Effects of Parallelism on Blackboard System Scheduling

Keith Decker, Alan Garvey, Marty Humphrey and Victor Lesser *
Department of Computer and Information Science
University of Massachusetts
Amherst, MA 01003

Abstract

This paper investigates the effects of parallelism on blackboard system scheduling. A parallel blackboard system is described that allows multiple knowledge source instantiations to execute in parallel using a shared-memory blackboard approach. New classes of control knowledge are defined that use information about the relationships between system goals to schedule tasks — this control knowledge is implemented in the DVMT application on a Sequent multiprocessor using BBI-style control heuristics. The usefulness of the heuristics is examined by comparing the effectiveness of problem-solving with and without the heuristics (as a group and individually). Problem solving with the new control knowledge results in increased processor utilization and decreased total execution time.

1 Introduction

From the beginning the blackboard paradigm has been developed with parallelism in mind [Lesser *et al.,* 1975]. The idea of independent Knowledge Sources (KSs) that communicate only through a shared blackboard is a model that inherently encourages parallel execution. Many researchers have looked at making blackboard systems execute in parallel [Corkill, 1989, Fennell and Lesser, 1977, Rice *et* a/., 1989].

In particular the execution of multiple Knowledge Source Instantiations (KSI) in parallel (known as knowledge source parallelism) is discussed in several places in the literature. The Advanced Architectures Project at Stanford University [Rice, 1989, Rice *et* al., 1989] has investigated blackboard parallelism at several levels of granularity. Their Cage architecture takes the existing AGE blackboard architecture and extends it to execute concurrently at several granularities, including knowledge source parallelism [Nii *et* al., 1989]. Another project that investigated knowledge source parallelism is the work by [Fennell and Lesser, 1977] to study the effects of parallelism on the Hearsay II speech understanding system [Erman *et* al., 1980]. One major contribution of that project is a detailed study of blackboard locking mechanisms. An alternative method for enforcing data consistency is the use of transactions as described by [Ensor and Gabbe, 1985].

One distinguishing feature of these studies of parallelism in blackboard systems is that they used simulated parallelism. Concurrently executing processes and interprocess communication were simulated using complex models of parallel environments. This was the case primarily because of the primitive nature of existing parallel hardware and the lack of sophisticated software development environments. Only recently have hardware and software capabilities come together to allow the actual implementation of parallel blackboard systems [Bisiani and Forin, 1989]. Useful parallel programming environments now exist, including implementations of Lisp. The work described in this paper was done on a Sequent multiprocessor using Top Level Common Lisp[1] (a version of Lisp that supports concurrent processing) and a special version of GBB 2.0[2] that was modified to allow parallel read access to the blackboard and to allow locking only part of the blackboard on write accesses.

Along with our actual use of parallel hardware, a major difference between our work and previous research is our focus on how control knowledge needs to change in a parallel environment. Previous work was more interested in investigating specific parallel models, while we are interested in what new control knowledge is useful. As we show, it is not enough to just add locks to a sequential blackboard system and execute the top n KSIs as rated by sequential control heuristics on *n* parallel processors. New heuristics need to be added that are sensitive to the requirements for effective parallel execution. Earlier work on Partial Global Planning[Durfee and Lesser, 1987] showed that constructing schedules using a high-level \iew of the solution space (derived by distributed agents from goal relationships) improved the utilization of distributed processors. This leads to the intuition that using goal relationships in scheduling may be helpful in

*The authors are listed in alphabetical order. This work was partly supported by the Office of Naval Research under a University Research Initiative grant number N00014-86-K-0764, NSF contract CDA 8922572, ONR contract N00014-89-J-1877, and a gift from Texas Instruments.

[1]Top Level Common Lisp is a trademark of Top Level, Inc.

[2]GBB 2.0 is a trademark of Blackboard Technologies, Inc.

a single agent, parallel-processing situation.

The next section discusses the details of our parallel architecture. Section 2.3 describes the new kinds of control knowledge that are useful in a parallel environment and identifies the kinds of data that are required to implement those new kinds of knowledge. Section 3 briefly introduces the Distributed Vehicle Monitoring Testbed, the application that motivates this work, and describes the new heuristics that were added for parallelism. Section 3.1 presents and discusses the results of the experiments we performed on the system, including the performance of the implementation and the effect of the added control heuristics. The final section summarizes the work and describes future research directions.

2 Architecture

The architecture is a straightforward extension to the BB1-stvie uniprocessor blackboard architecture described in [Decker *et* al., 1989, Decker *et* al., 1990]. In that architecture the processor takes the currently top rated KSI from the executable agenda. It executes the KSI action, which creates and modifies blackboard hypotheses. These hypotheses in turn stimulate goals, which trigger new KSs for execution. A scheduler orders the executable agenda and the loop begins again. This low-level control loop is highly parameterized; the parameters are set by control knowledge sources.

In our parallel system, several processors execute the low-level control loop concurrently. AD of the hypotheses, goals, agendas, KSs, and control parameters are posted on shared global domain and control blackboards that are accessible to each processor. This basic architecture is very similar to the Cage simulations with KS-level parallelism and asynchronous control [Rice *et a*/., 1989].

Besides executing the low-level control loop in parallel, three major modifications to the existing system are made to allow effective use of parallelism. Locking mechanisms are provided to prevent conflicting blackboard accesses; the control knowledge sources are run sequentially, but the actions of many control KSs are broken up and run in parallel; and new classes of control knowledge are identified and implemented. The next three sections describe each of these modifications in more detail.

2.1 Blackboard Locking

Various schemes for blackboard locking appear in the literature. The most detailed is that described by Fennell and Lesser, who describe a method for locking blackboards that assures data integrity [Fennell and Lesser, 1977].

We have found that a much simpler locking mechanism is sufficient for our system. The only locking mechanism we provide is atomic read/write locks for blackboard writes. This mechanism is invoked when a blackboard write is done. It executes a read to see if the object to be written already exists. If it does, then the new object is merged with the existing object, otherwise the new object is written. Knowledge sources are designed so that they create hypotheses one at a time. Since only one lock is ever acquired at a time, deadlock is impossible. The operating system scheduler prevents starvation.

This simple mechanism is sufficient because the system can build several, possibly conflicting, partial solutions to a problem [Lesser and Corkill, 1981]. It does not require exactly one consistent working solution, so it does not return to and delete objects that cause inconsistencies. Because hypotheses are never deleted, the structure of the hypotheses on the blackboard never changes; only the beliefs in existing hypotheses may change. Changes in belief can be recognized and propagated by a separate knowledge source. If new hypotheses are created that would produce different results, then their creation will trigger new knowledge source instantiations that may be scheduled. We believe that other systems that share this characteristic will find that simple locking mechanisms are adequate. For example, the AGORA system uses "write-once" memory management where a blackboard element cannot be updated in place, but rather a copy is made [Bisiani and Forin, 1989].

Several types of locks were used in the implementation. Each blackboard level (space) is divided into a set of *buckets. A* blackboard data unit is stored in a small number of buckets based on its characteristics. Each bucket is given its own lock. Thus two KSIs can always write to different blackboard levels in parallel but one might block if they both write to the same bucket. Locks also control access to the KSI agenda and other internal data structures associated with the low-level control loop.

2.2 Executing the Control Knowledge Sources

Control knowledge sources change parameters that allow control over mechanisms such as filtering hypotheses or goals, merging hypotheses or goals, mapping from hypotheses to goals or goals to KSs, and the agenda rating mechanism. Control knowledge triggers on events on both the domain and control blackboards. In our current system control knowledge sources are run sequentially (meaning only one control KSI is executing at a time), however the actions of many of the control KSIs divide up the work and execute it in parallel on several processors. In the future we would like to investigate executing the control knowledge in parallel as well.

2.3 New Classes of Control Knowledge for Parallelism

Normal control knowledge (as used in a sequential environment) rates KSIs based on knowledge such as the belief of their input data, the potential belief in the output data, the significance of the output data given the current system goals, and their efficiency or reliability. In addition to these kinds of control knowledge there are several general classes of control knowledge that can be added to more effectively execute KSs in parallel. These general classes of control knowledge include:

Access Collisions: To avoid excessive conflicts for blackboard access, do not schedule two KSIs to work in the same part of the search space at the same time. For example, if KSI A and KSI B both write to a particular level of the blackboard, then they should not be scheduled for execution at the same

time, because one of them will have to wait for the other to relinquish blackboard locks.

Task Ordering: Tasks may have absolute, unchangeable orderings (meaning they cannot be scheduled to execute in parallel at all), or there may be interdependence among tasks that lead to ordering preferences (one task provides data that will significantly affect the speed or quality of the result of another task.) For example, Task A may produce a result that makes the performance of Task B much faster. If so, Task A should be scheduled before Task B.

Task Bottlenecking: Performing certain tasks earlier in problem-solving may reduce future sequential bottlenecks. In general it is preferable to execute tasks that will allow more parallel options later. For example, there may be an absolute task ordering that requires that Task A be performed before Tasks B, C, and D, which can then be performed in parallel. Task A should be performed as soon as possible, because it will allow more parallelism later.

Task Invalidation: This is based on the "Competition Principle" in Hearsay-II [Hayes-Roth and Lesser, 1977]: the results of some tasks may completely remove the need to execute other tasks. Thus, when currently executing tasks are taken into account (assumed to complete), some pending tasks will be obviated. For example, Task A and Task B may perform the same operation, and produce the same result, in different ways. If Task A has been scheduled, then Task B should not be immediately scheduled, because it will be obviated if Task A completes successfully.

To take these general classes of control knowledge into account the system requires particular kinds of knowledge about the domain KSIs. In particular, avoiding *access collisions* requires knowledge about the input/output characteristics of a KSI (i.e., what parts of the blackboard it accesses and modifies.) *Task ordering* requires knowledge about task interactions. Often this knowledge is best captured through relationships among the goals of particular tasks. Avoiding *task bottlenecking* requires knowledge about the probable outcomes of tasks, again often expressed through goal relationships. *Task invalidation* uses knowledge about supergoal and subgoal relationships to understand the effect of KSI executions on other KSIs' goals.

There are four general categories of goal relationships that can be used (via KSI rating heuristic functions) to schedule domain KSs [Decker and Lesser, 1990]:

Domain Relations: This set of relations is generic in that they apply to multiple domains and domain dependent in that they can be evaluated only with respect to a particular domain, e.g., inhibits, cancels, constrains, predicts, causes, enables, and supergoal/subgoal (from which many useful graph relations can be computed, as shown below). These relations provide *task ordering* constraints, represented by temporal relations on the goals (see below).

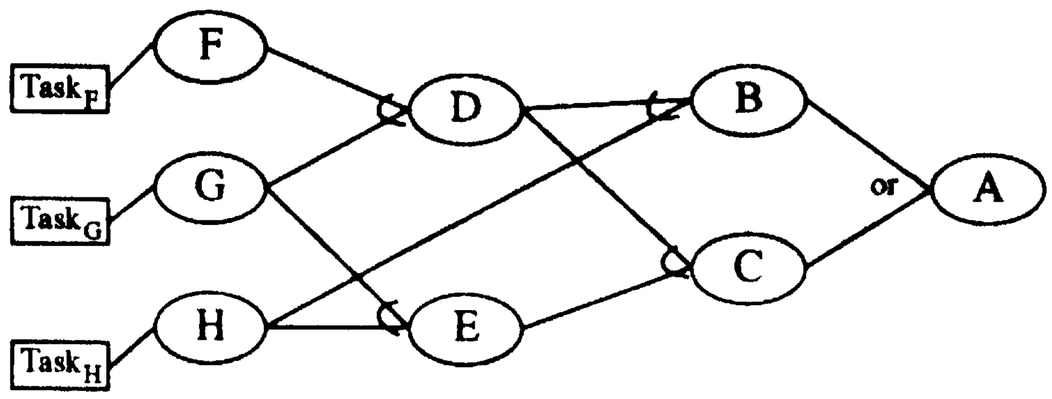

Figure 1: An abstracted goal relation graph

Graph Relations: Some generic goal relations can be derived from the supergoal/subgoal graphical structure of goals and subgoals, e.g., overlaps, necessary, sufficient, extends, subsumes, competes. The *competes* relation is used to produce *task invalidation* constraints. These relations also produce *task bottlenecking* information.

Temporal Relations: From Allen [Allen, 1984], these include before, equal, meets, overlaps, during, starts, finishes, and their inverses. They can arise from domain relations, or depend on the scheduled timing of goals — their start and finish times, estimates of these, and real and estimated durations.

Non-computational Resource Constraints: A final type of relation is the use of physical, non-computational resources. Two tasks that both use a single exclusive resource cannot execute in parallel. For example, if two tasks require that a single sensor be aimed or tuned differently, they cannot execute in parallel.

For example, examine the goal structure in Figure 1 (abstracted from an actual domain goal relation graph). Assume that task $Task_F$ is currently executing on a processor. The arcs in the graph represent the *goal/subgoal* domain relation on the goals[3]. From only this one domain relation, we can tell for example that *F* and *G* are *necessary* for D, *D* is *necessary* for *B* and C, and *B* is *sufficient* for *A*. *F* and *G extend*[4] one another, as do *D* and *H*. Goal *B competes* with C

Thus a *task invalidation* heuristic might avoid scheduling a task that achieves goal *B* in parallel with one that achieves goal C. In the given situation (with $Task_F$ executing, and *G* and *H* available for processing), a *task bottleneck* heuristic might prefer to schedule a task to satisfy *G,* which will allow work on goals *D* and *H* in the future[5], over a task to satisfy *H,* which would allow

[3] While it looks similar, this is different from a typical data dependency diagram both in granularity and in the fact that it would be constructed dynamically during problem solving. At the present time we constructed one by hand to develop possible parallel heuristics for our domain.

[4] Goal 1 extends goal 2 if there exists a supergoal, goal 3, such that goals 1 and 2 are in the same AND conjunct.

[5] Goal *D* would become open, since its necessary subgoals *F* and *G* would be completed; goal *H* already was open.

only work on goal *G* in the future. Of course, tasks may accomplish multiple goals, a fact that is simplified in this example. A *task ordering* heuristic would not find any temporal relations in this example; they are induced by domain relations where goals *constrain* or *predict* others.

3 Experiments

Experiments were run on the Distributed Vehicle Monitoring Testbed (DVMT)[Lesser and Corkill, 1983], a knowledge-based signal interpretation system. The input to the DVMT is acoustic signals generated by moving vehicles and detected by acoustic sensors. The goal of the DVMT is to identify, locate and track patterns of vehicles moving through a two-dimensional space. The four main blackboard levels are: *signal* (for processing of signal data), *group* (for collections of signals attributed to a single vehicle), *vehicle* (for collections of groups that correspond to a single vehicle), and *pattern* (for collections of vehicles acting in a coordinated manner).

For our purposes, DVMT domain KSs can be divided into two main classes: synthesis and track extension. Synthesis KSs combine one or more related hypotheses at one level of the blackboard into a new hypothesis at the next higher level. Track extension KSs output track hypotheses, where a track is a list of sequential pieces of time-location data that identify the movements of a vehicle. The control KSs of the system can also be divided into two main classes: those that implement a goal-directed strategy and those that extend that strategy for parallel execution.

For these experiments, the DVMT processes the input data in three distinct phases. In the first phase *(find initial vehicles),* an initial set of control KSs execute, configuring the DVMT to perform a thorough analysis of all data at time 1. The purpose of this phase is to roughly identify the type and position of all vehicles that will be tracked in the experiment[6]. The data file we used contained 12 vehicles, and we defined four possible vehicle types with some signals and groups of signals shared by multiple vehicles. After these control KSs execute, and the domain KSs process all time 1 data, more control KSs are triggered and execute, configuring the DVMT for the second phase of processing. In the second phase *(approximate short tracks)*, the DVMT performs quick, approximate processing to determine the likely identity and patterns of the vehicles being tracked. This phase ends when control KSs recognize that the DVMT has established a pattern (or explanation) for all the vehicles, though the patterns may be uncertain. For these experiments, we defined a short track to contain at least four time-location data points-thus, this phase ended after processing the time 4 data. Again, control KSs assign new values to system parameters, and the third phase *(perform pattern-directed processing)* commences. The DVMT devotes most of its processing in this phase to tracking vehicles involved in primary patterns, while performing cursory processing on vehicles involved in secondary patterns. This phase continues until all data in the input file has been processed. In these experiments, we included data until time 9.

[6]We have restricted these experiments such that every vehicle appears in the first set of acoustic samples, in order to simplify processing.

3.1 Examining the Basic Parallel Architecture

The first set of experiments involved collecting statistics on the basic parallel architecture without any added heuristics to take advantage of the parallelism. These experiments demonstrate that the locking system works, that the basic architecture provides for a good utilization of processors, and that the domain and our problem-solving method provide inherent parallelism.

Data for runs of the environment on 1 processor with no special parallel heuristics are summarized in Table 1. This table shows the time the single processor uses in each phase (and between phases) and the percent of the total time spent in each phase (and between phases). This data is used in comparisons to the other experiments described later. In this and all later experiments, data was collected with the locking and metering mechanisms enabled. The locking mechanism itself had almost no overhead, and as much of the metering as possible is done on a separate processor, completely outside of the processors being used for the experiment. The data collected by the metering processor did not involve locking any of the target processors. All of the experiments were conducted on a 16 processor Sequent Symmetry, and all of the experiments used less than 16 available processors (so no tasks were swapped off a processor).

Table 2 shows the speedup resulting from 5 processors and no parallel heuristics. Phase 1 parallelism arises mostly from being able to process all the data from the sensors in parallel. The tasks in phases 1 and 2 all must be executed, which also causes a high degree of inherent parallelism.

Centralizing the meta-controller that is implemented by control KSs proved not to be a bottleneck in processing; when not changing phases the meta-controller is almost dormant (simply checking for the end of a phase), and most of the work involved in changing phases (setting up new hypothesis and goal filters and running the hypotheses through them) is done in parallel. By running the low-level control loop (hyp to goal to KS mapping) in parallel we avoided the control bottleneck observed by Rice *et al.* in their first Cage experiment, where a set of KSs was executed synchronously by the controller [Rice *et al.*, 1989]. The only time that the system ever has to synchronize is at the beginning of a phase change, as the agenda from the last phase is empty and that of the next phase is still being generated. We have already begun work to eliminate this bottleneck as well, using a *channelized* architecture that gives us the ability to have different pieces of data at different phases of processing simultaneously.

3.2 Examining the Parallel Heuristics

By simply allowing KSIs to run in parallel, we achieved a significant improvement in the DVMT performance. However, it is clear from the discussion in Section 2.3 that we should be able to do better than just taking the top (single processor) rated KSI off of the agenda.

	Phase 1	1–2	Phase 2	2–3	Phase 3	Total
Real Time (seconds)	5224	1066	2118	1796	5656	15860
Percent of total time	33.0	6.7	13.3	11.3	35.7	100

Table 1: Summary of results of basic system with 1 processor and without parallel heuristics

	Phase 1	1–2	Phase 2	2–3	Phase 3	Total
Real Time (seconds)	1453	314	481	462	1433	4143
Speedup over 1 processor	3.6	3.4	4.4	3.9	3.9	3.8

Table 2: Summary of results of basic system with 5 processors and without parallel heuristics

Four new heuristics were added to incorporate knowledge about how parallelism affects scheduling. The BBI-style controller [Hayes-Roth, 1985] rates each KSI against each heuristic of each active focus. The architecture supports two types of heuristics — numeric and pass/fail. Numeric heuristics are summed to produce a rating; pass/fail heuristics must pass a KSI or it will not be executed. Either type of heuristic can also decide not to rate a KSI — the effect is of a rating of 0 or 'pass' but it is recorded differently. All the previous non-parallel domain heuristics were numeric but some of the new parallel heuristics are pass/fail.

1. *Pass Non-obviated Outputs.* Schedule KSIs that will not produce output that will be obviated if the currently executing KSIs complete successfully. This heuristic implements the *task invalidation* criteria described in Section 2.3. The usefulness of this heuristic is tied to the success rate of the KSIs in question — if the KSI currently executing is likely to finish successfully, then the heuristic will be likely to avoid duplicate work. This is a pass/fail heuristic — if there are no tasks available that will not be obviated by existing tasks, then the processor will wait. This heuristic is not needed in the single processor case because when a KSI completes its action, all KSIs that it obviates are removed from the agenda before the next KSI is chosen.

2. *Pass Primary Patterns.* This heuristic, which was a numeric rating heuristic in the single-processor system, was changed to a pass/fail heuristic. It is an example of a *task ordering* heuristic as described in Section 2.3. An implicit assumption of this heuristic is that KSs are not interruptible; so, when low priority KSIs are started, later arriving higher priority KSIs may not get a processor. When running an existing blackboard system in parallel, one should carefully examine the existing control heuristics to see if they will have the desired effect with multiple processors. While a high rating is sufficient in a single processor system to indicate that the KSs involved in a task are important to execute, in the multiprocessor case a decision must be made as to whether a processor should execute a KSI from a useful, but less important, task or wait idle for known important future tasks.

3. *Prefer Outputs on Different Regions.* Schedule KSIs that do not access the same blackboard regions as the currently executing KSIs. This heuristic implements the general access *collision* control knowledge described in Section 2.3. In our case, only blackboard write operations need to be locked. This heuristic will be more applicable in systems such as those described by [Fennell and Lesser, 1977] that do more elaborate locking. This is a numeric preference heuristic. Obviously this heuristic is not needed in the single processor case because only one KSI is being executed, so there cannot be any blackboard access collisions.

4. *Prefer Many Output Hyps.* Schedule KSIs that expect to produce many output hyps before those that expect to produce fewer output hyps. This heuristic implements the *task bottleneck* avoidance class of heuristics described above. By preferring to produce many outputs, more possible tasks may be enabled in the future. This is a weak numeric preference heuristic. This heuristic is not needed in the single processor case (in the DVMT domain) because the single processor will still have to execute all of the (non-obviated) KSIs, no matter how long the agenda is. The purpose of this heuristic is merely to get the queue to a long length quickly, improving multiple processor performance.

Table 3 is a comparison of the system with the four heuristics and 5 processors with the 1 processor system and the 5 processor system. We did not get as much speedup over the experiment without parallel heuristics as we had hoped. A primary reason for this is that the parallel version without heuristics was developed to run as fast and as efficiently as possible with 1 or more processors; we did not handicap it in any way. Our parameterized low-level control loop allows very few KSIs through that should not be executed (i.e., very little search). This hampers the heuristics especially in phases 1 and 2, which have very tight and precise control plans.

For example, the task obviation heuristic finds very few tasks to obviate. This is because we try to identify and filter out or merge hypotheses and goals that might create redundant tasks as early as possible (before they trigger KSs to form KSIs). However, this may not always be the best course to take — even our own system is being expanded to include multiple methods of achieving the same result by trading off some of the

	Phase 1	1–2	Phase 2	2–3	Phase 3	Total
Real Time (seconds)	1425	313	476	445	1181	3840
Speedup over 1 processor	3.7	3.4	4.5	4.0	4.8	4.1
Percent faster than 5 processors without heuristics	2.0	0.3	1.1	3.8	21.3	7.9

Table 3: Summary of results with 5 processors and parallel heuristics

	Phase 1	1–2	Phase 2	2–3	Phase 3	Total
Real Time Without Heuristic (seconds)	1474	555	755	455	1504	4743
Real Time With Heuristic (seconds)	1348	557	587	459	1436	4387
Percent Faster	8.5	0.0	22.2	-0.8	4.5	7.5

Table 4: Summary of results with and without the Task Obviation heuristic on 5 processors running a modified system

characteristics (such as precision and certainty) for time [Decker *et* al., 1990]. This may result in more potentially obviatable tasks on the agenda. We test this hypothesis in Section 3.2.1.

The access collision heuristic is also relatively weak. This is because, as we have previously stated, KSIs seldom block on writing to the same area of a blackboard level, and may read in parallel. Access collision avoidance may be more important in systems that must lock objects for a long time to modify them. We tested this hypothesis in Section 3.2.2. Another problem stems from the *prefer many output hyps* heuristic; the DVMT tends to already work this way as a side effect of the domain heuristics, therefore the heuristic will not show an appreciable improvement when present.

As stated previously, the agendas in phases 1 and 2 were tightly regulated — the execution of a KSI on the agenda was necessary for overall problem solving progress. KSIs were not likely to be obviated by other KSIs, and most KSIs were involved in "good" work. However, this was not the case in phase 3. KSIs were created whose output would often be subsumed by the output of another KSI if this other KSI were given an opportunity to run (the first KSI is a candidate for obviation), and a fair number of secondary pattern KSIs existed on the agenda at any point in time. Given these characteristics of the agenda in phase 3, 21.3% speedup over the 5 processor system to 4.8 times speedup over 1 processor was achieved. The parallel heuristics allowed processors to make intelligent decisions regarding the next KSI to execute. As a result of the parallel heuristics, more KSIs were obviated, and processors often delayed executing the next KSI if none of the KSIs on the agenda appeared particularly appealing. It is also important to note that utilization was down somewhat in phase 3 because of the parallel heuristics, but the overall end-to-end processing time was reduced.

3.2.1 Testing the Task Obviation Heuristic

In order to test the *task obviation* parallel heuristic, we modified the basic parallel system by disabling the *KSI-merging* feature. This results in many KSIs, triggered by different data but intending to satisfy identical or similar goals, being placed on the agenda[7]. The scenario was run once with five processors without any parallel heuristics, and then again with the addition of the single task obviation heuristic. The results are shown in Table 4. Speedup was achieved particularly in phase two, when the addition of the task obviation heuristic caused a sufficient number of KSIs to be obviated, because KSIs were executed in a data-directed manner and thus tended to obviate others upon completion.

3.2.2 Testing the Access Collision Heuristic

In order to test the *access collision* parallel heuristic, we configured the DVMT to allow higher potential contention for system locks, without otherwise handicapping the system. Specifically, we modified the basic parallel system by disabling the *KSI-merging* feature and by artificially lengthening the time processors spend in the blackboard bucket locks. The first modification forces the creation of a separate KSI for each output goal (rather than merging similar goals). Since KSIs that perform essentially the same activities tend to get rated approximately the same, a processor will select a KSI for execution that will create results that one or more other KSIs running at the same time might produce. The effect of not allowing similar KSIs to merge will produce a high amount of contention for blackboard regions, because similar KSIs will be executing at the same time. The second modification simulates a system that requires a larger context for KSI execution — one that keeps more of the blackboard locked for longer times. The larger the context, the higher the probability of contention. The scenario was run once without any parallel heuristics, and then again with the addition of the single access collision heuristic. Both runs were with five processors.

Adding the heuristic to avoid regions that other processors are utilising resulted in a significantly decreased

[7]A similar effect might have been achieved by activating multiple approximate processing methods, in addition to the normal precise methods, for each goal.

amount of time spent in locks. A processor in the run without the access collision heuristic spent an average of 18.4% of its time in locks, while the addition of the access collision heuristic reduced this time to 11.0%.

4 Conclusions

We investigated the effects of parallelism on blackboard scheduling. We have shown that we are able to get a speed-up for the DVMT application. We also showed that at least some of the speed-up was produced by our new heuristics that take parallel knowledge into account. It is our hypothesis that as the amount of search and interaction among search paths increases the heuristics will become more important. We are working to test this hypothesis.

As mentioned briefly in Section 3.1, we are currently investigating a *channelized* architecture that allows different vehicles to be in different phases of problem-solving simultaneously. That is, data for different vehicles can be processed using different filters, rated using different heuristics, and even use different approximate processing problem-solving methods all concurrently. We believe this has the potential to significantly increase parallelism, both because it removes the phase-changing bottleneck and because it clearly and simply divides up the work for parallel execution.

Acknowledgments

The authors would like to thank Kevin Q. Gallagher for his work in creating a shared memory parallel processing version of GBB 2.0.

References

[Allen, 1984] James F. Allen. Towards a general theory of action and time. *Artificial Intelligence,* 23:123-154, 1984.

[Bisiani and Forin, 1989] Roberto Bisiani and A. Forin. Parallelisation of blackboard architectures and the Agora system. In V. Jagannathan, Rajendra Dodhiawala, and Lawrence S. Baum, editors, *Blackboard Architectures and Applications.* Academic Press, 1989.

[Corkill, 1989] Daniel D. Corkill. Design alternatives for parallel and distributed blackboard systems. In V. Jagannathan, Rajendra Dodhiawala, and Lawrence S. Baum, editors, *Blackboard Architectures and Applications.* Academic Press, 1989.

[Decker and Lesser, 1990] Keith Decker and Victor Lesser. Extending the partial global planning framework for cooperative distributed problem solving network control. In *Proceedings of the Workshop on Innovative Approaches to Planning, Scheduling and Control,* pages 396-408, San Diego, November 1990. Morgan Kaufmann. Also COINS TR-90-81.

[Decker *et* al., 1989] Keith S. Decker, Marty A. Humphrey, and Victor R. Lesser. Experimenting with control in the DVMT. In *Proceedings of the Third Annual AAAI Workshop on Blackboard Systems,* Detroit, August 1989. Also COINS TR-89-85.

[Decker *et al,* 1990] Keith S. Decker, Victor R. Lesser, and Robert C. Whitehair. Extending a blackboard architecture for approximate processing. *The Journal of Real- Time Systems,* 2(I/2):47-79, 1990. Also COINS TR-89-115.

[Durfee and Lesser, 1987] Edmund H. Durfee and Victor R. Lesser. Using partial global plans to coordinate distributed problem solvers. In *Proceedings of the Tenth International Joint Conference on Artificial Intelligence,* August 1987.

[Ensor and Gabbe, 1985] J. Robert Ensor and John D. Gabbe. Transactional blackboards. In *Proceedings of the Ninth International Joint Conference on Artificial Intelligence,* pages 340-344, August 1985. Also published in Readings in Distributed Artificial Intelligence, Alan H. Bond and Les Gasser, editors, p. 557-561, Morgan Kaufman, 1988.

[Erman *et al,* 1980] L. D. Erman, F. Hayes-Roth, V. R. Lesser, and D. R. Reddy. The Hearsay-II speech-understanding system: Integrating knowledge to resolve uncertainty. *Computing Surveys,* 12(2):213-253, June 1980.

[Fennell and Lesser, 1977] R. D Fennell and V. R. Lesser. Parallelism in AI problem solving: A case study of Hearsay-II. *IEEE Transactions on Computers,* C-26(2):198-111, February 1977.

[Hayes-Roth and Lesser, 1977] Frederick Hayes-Roth and Victor R. Lesser. Focus of attention in the Hearsay-II speech understanding system. In *Proceedings of the Fifth International Joint Conference on Artificial Intelligence,* pages 27-35, August 1977.

[Hayes-Roth, 1985] Barbara Hayes-Roth. A blackboard architecture for control. *Artificial Intelligence,* 26:251-321, 1985.

[Lesser and Corkill, 198l] Victor R. Lesser and Daniel D. Corkill. Functionally accurate, cooperative distributed systems. *IEEE Transactions on Systems, Man, and Cybernetics,* SMC-II(I):81-96, January 1981.

[Lesser and Corkill, 1983] Victor R. Lesser and Daniel D. Corkill. The distributed vehicle monitoring testbed. *AI Magazine,* 4(3):63-109, Fall 1983.

[Lesser *et al,* 1975] V. R. Lesser, R. D. Fennell, L. D. Erman, and D. R. Reddy. Organization of the Hearsay-II speech understanding system. *IEEE Transactions on Acoustics, Speech, and Signal Processing,* ASSP-23:II-23, February 1975.

[Nii *et al.,* 1989] H. Penny Nii, Nelleke Aiello, and James Rice. Experiments on Cage and Poligon: Measuring the performance of parallel blackboard systems. In M. N. Huhns and L. Gasser, editors, *Distributed Artificial Intelligence, Vol. II.* Morgan Kaufman Publishers, Inc., 1989.

[Rice *et al,* 1989] James Rice, Nelleke Aiello, and H. Penny Nii. See how they run... the architecture and performance of two concurrent blackboard systems. In V. Jagannathan, Rajendra Dodhiawala, and Lawrence S. Baum, editors, *Blackboard Architectures and Applications.* Academic Press, 1989.

[Rice, 1989] James Rice. The Advanced Architectures Project. Technical report KSL-88-71, Knowledge Systems Laboratory, Stanford University, 1989.