Control Heuristics for Scheduling in a Parallel Blackboard System

Keith Decker, Alan Garvey, Marty Humphrey and Victor Lesser [1]
Department of Computer Science
University of Massachusetts
Amherst, MA 01003
Phone: (413)545-3444
Fax: (413)545-1249
Email: DECKER@CS.UMASS.EDU

Keywords: *Blackboard Systems, Parallelism, Control Knowledge*

March 11, 1993

Abstract

This paper investigates the effects of parallelism on blackboard system scheduling heuristics. A parallel blackboard system is described that allows multiple knowledge source instantiations (KSIs) to execute in parallel using a shared-memory blackboard approach. New classes of control knowledge are defined that order the agenda by using information about the relationships between the goals of the KSIs. This control knowledge is implemented and tested in the DVMT application on a Sequent multiprocessor using BB1-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 improved system performance.

[1] 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. The content of the information does not necessarily reflect the position or the policy of the Government, and no official endorsement should be inferred.

1 Introduction

From the beginning the blackboard paradigm has been developed with parallelism in mind [21]. The concept of independent Knowledge Sources (KSs) that communicate only through a shared blackboard is a model that inherently encourages parallel execution.

There are many alternatives for parallelizing a blackboard system at the knowledge source execution level. Corkill [4] analyzes three (see Figure 1): a distributed blackboard approach with multiple Knowledge Source Instantiation (KSI) queues and multiple blackboards, a blackboard server approach with multiple KSI queues and a single blackboard located at one processor, and a shared memory approach, with all processors sharing a single blackboard and a single KSI execution queue.

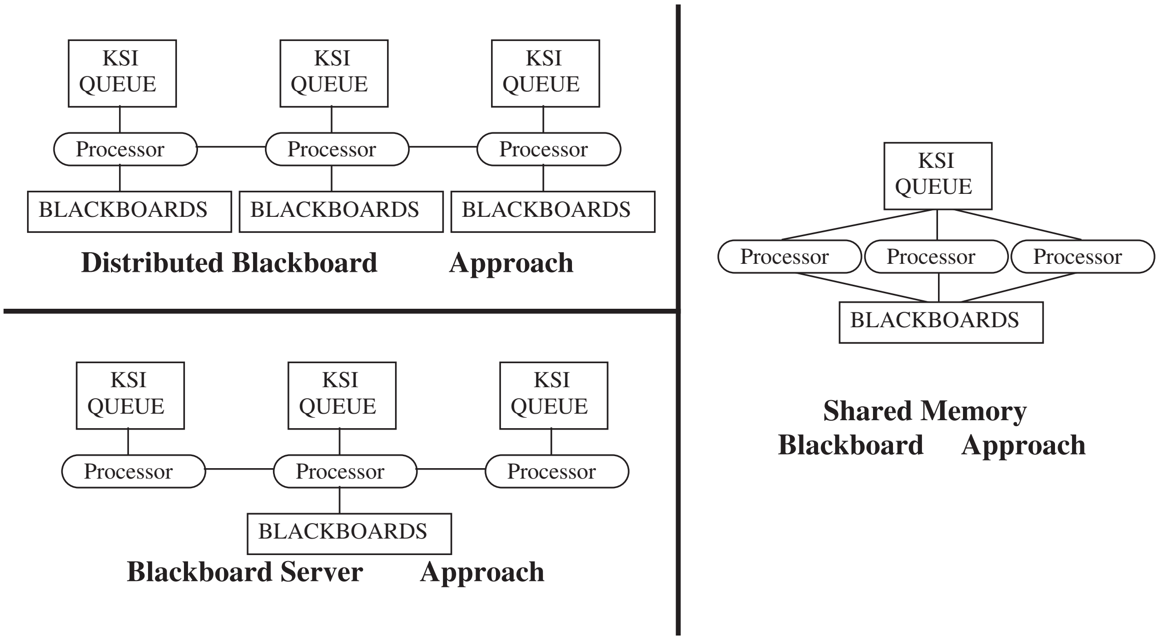

Figure 1: Three Design Alternatives for KS-level Parallelism

The focus of this work is knowledge source parallelism using the shared memory blackboard approach, which has been discussed in several places in the literature. Early work by Fennell and Lesser [16] studied the effects of parallelism on the Hearsay II speech understanding system [15]. One major contribution of that project is a detailed study of blackboard locking mechanisms. More recently, the CAGE architecture has been developed that takes the existing AGE blackboard architecture and extends it to execute concurrently at several different granularities, including that of knowledge source parallelism [25]. Another similar parallel blackboard system was built by Velthuijsen, *et al.* [27].

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 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 [2]. Useful parallel programming environments now exist, including parallel 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 specially-created version of GBB 2.0[2] (Generic BlackBoard) that supports parallelism.

Along with our actual use of parallel hardware, a major difference between our work and previous research is our focus–while previous research was directed at merely obtaining parallelism in the blackboard model, we are interested in creating and *controlling* a blackboard system that has multiple knowledge sources executing in parallel. The essential problem is: which KSI should a processor select when it next becomes free, given that (1) knowledge sources naturally interact with each other through shared data structures and (2) knowledge sources have ordering constraints among them that suggest that certain pairs or groups should not be executed in parallel. The goal of this research is to define and use control knowledge that improves system performance. As we show, it is not enough to simply parallelize a sequential blackboard system by adding locks to the appropriate data structures, and control the system by making each processor select the first KSI on the queue as ordered by the control heuristics of the previously-uniprocessor system, irrespective of which KSIs are executing on the other processors. New control heuristics need to be added for effective parallelism.

Early work on Partial Global Planning[13] showed that constructing schedules using a high-level view of the solution space (derived by distributed agents from task relationships) improved the utilization and effectiveness of distributed processors. This leads to the intuition that task relationships may be helpful for scheduling in a single agent, parallel-processing environment.

In focusing on the control of a blackboard system, we address the larger issue of how to control parallel search in an environment that consists of multiple, interrelated subproblems, where not all subproblems must be completed in order for the system to terminate. The completion of one subproblem could change the requirements for the successful completion of subsequent subproblems. Knowledge about the dynamic, projected utility of a subproblem, along with relationships among subproblems, can be used to order the execution of these subproblems to achieve effective parallelism. The desired effect of a control component in a parallel search is thus not discussed in terms of processor utilization but rather in terms of the processors' contributions to system termination. A coordinated scheduling of activities on processors should result in an efficient, effective search.

The next section discusses the details of our parallel architecture. Section 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 4 briefly introduces the Distributed Vehicle Monitoring Testbed, the application that motivates this work, and describes the new heuristics that were added for parallelism. In this section we also present and discuss 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.

[1] Top Level Common Lisp is a trademark of Top Level, Inc.
[2] GBB 2.0 is a trademark of Blackboard Technologies, Inc.

2 Architecture

This section presents the basic architecture of our uniprocessor system in Section 2.1. In Section 2.2 we discuss how the uniprocessor architecture was made into a multiprocessor architecture. Section 2.3 presents details of the blackboard locking mechanisms.

2.1 Uniprocessor Architecture

The low-level control loop of our uniprocessor blackboard architecture is depicted in Figure 2 and described in more detail in [7, 11]. In this architecture the processor selects the currently

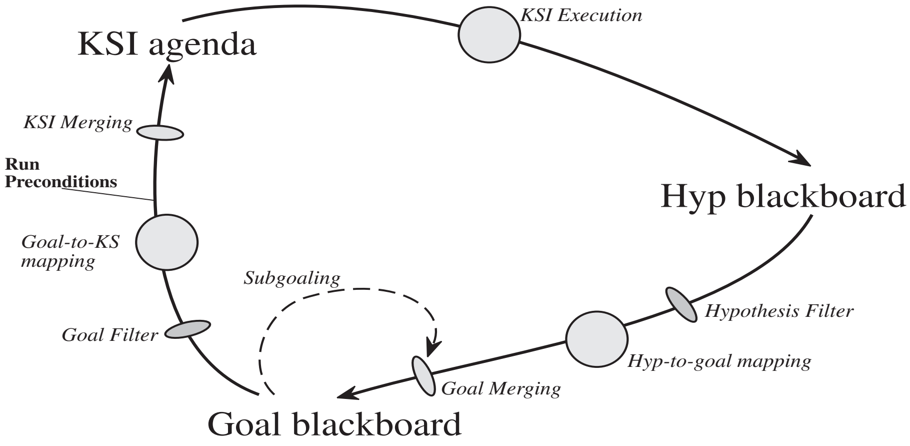

Figure 2: The Parameterized Low-level Control Loop

top rated KSI from the executable agenda and executes the KSI action, which creates and modifies blackboard hypotheses. These hypotheses stimulate goals, which trigger new KSs for execution. A scheduler orders the executable agenda and the loop begins again. Within the low-level control loop, there is a hypothesis filter that is used to select which hypotheses will be used to create goals via the hypothesis-to-goal mapping. Similarly, there is a goal filter to control which goals will be used to trigger new KSs. Each step of this low-level control loop is highly parameterized, which allows for a variety of problem solving strategies. For example, the hypothesis filter can be set to pass only a few hypotheses for a very restricted, expectation-driven style of processing, or it can allow passage of all hypotheses.

In our system, problem solving consists of a number of different styles, or phases. A phase is defined as a period of domain processing in which no major changes to the low-level control loop parameters have been made. At the end of one phase, a phase change occurs when the normally dormant control KSs activate to complete the processing of the phase, establish the initial work of the next phase, and set the parameter values in accordance with the style of the next phase. Often in a phase, there is potential work, in the form of unexecuted KSIs, filtered goals, filtered hypotheses, etc., that is not performed because it is inappropriate for that phase. The initial work of the next phase is generally created from this potential work—unexecuted KSIs are selected for the initial agenda, certain filtered hypotheses are passed through a new hypothesis-to-goal mapping to create goals which create new KSIs, etc. As an example of a phase and a phase change, a typical first phase is to process all of the data roughly to obtain a high-level view of the data (if all of the data is present before the system runs, in contrast to a real-time interpretation situation). When all of the data is coarsely interpreted, control KSs execute to determine which data is more important and should be the focus of the next phase. The control KSs then set the parameters of the low-level control loop accordingly and create the initial work of the next phase by re-passing certain low-level input data back through the hypothesis-to-goal mapping, generating a different set of goals than those generated as part of the coarse processing. These goals generate new KSIs as well. The next phase commences with the execution of the highest-rated KSI on the new agenda.

The values of the parameters of the low-level control loop are set by BB1-style control knowledge sources[18]. The control KSs can conceptually run asynchronously with the domain processing—it could exist as a separate process running concurrently with the domain processing, or it could be activated at a particular point in the low-level control loop. For simplicity, we make the latter choice. We choose to schedule and run the control KSs immediately before rating and selecting the next domain KS for execution. By doing so, there is minimal latency between the establishment of a new problem solving style and KSI-selection based on the new criteria.

2.2 Multiprocessor Architecture

Our parallel implementation of this blackboard architecture uses a shared memory approach in which several processors are concurrently and asynchronously executing the low-level control loop (Figure 3). As part of this control loop, each processor is responsible for selecting and executing domain KSIs. The agenda, the blackboard, and the values of the parameters of the low-level control loop are the only shared data structures. This basic architecture is similar to the architecture of the Cage simulation that utilized KS-level parallelism and asynchronous control [26].

In our implementation of the meta-control for the application described later, there was generally only one control knowledge source on the agenda at any one time. Due to this fact and the difficulty of redoing the underlying implementation of the BB1-style control architecture to permit concurrent execution, it was decided that there was not much to be gained from running control KSs in parallel. However, in order not to cause this sequential meta-control to be a major bottleneck to processing, we did explore *intra*-KS parallelism in the control KSs in order to speed up the meta-control. That is, certain control KSs were re-written to divide up their work into sections that could be executed in parallel with each other. As will be discussed in future research, we believe that in more sophisticated meta-control applications, there is potential for extensive parallelism.

2.3 Blackboard Locking

Various schemes for blackboard locking appear in the literature. The most detailed is that presented by Fennell and Lesser, who describe a method for locking blackboards that assures data integrity [16]. Their method provides two kinds of mechanisms for accomplishing this: various locking mechanisms that provide exclusive access to blackboard objects and regions, and a data-tagging facility that allows processors to state their assumptions about data values and receive messages when those assumptions are violated. When a KS is triggered, all hypotheses

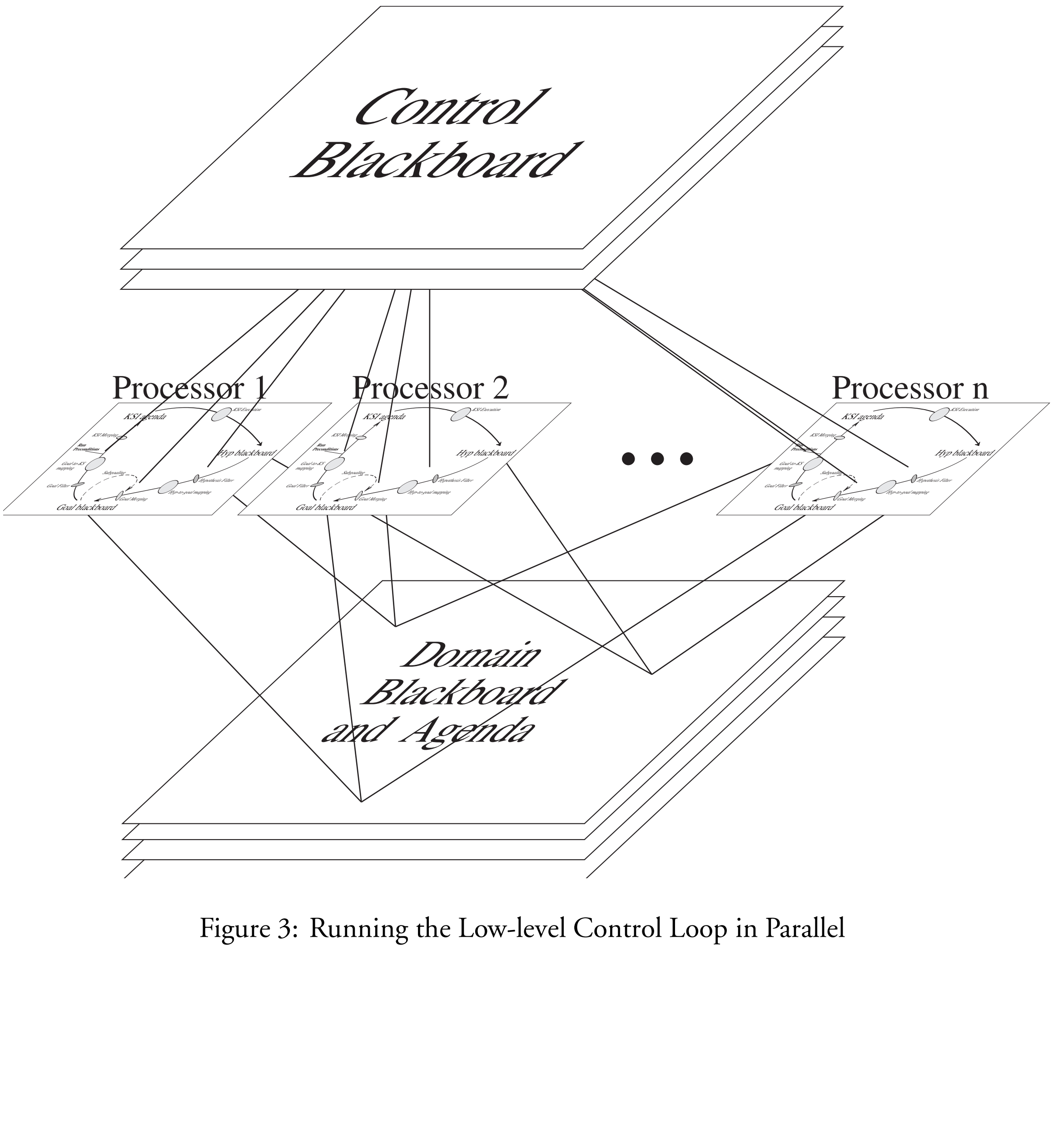

Figure 3: Running the Low-level Control Loop in Parallel

that contributed to the triggering are either locked or tagged. The regions[3] of the blackboard that the KSI expects to output to are also locked. This assures that a KSI will have exclusive access to the parts of the blackboard that it needs in order to execute. An alternative method for enforcing data consistency is the use of transactions as described by Ensor and Gabbe [14].

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. Thus, no KSI will have to wait for very long. 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 [22]. 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 [2].

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.

Two locks control access to the list of KSIs pending execution (the agenda), and to the list of KSIs that have finished execution. Access is controlled to the list of hypotheses that did not make it through the hypothesis filter. These hypotheses can be refiltered when the hypotheses filter changes, typically between phases. Access to the list of filtered goals is similarly controlled.

3 New Classes of Control Knowledge for Parallelism

Control knowledge in a sequential environment rates a KSI based on knowledge such as the belief of its input data, the potential belief in the output data, the significance of the output data given the current system goals, and the knowledge source's 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 KSIs in parallel. These general classes of control knowledge include, but are not limited to:

[3] A region is a part of a blackboard level, such that objects within a region are similar along a particular dimension or attribute.

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 may have to wait for the other to relinquish blackboard locks.

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

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

KSI Invalidation: This is based on the "Competition Principle" in Hearsay-II [19]: the results of some KSIs may completely remove the need to execute other KSIs. A KSI should not be selected for immediate execution if it will be obviated by the successful completion of a currently-executing KSI. For example, KSI A and KSI B may perform the same operation, and produce the same result, in different ways. If KSI A has been scheduled, then KSI B should not be immediately scheduled, because it will be obviated if KSI A completes successfully.

These classes of control knowledge can be obtained from an analysis of the domain KSIs. For example, avoiding *access collisions* requires knowledge about the input/output characteristics of a KSI (i.e., what parts of the blackboard it accesses and modifies.) *KSI ordering* requires knowledge about KSI interactions. Often this knowledge is best captured through relationships among the goals of particular KSIs [5, 24]. Avoiding *KSI bottlenecking* requires knowledge about the probable outcomes of KSIs, again often expressed through goal relationships. *KSI 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 KSIs [9]:

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. Examples of domain relations include inhibits, cancels, constrains, facilitates, causes, enables, and supergoal/subgoal (from which many useful graph relations can be computed, as shown below). These relations provide *KSI ordering* constraints, represented by temporal relations on the goals (see below).

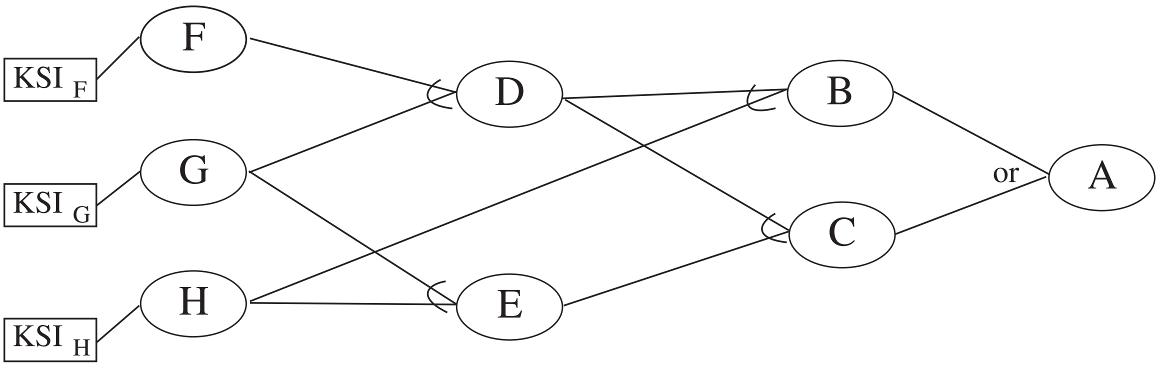

Figure 4: An abstracted goal relation graph

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 *KSI invalidation* constraints. These relations also produce *KSI bottlenecking* information.

Temporal Relations: From Allen [1], 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 KSIs that both use a single exclusive resource cannot execute in parallel. For example, if two KSIs require that a single sensor be aimed or tuned differently, they cannot execute in parallel.

For example, examine the goal structure in Figure 4 (abstracted from an actual domain goal relation graph). The arcs in the graph represent the *goal/subgoal* domain relation on the goals[4]. 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*[5] one another, as do D and H. Goal B *competes* with C.

Assume the following system state when a processor becomes free to select its next KSI: KSI_F is executing on a different processor, and KSI_G and KSI_H are available for execution. A *KSI invalidation* heuristic would avoid scheduling a KSI that achieves goal B in parallel with one that achieves goal C. A *KSI bottleneck* heuristic would prefer to schedule a KSI to satisfy G, which will allow work on goals D and H in the future, over a KSI to satisfy H, which would allow only work on goal G in the future. Of course, KSIs may accomplish multiple goals, a fact that is simplified in this example. A *KSI ordering* heuristic would not find any temporal relations in this example; they are induced by domain relations where goals *constrain* or *facilitate* others.

[4]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.

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

4 Experiments

Experiments were performed in the Distributed Vehicle Monitoring Testbed (DVMT)[23], a knowledge-based signal interpretation system. The input to the DVMT is a representation of the sounds made by a set of vehicles moving through a two-dimensional space. The goal of the DVMT is to identify, locate, and track patterns of vehicles. The four 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).

There are two classes of DVMT domain KSs: 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 time-location positions of the vehicle. The control KSs can also be divided into two main classes: those that implement phases (and phase changes) and those that were specifically added to exploit parallelism.

For the experiments described in this paper, the input data consisted of four possible vehicle types with some signals and groups of signals shared by multiple vehicles. One type of primary pattern and two types of secondary patterns are defined. Twelve vehicles are included, from which there are seven possible instances of the primary pattern class. This is a relatively large data set for the DVMT.

For these experiments, the DVMT processes the input data in three phases. In the first phase (*find initial vehicles*), the DVMT performs a thorough data-directed analysis of all data at time 1 to identify the class and initial position of all vehicles that will be tracked in the experiment[6]. When the processing of the initial data is complete, control KSs trigger and execute the first phase change. Since the major work of a phase change is to select which potential work from the previous phase should be taken into the new phase, this phase change is relatively short, because most work generated in this phase was performed as part of its detailed, complete processing. In the second phase (*approximate short tracks*), the DVMT performs quick, approximate processing to determine the likely tracks and patterns of the vehicles. Blackboard level hopping is used to approximate vehicle level data directly from signal level data. This will result in conflicting interpretations for some of the data. 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. We defined the necessary vehicle explanation as containing at least four time-location points–thus, this phase ended after processing the time 4 data. At this point, control KSs execute to assign new values to system parameters and select work from the remaining potential work of the phase. In the third phase (*perform pattern-directed processing*), the DVMT devotes most of its processing 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.

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

Table 1: Results of Uniprocessor System

	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: Results of Five-Processor System without Parallel Heuristics

4.1 Examining the Basic Parallel Architecture

The first set of experiments were conducted to collect 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 combination of 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 (absolute and percent of total) the single processor uses in each phase and phase change. 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. (In each of the parallel experiments, each processor spent less than 2% of its time in locks.) 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 KSIs were swapped off a processor).

Table 2 shows the speedup resulting from 5 processors and no parallel heuristics. The overall speedup of 3.8 affirms the intuition behind the design of our parallel architecture. By running the low-level control loop 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 [26]. The most important numbers in the table are the speedups in the phases (the speedups in the phase changes are included only for completeness). Phase 1 showed a speedup of 3.6, primarily due to data parallelism. That is, part of this phase is to accept all of the data from the sensors, which is from time 1 through time 9. The data from each time can be processed in parallel, with the minimal amount of blocking from the other processors. The speedup in phase 1 was also due to the exhaustive nature of the time 1 data processing—most KSIs that were generated had to be executed. This was also the case in phase 2 (most KSIs that were generated in phase 2 also had to be executed). In fact, phase 2 showed the highest amount of inherent parallelism, because it was tightly controlled—phase 2 consisted of working on each of the tracks in a well-defined process, and none of the tracks interacted with each other. Phase 3 was the most complicated phase, because of many track interactions. Note however that phase 3 still showed a speedup of 3.9.

4.2 Examining the Parallel Heuristics

By simply allowing KSIs to run in parallel, we achieved a significant improvement in the DVMT performance. However, from the discussion in Section 3 we should be able to do better than just taking the top (single processor) rated KSI off of the agenda. Four new scheduling heuristics were added to incorporate knowledge of parallelism. In our system, a BB1-style controller [18] rates each KSI with a set of active heuristics. In the parallel system, we defined 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. All the previous non-parallel domain heuristics were numeric but some of the new parallel heuristics are pass/fail.

Pass Non-obviated Outputs. Don't schedule a KSI that is expected to produce the same results as a currently-executing KSI, because, if the currently-executing KSI terminates successfully, there is no reason to run the KSI in question (it will be obviated). This heuristic implements the *KSI invalidation* criteria described in Section 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 KSIs available that will not be obviated by existing KSIs, 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.

Pass Primary Patterns. The single-processor DVMT has a heuristic that rates primary pattern KSIs numerically higher than secondary or unknown pattern KSIs (a KSI can be classified as such through an analysis of its goals)[7]. In the parallel DVMT, a numeric version of this heuristic would not necessarily produce the desired effects. When primary pattern KSIs are being executed by one or more processors and there are no primary pattern KSIs on the agenda, a processor using the numeric heuristic would select a secondary pattern KSI. Because primary pattern KSIs usually generate more primary pattern KSIs, a better behavior from the processor would be to wait until the completion of all executing primary pattern KSIs before resorting to executing a secondary pattern KSI. While the numeric heuristic would result in the single processor system only running primary pattern KSIs, only a pass/fail heuristic could achieve this in a parallel system. This heuristic is an example of a *KSI ordering* heuristic as described in Section 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.

[7] We did not avoid the creation of secondary pattern KSIs in either the single or multiprocessor case because in the future we may wish to run the system, for instance, in a less time-constrained situation. In such a situation, the system would prefer to execute primary pattern KSIs, though it would execute secondary pattern KSIs if it had time.

	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

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 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 [16] 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.

Prefer Many Output Hypotheses. Schedule KSIs that expect to produce many output hypotheses before those that expect to produce fewer output hypotheses. This heuristic implements the *KSI bottleneck* avoidance class of heuristics described above. By preferring to produce many outputs, more possible KSIs 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 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.

To test these heuristics, we ran a 5 processor system with the parallel scheduling heuristics. Figure 5 shows how the KSI queue length varied over time, and how the utilization of processors (with respect to domain processing and the low level control loop, not meta-control) varied with time. Phase 1 consists of three distinct sub-phases. In the first, data for all time points are input in parallel. The agenda slowly shrinks until eventually it increased sharply as KSIs associated with the time 1 data are generated. In the second, KSIs for time 1 are removed and executed (possibly spawning more KSIs for time 1) until all there are no more KSIs on the queue. In the third, processor utilization decreases to zero, at which point the control KSs change phase. During the phase changes, utilization is zero (utilization here is defined as in terms of domain work, not control work) as the agenda grows. Phase two consists of processing a relatively large initial agenda (36 KSIs) and KSIs spawned from the initial agenda until the agenda is empty. In the second phase change, the agenda grows to a high number (61). Finally, in phase 3 the agenda size remains relatively large.

Table 3 is a comparison of the system with the 5 processors and the four heuristics with the 1 processor and 5 processor systems without the four heuristics. While 2.0% and 1.1% speedup in phase 1 and phase 2, respectively, is insignificant speedup, it was not unexpected after running the system with 5 processors and no parallel heuristics. The primary cause for the lack of any significant speedup in these phases is that the parallel version without heuristics was developed to run as fast and as efficiently as possible, irrespective of our intent to perform

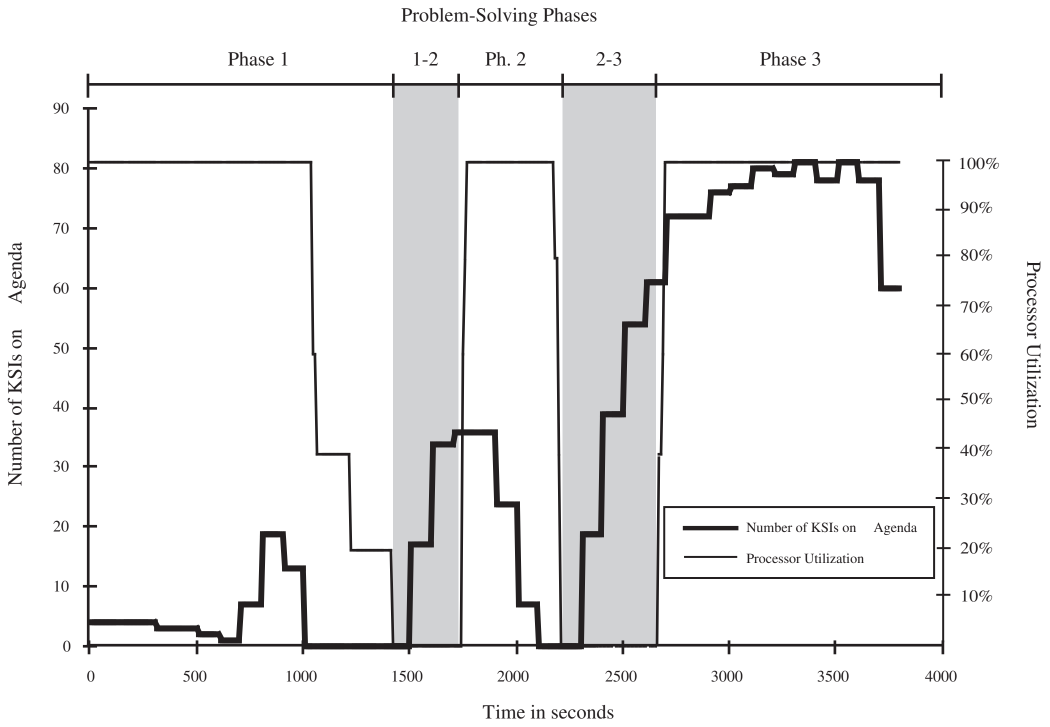

Figure 5: KSI Queue Length and Processor Utilization vs. Time: 5 processors with heuristics.

parallel scheduling experiments. The phases were defined as single-processor control plans, and as such, the 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, because the heuristics are designed to order or otherwise prune a large search.

For example, the KSI obviation heuristic finds very few KSIs to obviate. This is because we try to identify and filter out or merge hypotheses and goals that might create redundant KSIs 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 characteristics (such as precision and certainty) for time [11]. This may result in more potentially obviatable KSIs on the agenda. We test this hypothesis in Section 4.2.1.

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 4.2.2. Another problem stems from the *prefer many output hypotheses* 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 generally 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 (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 productive, relative to the system goals.

Figure 6 shows the KSI queue length and processor utilization for a 10 processor run, and

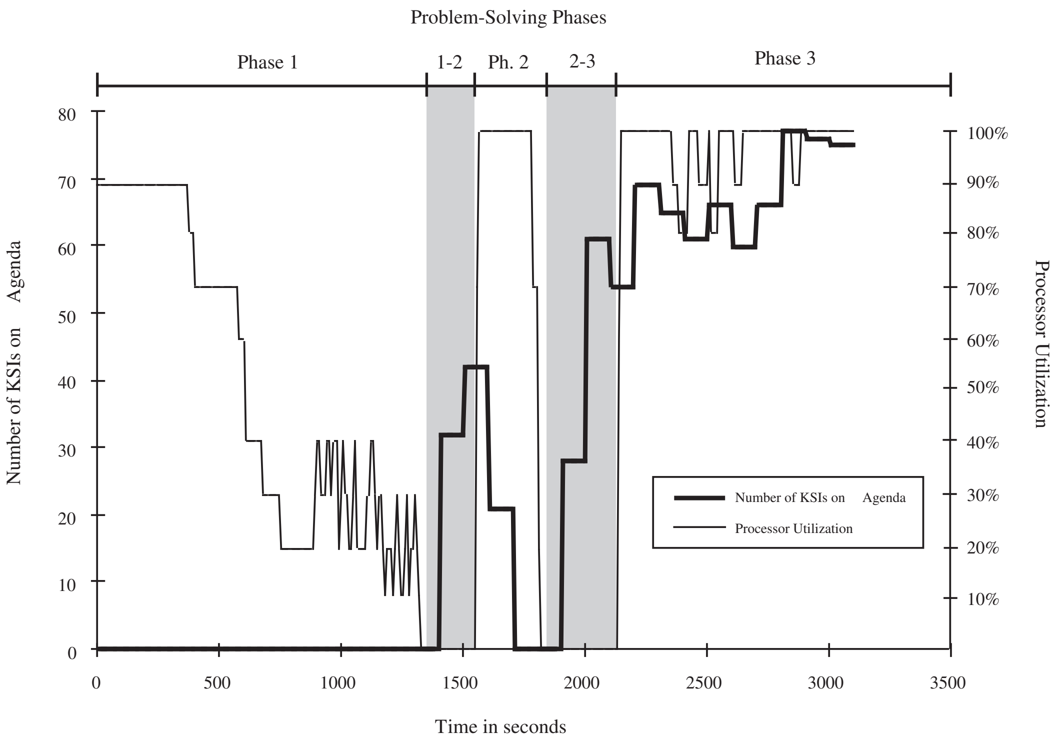

Figure 6: KSI Queue Length and Processor Utilization vs. Time: 10 processors with heuristics.

Table 4 shows a comparison of timing information against the single processor case. Phase 1

	1	1–2	2	2–3	3	Tot
Time	1330	217	275	301	1074	3197
S–up	3.9	4.9	7.7	6.0	5.3	5.0

Table 4: Summary of results with 10 procs, parallel heuristics

achieved a speedup of only 3.9 because (as seen in Figure 6) processor utilization was below 40% for about half of the phase–there was no available work to perform. Phase 2 was a speedup of 7.7–processor utilization was near 100% for most of the phase, with processors involved with KSIs that were necessary for the completion of the phase. Phase 3 is the most interesting phase, because utilization was high (it briefly dropped to 80% twice), but speedup was only

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

5.3. This was because frequently KSIs were being grabbed from the agenda as soon as they arrived. This is unlike the uniprocessor system, where they would remain on the agenda long enough for later-arriving KSIs to merge into them. In the multiprocessor system, this often resulted in similar, but nonobviated, KSIs being executed at the same time.

Because our basic parallel architecture showed an impressive speedup without the parallel heuristics, the effects of the parallel heuristics outside of phase 3 were not significant. To test the merit of the parallel heuristics, we decided to reconfigure our system, removing certain properties that we believed were causing the parallel heuristics to appear unimportant. While these modifications do not make sense as stand-alone changes (i.e., we would probably not run our system this way), they are effective in simulating a blackboard system environment that does not contain the properties in question. It is in these environments that we are testing the usefulness of the parallel heuristics. Section 4.2.1 tests the KSI obviation heuristic, and Section 4.2.2 tests the access collision heuristic.

4.2.1 Testing the KSI Obviation Heuristic

To test the *KSI obviation* parallel heuristic, we disabled 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[8]. The scenario was run with five processors without any parallel heuristics, and then with the addition of the single KSI obviation heuristic. The results are shown in Table 5. Speedup was achieved particularly in phase two, when the addition of the KSI obviation heuristic caused a significant number of KSIs to be obviated. This is because KSIs were executed in a data-directed manner and were apt to obviate other data-directed KSIs upon completion. (Different low-level data often merges into the same, high-level, abstracted data structure.)

4.2.2 Testing the Access Collision Heuristic

To test the *access collision* parallel heuristic, we configured the DVMT to allow higher potential contention for system locks, without otherwise handicapping the system. This was implemented by two modifications: disabling the *KSI-merging* feature and artificially lengthening the time processors spend in the blackboard bucket locks. The first modification forces the

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

creation of a separate KSI for each output goal (rather than merging with a KSI that had similar goals). Since KSIs that perform similar activities tend to get rated approximately the same, this modification increases the probability that a processor will select a KSI for execution that will create results that one or more other KSIs running at the same time should produce. The effect of not allowing similar KSIs to merge thus is increased 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 without any parallel heuristics, and then with the addition of the single access collision heuristic. Both runs were with 5 processors. Adding the heuristic to avoid regions that other processors are utilizing resulted in a significantly decreased 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%. While these particular numbers are not important (these percentages are related to the added, artificial time spent after obtaining a lock), with the heuristic a processor spent about 40% less time in locks.

5 Conclusions/Future Directions

In investigating control heuristics for scheduling in a parallel blackboard system, we have shown significant speedup with a shared memory multiprocessor version of the DVMT application that executes KSIs in parallel with each other. Most of this parallelism was due to our choice of architecture–we have avoided synchronization and under-utilization by making each processor run the low-level control loop as part of its basic execution cycle. Because of the specifics of the application (i.e., its highly deterministic and efficient control), a comparison of the system with and without parallel heuristics did not conclusively show the utility of the parallel heuristics in all phases of problem solving. However, two heuristics were shown to be useful in separate tests on a modified system.

One source of poor performance was the synchronization caused by system-wide phases (and phase changes). At the end of a phase, performance was hindered when many processors were waiting for one or two processors to finish executing their KSIs. This low-performance situation is even more prominent when only one processor is executing a KSI, and this KSI spawns another KSI, which spawns another, etc. This chain of KSIs theoretically can only be executed by one processor. Obviously, making the other processors remain idle is wasteful.

Even in the uniprocessor version, system-wide phase changes limit the robustness of the DVMT application. Whenever data (vehicles) arrive (are first detected) at different times, the system cannot process all data in system-wide phases. That is, the first phase is to process all the time 1 data in order to establish the initial identity and location of all vehicles. But what if the vehicle does not appear until time 5? In this case, instead of processing all of the data in a phase, we would like to establish *channels* for each vehicle (or, more abstractly, groups of data, regions in the XY plane, etc.) [12]. Channels are created as vehicles appear and allow different data to be in different phases of problem solving simultaneously. A channelized architecture permits real-time problem solving, which often requires being able to separate and work on certain KSIs to the exclusion of others. Unimportant channels can be ignored if deemed necessary. A channelized version version of the uniprocessor DVMT was implemented and used for experiments in real-time problem solving [6, 17].

Our intent is to eventually create a parallel version of this channelized architecture in which one or more processors are devoted to each channel. The allocation of processors to channels would be dynamic and controlled according to the system goals. Channels would permit investigations into both data parallelism (vehicle being tracked in parallel) and search parallelism (e.g., two channels that interact could be allocated processors in a coordinated manner, and the same data could be interpreted by two different problem solving styles if there were sufficient processors available). Of course, in a channelized architecture, our approach to a parallel implementation of the meta-control would be changed since meta-control would be performed on a channel-by-channel basis.

Although a channelized architecture presents the potential for added parallelism, a channelized architecture is of little value if the parallel control heuristics are ineffective. We believe the limited effect of the parallel control heuristics as a whole was because the system performed very little search. In each of the three phases, we exercised tight control over the potential work (hypotheses, goals, and KSIs) that was generated. We could have artificially modified the system to make it perform more search, but we believed that this would have resulted in an illogical configuration for the DVMT. We could not justify the applicability and generality of results gathered from such a system. It is our expectation that, as the amount of search and interaction among search paths increases, the heuristics will become more important.

In addition to exercising tight control over the potential work generated, another reason for the limited effect of the parallel control heuristics is the lack of a high-level understanding of how each KSI relates to the termination of a problem solving phase. The system does not possess an explicit representation of the connection between the individual KSIs and the goal of the phase, so often a processor could not find a valid reason for *not* running a KSI. That is, a processor often ran a KSI, not understanding that the KSI would probably not contribute to the overall system goals. The only knowledge of this type that exists in the system is at the lowest level–scheduling is essentially at the micro-level, in that the scheduling primarily attempts to determine the best *order* of execution of the available KSIs. The scheduling process is, in general, not trying to determine the overall importance of each KSI to system termination.

One of the reasons why the scheduler could not ascertain the importance of a KSI to the termination of a phase is that the criteria for termination was not explicit and capable of being reasoned about. Though this is most apparent in the final phase, it is also apparent in the termination of phases 1 and 2. In each phase, termination procedures were triggered when a certain event occurred. Unfortunately, there was no scheduling process that reasoned about how to *cause* this event—processing simply proceeded according to some implicit, high-level criteria without looking at explicit termination criteria. Without a fundamental representation of termination criteria (and thus what each phase was attempting to achieve), nor a fundamental understanding of how each KSI is expected to contribute to this termination, the system made ill-advised scheduling choices. Without this more knowledgeable scheduler, the parallel heuristics could not be more fully exploited.

The RESUN[3] architecture for interpretation problems possesses this high-level representation. Termination criteria are explicit, as is a complete representation of how the primitive actions impact the termination criteria. We believe future work in a parallel version of this architecture will provide valuable insights on a parallel scheduling theory.

Finally, we are attempting to formalize and generalize the results from these studies. The results discussed in this paper are important though limited by a single-instance experimental methodology. We are developing an abstract task generator capable of capturing all the interactions of KSIs in any domain [10]. We have also worked on quantifying these interactions (called *coordination relationships*) for distributed and parallel processing[8]. Work developing a parallel scheduling theory in this abstract environment has begun [20] with the hopes of providing general results with wider applicability.

Acknowledgments

We thank Kevin Q. Gallagher for his work in creating a shared memory parallel processing version of GBB 2.0.

References

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

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

[3] Norman Carver and Victor Lesser. A new framework for sensor interpretation: Planning to resolve sources of uncertainty. In *Proceedings of the Ninth National Conference on Artificial Intelligence*, pages 724–731, August 1991.

[4] 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.

[5] Daniel D. Corkill, Victor R. Lesser, and Eva Hudlická. Unifying data-directed and goal-directed control: An example and experiments. In *Proceedings of the National Conference on Artificial Intelligence*, pages 143–147, Pittsburgh, Pennsylvania, August 1982.

[6] Keith S. Decker, Alan J. Garvey, Marty A. Humphrey, and Victor R. Lesser. A real-time control architecture for an approximate processing blackboard system. *International Journal of Pattern Recognition and Artificial Intelligence*, 7(2), 1993.

[7] 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.

[8] Keith S. Decker and Victor R. Lesser. Analyzing a quantitative coordination relationship. COINS Technical Report 91–83, University of Massachusetts, November 1991. To appear in the journal *Group Decision and Negotiation*, 1993.

[9] Keith S. Decker and Victor R. Lesser. Generalizing the partial global planning algorithm. *International Journal of Intelligent and Cooperative Information Systems*, 1(2), June 1992.

[10] Keith S. Decker and Victor R. Lesser. Quantitative modeling of complex computational task environments. In *Proceedings of the Eleventh National Conference on Artificial Intelligence*, Washington, July 1993.

[11] Keith S. Decker, Victor R. Lesser, and Robert C. Whitehair. Extending a blackboard architecture for approximate processing. *The Journal of Real-Time Systems*, 2(1/2):47–79, 1990.

[12] Rajendra T. Dodhiawala, N. S. Sridharan, and Cynthia Pickering. A real-time blackboard architecture. In V. Jagannathan, Rajendra Dodhiawala, and Lawrence S. Baum, editors, *Blackboard Architectures and Applications*, pages 219–237. Academic Press, Inc., 1989.

[13] E.H. Durfee and V.R. Lesser. Partial global planning: A coordination framework for distributed hypothesis formation. *IEEE Transactions on Systems, Man, and Cybernetics*, 21(5):1167–1183, September 1991.

[14] 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.

[15] 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.

[16] 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):98–111, February 1977.

[17] Alan Garvey and Victor Lesser. Design-to-time real-time scheduling. *IEEE Transactions on Systems, Man, and Cybernetics*, 23(6), 1993. Special Issue on Scheduling, Planning, and Control.

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

[19] 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.

[20] Marty A. Humphrey and Victor R. Lesser. Parallel scheduling based on abstract task models. Working Paper, Cooperative Distributed Problem Solving Laboratory, 1992.

[21] 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-23:11–23, February 1975.

[22] Victor R. Lesser and Daniel D. Corkill. Functionally accurate, cooperative distributed systems. *IEEE Transactions on Systems, Man, and Cybernetics*, SMC-11(1):81–96, January 1981.

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

[24] Victor R. Lesser, Daniel D. Corkill, Robert C. Whitehair, and John A. Hernandez. Focus of control through goal relationships. In *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence*, Detroit, August 1989.

[25] 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.

[26] 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.

[27] H. Velthuijsen, B.J. Lippolt, and J. C. Vonk. A parallel blackboard system for robot control. In *Proceedings of the Tenth International Joint Conference on Artificial Intelligence*, pages 1157–1159, August 1987.