use three basic types of operators:[1]

1. Scanner—scans input tokens.
2. Predictor—makes top-down predictions that specific grammar rules will be used.
3. Completor—indicates bottom-up completions of grammar rules predicted by an earlier prediction operation.

The basic structure of our TAG parsing algorithm is shown in Algorithm 1. The algorithm maintains an ***agenda*** of ***states***. The agenda is a queue of states awaiting processing. Each state represents a single application of one of the above operators. Step 4 of the algorithm typically generates new states that are added to the agenda.

[1] These operators are sufficient for context-free grammar parsers. TAG parsing is considerably more complex and requires more operators. However, for the purposes of this paper, this overview is sufficient.

1. Initialize the agenda
2. **while** states remain on the agenda
3. Retrieve the first state on the agenda
4. Select and execute a state-specific operator
5. Recognize

Algorithm 1: Earley-style TAG parsing algorithm

The algorithm makes extensive use of hashing. Informally, it must match the states processed during the prediction operation (while working top-down) with those generated during the corresponding completion operation (while working bottom-up). The algorithm also uses hashing to avoid processing duplicate states.

## 4 Parallel Formulation

We have parallelized Algorithm 1 on the KSR1 using its ***pthread*** mechanism. A pthread is a lightweight process that is the fundamental unit of concurrency on the KSR1 [4]. Each pthread executes on a separate processor, which has its own local memory. Pthreads communicate by accessing other processors' memories using the KSR ALLCACHE mechanism. ALLCACHE provides a single global address space and treats the entire physical memory as cache. That is, a data object has no permanent home in memory—it can be moved from one processor's cache to another as it is accessed and updated. ALLCACHE moves data between processors in 128-byte units called ***subpages***. The KSR1 provides exclusive access to data through its ***mutex*** primitives, which are fairly standard blocking mutual exclusion operations.

In the parallel formulation, each pthread executes the **while** loop in Algorithm 1. A simple minded approach to parallelization would be to use a single global agenda from which each processor retrieved states to be processed. However, for a large number of processors, a global agenda would become a bottleneck, resulting in idling during agenda access and reducing the algorithm's efficiency.

To prevent such a bottleneck, each pthread maintains a local agenda; as states are generated in step 4, they are added to the local agenda. Pthreads perform dynamic load balancing when their local agendas become empty. The algorithm associates a mutex with each agenda. When a pthread runs out of work, it randomly chooses a ***donor*** pthread, locks the donor's agenda, and steals a fixed fraction of the donor's work. The performance and scalability of this load balancing scheme is expected to be similar to that of Kumar et. al.'s random polling scheme [6].

Conceptually, the KSR1 is a shared-memory architecture. However, it is a non-uniform memory access (NUMA) machine—the ratio of local[2] to remote access times is about 1:300 [4]. This ratio implies that the KSR1 cannot be programmed as if it were a true shared memory machine. We found the following guidelines critical to implementing a high-efficiency KSR1 algorithm.

**Avoid frequent updates to shared data.** When a pthread reads from memory, ALLCACHE duplicates the subpage on which the data is stored in the local cache of the reading pthread. However, when any processor writes to a subpage, ALLCACHE must grant it exclusive access to the corresponding subpage. All copies of the subpage at other processors are invalidated by the KSR1, causing a subpage miss when another processor next reads any data on the subpage.

Our implementation makes liberal use of the large KSR1 memory space (32 MB per processor) to avoid updates to shared data. The entire grammar is loaded into memory, and read by each pthread before parsing begins. This has the effect of a one-to-all broadcast of the grammar, since each pthread acquires a copy of the grammar in its local memory.

The global hash tables for duplicate state elimination and prediction-completion matching are also optimized to avoid writes. A common strategy for implementing hash tables on a sequential machine is to update the contents of each hash bucket by inserting new elements at the head of a linked list, thus avoiding a list traversal. However, on the KSR1, updating the head of each list requires exclusive access to its subpage, invalidating other copies of the list head. Instead, our implementation adds to the *end* of the list, leaving copies of earlier list elements intact on other processors.

**Avoid blocking due to mutual exclusion.** To implement mutual exclusion, ALLCACHE puts a subpage into ***atomic*** state. This invalidates all other copies of the subpage; thus, when the locked subpage is released and requested by another processor, a subpage miss occurs. Although misses are satisfied quickly (300 $\mu$s maximum when the network is not saturated), heavy use of a small number of mutexes can severely limit performance.

[2]Here, "local" refers to the ***subcache*** in each KSR1 processing cell; the subcache lies between the processor and the main ALLCACHE memory for that cell.

To avoid contention over mutexes, our implementation uses a mutex for each agenda, and one for every bucket of each hash table. Thus, the amount of overhead due to locking minimized.

**Avoid standard C libraries.** The KSR1's standard C I/O library is *not* re-entrant.[3] Thus, calls to the library must be protected by mutexes, which quickly become a performance bottleneck. Furthermore, dynamic memory allocation is serialized. Programs should avoid calling library functions based on the `malloc` memory allocator.

Our algorithm initially used the C `sprintf` library function to generate hash keys, thus making frequent use of a single mutex to lock access to the standard I/O library. Rewriting the code to avoid the mutexes required by the I/O library approximately halved the algorithm's execution time.

In addition, our implementation initially called the standard `strdup` function (which calls `malloc`) to dynamically allocate small strings for hash keys and other uses. On one pthread, `strdup` executed in an average time of 179 $\mu$s; on 16 pthreads, the same call took 2.86 ms, almost exactly 16 times as long. Rewriting to avoid the `strdup` function improved the algorithm's speedup from 4.9 to 25.6 on 32 pthreads.[4]

**Use a dedicated processor set.** Several KSR1 users can share the same set of processors simultaneously. We found that operating in such a shared partition can yield unpredictable and misleading performance results, apparently due to the overhead incurred by the KSR1 to perform context switching between processors. This overhead results in timing inconsistencies both within and between experimental trials.

Our timing data were gathered using the KSR1's `allocate_pset` command. This command makes it possible for a single user to reserve a dedicated set of processors. Using such a processor set yields much more consistent performance measurements.

## 5 Experimental Results

Tables 1 and 2 show the algorithm's performance on a 1024-tree Tree Adjoining Grammar. For $p$ pthreads, the table shows parallel run time in seconds ($T_p$), speedup ($S = T_1/T_p$), and efficiency ($E = S/p$).

| $p$ | $T_p$ | $S$ | $E$ |
|---|---|---|---|
| 1 | 568.649 | 1.000 | 1.000 |
| 2 | 324.588 | 1.752 | 0.876 |
| 4 | 168.290 | 3.379 | 0.845 |
| 8 | 82.415 | 6.900 | 0.862 |
| 16 | 39.128 | 14.533 | 0.908 |
| 32 | 19.078 | 29.807 | 0.931 |
| 64 | 8.743 | 65.040 | 1.016 |

Table 1: Performance results (user time) on a 1024-tree grammar.

| $p$ | $T_p$ | $S$ | $E$ |
|---|---|---|---|
| 1 | 643.210 | 1.000 | 1.000 |
| 2 | 365.106 | 1.762 | 0.881 |
| 4 | 183.521 | 3.505 | 0.876 |
| 8 | 96.715 | 6.651 | 0.831 |
| 16 | 43.355 | 14.836 | 0.927 |
| 32 | 25.112 | 25.614 | 0.800 |
| 64 | 10.859 | 59.233 | 0.926 |

Table 2: Performance results (wall time) on a 1024-tree grammar.

Table 1 shows the ***user time***, which is the time spent executing the parser itself. Table 2 shows the ***wall time***, which is the total execution time of the algorithm. The apparent superlinear speedup in Table 1 for $p = 64$ is due to inaccuracies in the KSR1 timing functions for small values of $T_p$. Wall time includes the overhead of the ALLCACHE mechanism as it retrieves subpages from other processors. The 64-processor ALLCACHE interconnect on our experimental platform consists of two rings of 32 processors, interconnected by a third ring. the latency.

Figures 4 and 5 plot the algorithm's speedup and efficiency on up to 64 processors. These figures are based on user time. Similarly, figures 6 and 7 plot the algorithm's speedup and efficiency based on wall time.

For small grammar sizes, efficiency drops off markedly as $p$ increases. This decline is particularly evident for wall time. As the grammar size increases, the total number of states to be processed in step 4 of the algorithm increases. This increase means that each pthread's local agenda will seldom be empty, minimizing idling and insuring high efficiency. As is evident from Figures 5 and 7, the algorithm demonstrates behavior typical for many parallel algorithms—efficiency varies inversely with the number of processors and directly with the problem size.

[3] Nor is this fact documented by KSR!

[4] These times were obtained while running in a shared partition of KSR processors, rather than on a dedicated partition. The latter would have yielded better performance.

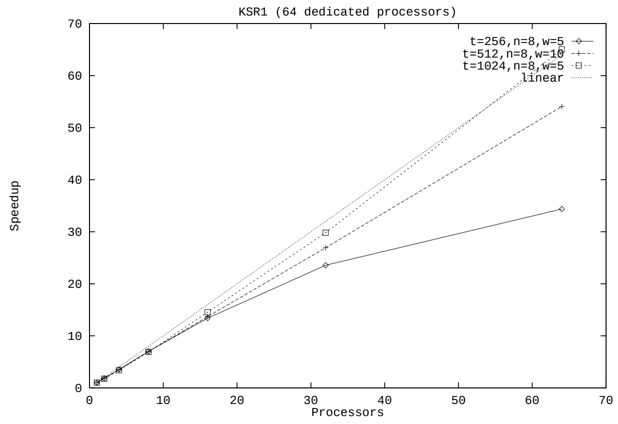


Figure 4: Speedup on up to 64 dedicated processors (based on user time). For each plot, **t** is the number of trees in the grammar, **n** is the average number of nodes per grammar tree, and **w** is the number of words in the input string.

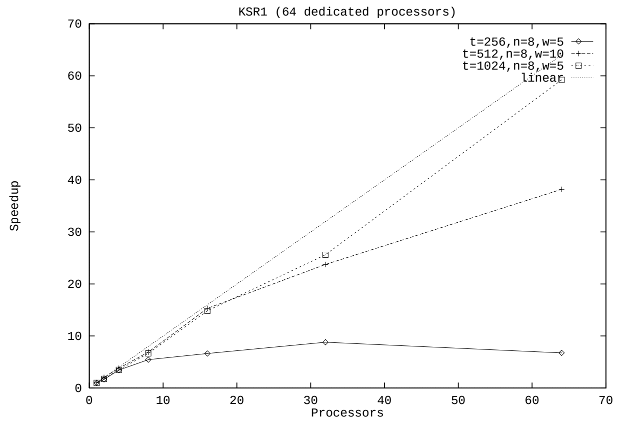


Figure 6: Speedup on up to 64 dedicated processors (based on wall time).

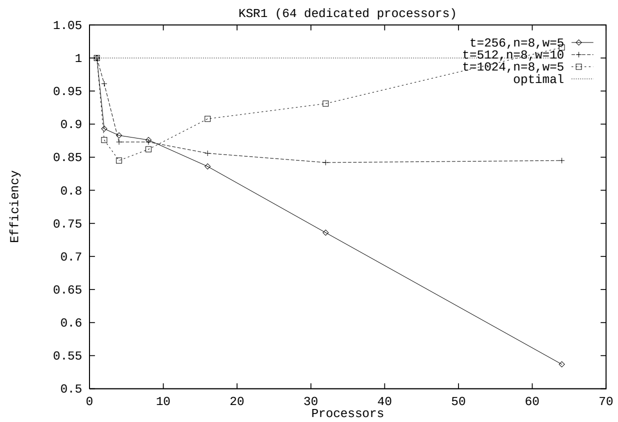


Figure 5: Efficiency on up to 64 dedicated processors (based on user time).

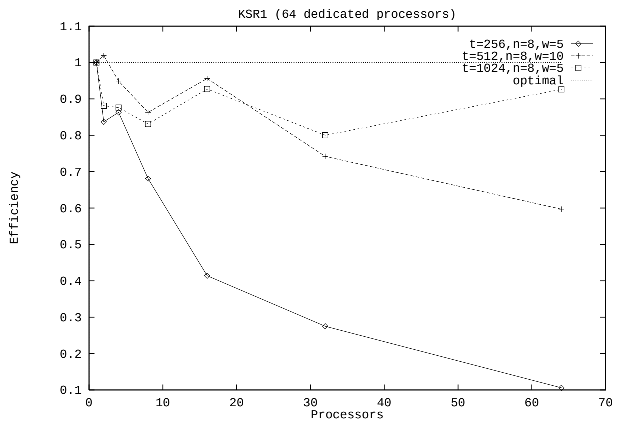


Figure 7: Efficiency on up to 64 dedicated processors (based on wall time).

For 1024-tree grammars, the wall time efficiency remains very high up to 64 processors. Thus, for sufficiently large grammars, the algorithm delivers nearly linear speedup. Note that 1024-tree grammars are only of moderate size, compared to the effective size of the grammar in actual use at the University of Pennsylvania. Because the Penn grammar will only grow in size, we expect to have access to grammars that are sufficiently large to insure excellent performance even on larger multicomputers.

## 6 Conclusion

We have successfully implemented a high-performance parallel TAG parser on the shared-address-space KSR1. Although it is highly unstructured, the algorithm exhibits good speedup and high efficiency, subject to the limitations of the KSR1's communication architecture.

With the exception of problems with its standard libraries, the KSR1 is exceptionally easy to program when compared to message-passing architectures. Operations like one-to-all and all-to-all broadcasts [5] become simple memory references. Distributed hashing is straightforward to implement and performs well by merit of the ALLCACHE mechanism. Complex tasks, like dynamic load balancing, require only a few lines of code for locks and memory reads.

Ease of use, however, does not come for free. Shared-address-space architectures, such as the KSR1, have higher cost than message-passing architectures (although the cost differential is decreasing). The shared-address-space programming paradigm also makes it deceptively easy for the algorithm designer to ignore data locality, which is critical to a high performance parallel algorithm. It is unclear whether the bandwidth of the interconnect is sufficient to handle traffic from a very large number of processors. The processor speed on our experimental platform (20 MHz) seems artificially slow in order to remain matched to the interconnect.

The ALLCACHE mechanism also introduces its own inefficiencies. For small data elements (i.e., those smaller than a subpage), several elements can be stored per subpage. But in such a scheme, updates to any elements on the subpage will invalidate all other cached copies of the subpage, even if remote processors never access the newly updated data. To avoid such subpage misses, data must be allocated on subpage boundaries, resulting in considerable wasted storage for small data objects.

On balance, however, the KSR1 is a very productive platform. It offers an easy-to-program environment in which to experiment with parallel algorithm designs.

## Acknowledgments

Sartaj Sahni at the University of Florida graciously provided access to their KSR1 multicomputer. Randy Fischer, also at the University of Florida, gave invaluable and timely assistance on the technical details of the KSR1.

## References

[1] A. Aho and J. Ullman. *The Theory of Parsing, Translation, and Compiling, Vol. 1: Parsing.* Prentice-Hall, 1982.

[2] J. Earley. An efficient context-free parsing algorithm. *Communications of the ACM*, 13:94–102, 1970.

[3] A. K. Joshi, L. S. Levy, and M. Takahashi. Tree adjunct grammars. *Journal of Computer and System Sciences*, 10:136–63, 1975.

[4] Kendall Square Research, 170 Tracer Lane, Waltham, MA 021454-1379. *KSR Parallel Programming*, Feburary 1992.

[5] Vipin Kumar, Ananth Grama, Anshul Gupta, and George Karypis. *Introduction to Parallel Computing: Design and Analysis of Algorithms.* Benjamin/Cummings, 1994.

[6] Vipin Kumar, Ananth Grama, and V. Nageshwara Rao. Scalable load balancing techniques for parallel computers. Technical Report 91-55, Computer Science Department, University of Minnesota, 1991. To appear in *Journal of Distributed and Parallel Computing*, 1994.

[7] Tom Nurkkala and Vipin Kumar. A parallel algorithm for natural language parsing using tree adjoining grammar. In *Proceedings of the International Parallel Processing Symposium*, 1994 (to appear).

[8] Yves Schabes. *Mathematical and Computational Aspects of Lexicalized Grammars.* PhD thesis, University of Pennsylvania, Department of Computer and Information Science, August 1990.

[9] K. Vijay-Shankar and A. K. Joshi. Some computational properties of tree adjoining grammars. In *Proceedings of the 11th Meeting of the Association of Computational Linguistics*, August 1986.