Design Alternatives for Parallel and Distributed Blackboard Systems

Daniel D. Corkill

August 1988
COINS Technical Report 88-38

To be presented at the Second AAAI Workshop on Blackboard Systems
St. Paul, Minnesota, August 1988.

Design Alternatives for Parallel and Distributed Blackboard Systems

Daniel D. Corkill

Department of Computer and Information Science
University of Massachusetts
Amherst, Massachusetts 01003

Abstract

Since its inception, the blackboard paradigm has been viewed as particularly appropriate for parallel and distributed hardware architectures. Yet, the paradigm's multiprocessing potential remains largely untapped. The availability of multiprocessing hardware and languages in conjunction with tools for building blackboard applications is sparking renewed interest in multiprocessing blackboard architectures. Multiprocessing can be introduced at a number of levels in the blackboard architecture, from low-level parallel coding techniques to concurrent execution of knowledge sources and control components. This paper details the issues associated with these multiprocessing levels and presents three high-level design alternatives for parallel and distributed blackboard architectures. Each design is appropriate for a particular architectural and application environment, and we discuss the selection criteria for each design and issues associated with its implementation. The three designs are being implemented as extensions to the Generic Blackboard Development System (GBB).

1 Introduction

The blackboard paradigm was developed with an eye toward parallel execution of knowledge sources (KSs). One of the initial design goals for the Hearsay-II speech understanding architecture was suitability for multiprocessor execution [1]. The cooperating KS model introduced in Hearsay-II is conceptually a parallel/distributed model. Yet, the promise of multiprocessing and distributed blackboard architectures remains largely untapped.

This research was sponsored in part by donations from Texas Instruments, Inc., by the National Science Foundation under CER Grant DCR-8500332, and by the Office of Naval Research, under a University Research Initiative Grant (Contract N00014-86-K-0764).

Why the delay? One reason is that constructing effective uniprocessing blackboard-based AI applications has been "hard-enough." The potential performance gains have been outweighed by the additional programming complexity involved with multiprocessing. In addition, a single processor has been "fast enough" for many blackboard-based applications. A second reason for the delay has been the lack of stable hardware and language facilities for multiprocessing exploration.

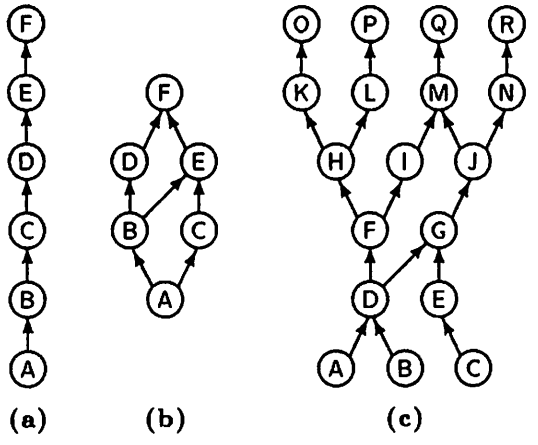
This situation is changing. More ambitious applications are being considered for blackboard approaches. Stock multiprocessors and distributed networks are proliferating, and language support for parallel and distributed computation (including Lisp) is becoming available. Simultaneously, tools for building blackboard-based applications are being developed. By providing multiprocessing support within these blackboard-building tools, investment in parallel/distributed blackboard approaches can be made more attractive. In particular, the design alternatives described in this paper are being implemented as part of the Generic Blackboard Development System (GBB) [2].

Parallel and distributed activities can be introduced at various levels within the blackboard paradigm. Low-bandwidth, physically distributed networks are best suited to a large-grained, loosely-coupled problem solving structure. On a shared-memory multiprocessor, however, reasonable decompositions include the entire range of large- to small-grained computations. As an extreme example, it is possible to take a uniprocess blackboard-based AI application written in Lisp, and simply run it on a multiprocessor using a parallel Lisp interpreter that transparently executes Lisp faster.[1]

We will not consider such "language transparent" parallelism in this paper.[2] Here, we are primarily

[1] Although simple for the application builder, there may not be significant parallelism in such an approach.

[2] Similarly, we will not consider software for implement-



Although developed independently, KSs have implicit dataflow relationships among themselves that determine the amount of concurrency available in the KS structure. A linearly-ordered ("pipelined") KS structure can only support data concurrency (a). If the ordering contains incomparable KSs, additional concurrency is possible among the incomparable KSs (b). Typically, the KS structure is dependent upon the blackboard data and may contain multiple minimal and maximal elements (c). In such structures a number of KS paths are possible, and not all paths need be initiated or completed.

Figure 1: Linearly- and Partially-ordered KS Structures.

concerned with the application-level issues facing explicit, concurrent KS execution. Because parallel and distributed blackboard approaches both involve concurrent KS activity, they share issues of synchronizing blackboard access and KS processing. In particular, we focus on the importance of *semantic synchronization* of blackboard data and how to achieve it or tolerate the lack of it. We then describe three basic design approaches for parallel and distributed blackboard applications. Each design is appropriate for a particular architectural and application environment, and we discuss the selection criteria associated with each design as well as the issues associated with its implementation.

2 Approaches to Blackboard Multiprocessing

Multiprocessing can be introduced into the blackboard paradigm at various levels:

1. **Blackboard interaction machinery can be implemented using parallelism.** By exploiting parallelism in the implementation of basic blackboard interaction operations (object creation, deletion, retrieval, and modification) the apparent speed of these operations can be increased. With the exception of complex retrieval operations, there is limited potential for significant speed increases with this approach.

2. **Each KS can be implemented as parallel processes.** This approach involves exploiting whatever opportunities for parallelism exist in the coding of the KS. The potential for speed increases is heavily dependent on the particulars of the KS. From the perspective of the blackboard paradigm (which views KSs as "black boxes"), KSs with internal parallelism are merely "faster" KSs. Issues of parallelism remain internal to the KS.

3. **Multiple KSIs can execute concurrently.**[3] This approach captures the conceptual parallelism of the cooperating KS problem solving model. Typically, KSs are relatively large computations, and this approach results in relatively large-grained parallelism. The potential for speed increases with this approach depends on the relationship among the KSs. A "pipeline" KS structure (Figure 1 a), where each KS is a consumer of the results of the preceding KS, will only support data parallelism. Less constrained KS structures, however, present the potential for parallelism in processing common data (Figure 1 b,c). In practice, KS parallelism on common data is hard and the synchronization issues are non-trivial. Issues of synchronizing concurrent KSI executions form a major part of the remainder of this paper.

4. **The blackboard control component can execute concurrently with domain KSIs.** Control remains an important aspect of multiprocessing blackboard applications. Opportunistic scheduling is still relevant when the number of possible activities exceeds the number of available processors. In most applications it will remain uneconomical to provide enough processors to be able to immediately execute all tasks. Yet, the sensitivity of problem solving to individual control decisions may decrease as the number

ing parallel/distributed approaches on particular hardware. Thus issues addressed by such systems as Agora [3,4] and MACE [5] are not directly discussed in this paper.

[3] We will consider the concurrent execution of KS *instances*. KSI concurrency subsumes concurrent execution of different KSs and further includes the possibility of simultaneous execution of different *instances* of the same KS. This means that KSs must not only worry about interference from different KSs, but that KSs must be written to be *reentrant*. Casual coding practices that are acceptable in a serial environment must be restricted in a concurrent KSI environment.

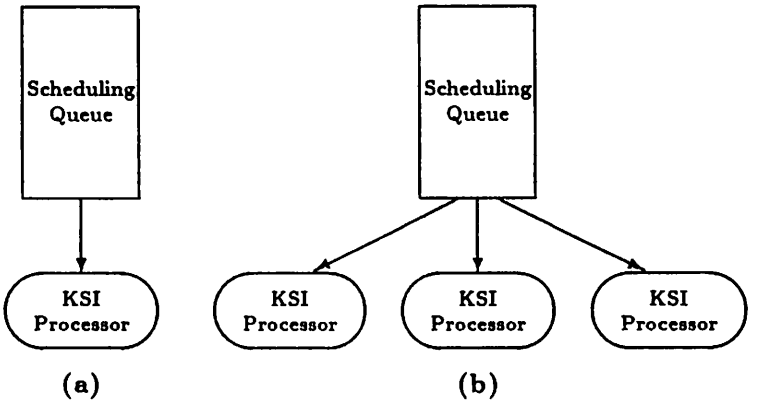


Figure 2: Single- and Multithread KSI Execution Models.

of available processors increases. This may provide even more opportunity for decoupling control and problem-solving activities, as slightly out-of-date control decisions may be acceptable [6]. Finally, if major control decisions are made using control KSs, much of the effort (and problems) of concurrent control and domain processing involve the concurrent execution of KSIs (option 3).

These options are mutually compatible, and all four can be applied as appropriate in a given architectural and application environment. A developer designing a multiprocessing application must choose an appropriate combination of options. If the developer is free to choose an underlying hardware platform, the freedom of choices makes the design decision even more difficult. Until a body of empirical data is accumulated, these decisions will often be based on intuition.

Combining options can create additional issues. For example, when combining internal KS parallelism and concurrent KSI execution (options 2 and 3), process scheduling is no longer internal to each KS. Simply multiplexing execution among all the active small-grained processes is a solution, but one that suppresses opportunistic scheduling. The processes associated with a particular KSI are executing as a group to achieve the purposes of that KSI, and an opportunistic scheduler must be cognizant of this relationship.

From the perspective of the Generic Blackboard Development System (GBB), providing these four options on diverse hardware platforms requires supporting a range of capabilities. We are constructing parallel and distributed GBB implementations for architectures ranging from loosely-coupled networks of heterogeneous machines to shared-memory multiprocessors running a shared-heap multiprocessing Common Lisp implementation.[4] This perspective has resulted in a unified view of a range of parallel/distributed implementation options, and the remainder of this paper presents this view and the conceptual and implementation issues we are addressing.

In particular, we believe that concurrent KSI execution (option 3) is likely to be selected (in concert with other options) for most applications, due to the natural concurrency available in the cooperating KS model. Before describing our designs, we consider the complexities that arise when KSIs execute concurrently.

3 Executing KSIs Concurrently

Concurrent execution of KSIs can be introduced into the the blackboard paradigm by replacing a single-thread KSI execution model with a multithreaded one. As an example of a single-thread execution model, consider the "simple" scheduler used in the Hearsay-II speech understanding system [7]. This scheduler assumes a single scheduling queue containing KSI tasks and a single processor to execute those tasks (Figure 2 a).

The KS execution cycle begins by selecting a queued KSI for execution. As the KSI executes, it makes changes to the blackboard (called *blackboard events*). KS are *triggered* in response to particular blackboard events by the *blackboard monitor*, which

[4] Our shared-heap implementation efforts are being developed using the UMass Parallel Implementation of Common Lisp (PICL) system, a shared-heap Common Lisp multiprocessing system (supporting futures) for Sequent Symmetry multiprocessors.

knows which classes of blackboard events interest which KSs. The occurrence of a blackboard event does not guarantee that there is sufficient information on the blackboard for a KSI to be executed. To investigate further, the blackboard monitor places a *precondition procedure* for each triggered KS on the scheduling queue. These precondition procedures make a more detailed examination of the blackboard to determine whether or not a KSI should be created in response to the blackboard event. If sufficient information is found, a KSI record is created and placed onto the *scheduling queue* based on a *priority rating* (typically a function of the state of the blackboard, the overall state of problem solving, and the analysis by the precondition procedure). When the currently executing *task* (either a KSI or precondition procedure) completes, the highest-rated task is removed from the scheduling queue for execution. This cycle continues until the problem is solved or until no executable tasks remain on the scheduling queue.

This Hearsay-II single-thread model can be converted to a multithread model by providing more than one processor to service the scheduling queue (Figure 2 b). When any processor completes execution of a task, it selects the highest-rated task from the queue for execution. Each processor would also act as the blackboard monitor for its KSI executions, instantiating KSIs and preconditions on the shared scheduling queue.[5]

Once multiple KSIs are executing concurrently, it is possible that they both may want to modify the same blackboard object (or the scheduling queue). We will consider synchronizing access to the blackboard shortly, but the possibility that a KSI execution can become blocked awaiting access to data must been addressed. Clearly we would like to postpone continued execution of the blocked KSI and use the processor to begin executing another KSI.

Although single-threaded execution model used in the Hearsay-II speech understanding system could support such multiplexed KSI execution, the complexity of writing interruptable KSs led to a non-interruptable scheduling model in the uniprocessor implementation. Once the execution of a task was initiated, it ran to completion before the next task was initiated. This greatly simplified the machinery required for scheduling tasks and avoided problems of KSI interference arising from interleaved activities. Because of the reduced complexity, this non-interruptable approach to scheduling has been used in most blackboard systems. However, by making tasks non-interruptable, opportunistic changes based upon the arrival of new data could not be made until the currently executing task completed.[6]

Multiplexing among queued tasks is another possible scheduling approach in a uniprocessor blackboard system, one that would directly support interrupted task execution. Tasks could be allocated time based on their priority, or tasks could be activated/deactivated based upon the current control focus. A major difficulty with this approach is the potential for changes to the blackboard made by other tasks while the current task is paused. Once the task is resumed, calculations performed prior to the interruption can be invalid due to the blackboard changes. Two "simple" approaches to handling this situation are:

1. detecting and redoing any corrupted computations;
2. proceeding by ignoring the changes in hope of later KSIs (triggered in response to the changes) appropriately updating the blackboard.

The first approach requires either monitoring for changes in the data being used by the paused KSI or having each KSI recheck its input data whenever it is reactivated. If changes are made that invalidate the computations, either the computations must be redone or the KSI can be terminated to be replaced by other KSIs triggered by the blackboard changes. When highly-interacting KSIs are multiplexed, restarting KSI computations can result in little forward progress; almost every time a computation is paused, it is invalidated by another KSI and must be redone.

The second approach requires that the knowledge and blackboard representation be capable of converging to a solution even if computations are made using temporarily incomplete data; KSI output must be always subject to revision by later KSIs. This approach is the basis for Lesser and Corkill's notion of *functionally accurate, cooperative* (FA/C) problem solving [8]. Although this approach has been primarily applied to distributed systems, it is relevant to parallel and even multiplexed implementations.[7]

To maintain any semblance of blackboard consistency, both approaches require performing all black-

[5] Due to the relatively large size of KSI computations, extracting KSIs (and preconditions) from a common queue is not likely to become a serious bottleneck. Maintaining an appropriately ordered queue, however, is an issue. The solution is dependent on the control approach used in the application.

[6] In practice, this delay is only a major problem in time-critical applications.

[7] In fact, Fennell and Lesser's simulation efforts with an early version of Hearsay-II provided evidence of the FA/C approach's potential in a parallel processing setting [9]. Their observations led to the elaboration of the FA/C approach.

board modifications during one or more *modification phases*, during which the KSI cannot be interrupted. Thus, there remain potential delays in pausing an executing task.

In general, dealing with multiplexed KSI executions in a uniprocessor environment has not been worth the effort.[8] However, if KSIs are allowed to execute concurrently in a parallel or distributed environment, the same context modifications that arise with paused KSIs are present. In fact, these context interactions are much more difficult to handle than system-level memory contention and interference. In the next section, we consider such *semantic interference* on the blackboard and what can be done about it.

4 Synchronizing Concurrent KSIs

If concurrency is entirely within sequentially executing KSIs (options 1 or 2), potential blackboard interactions can be staticly predicted and controlled. Concurrent execution of KSIs, however, complicates the handling of blackboard interactions. Arbitrary instantiations of independently developed KSs executing concurrently require dynamic synchronization techniques.

Fennell and Lesser performed a detailed simulation study of blackboard synchronization patterns by using an early version of the Hearsay-II speech understanding system [9], and Fennell's dissertation details the synchronization mechanisms they developed [10]. At the most primitive level, access to database resources (reading and writing the blackboard database) must be synchronized to avoid race conditions. For example, suppose two executing KSIs want to examine, test, and possibly modify the same slot value in an existing blackboard object. Neither process can be allowed to modify the value between the examine and test actions of the other process. The solution to such *system-level* synchronization issues is standard multiprocessing fare: the slot is treated as a resource which must be acquired before examining the slot and which is released after the modification is performed.

[8] The problem of queue *latency* in the serial execution model is directly related to the interaction issues associated with paused task execution. The scheduling priorities of KSIs awaiting execution on the scheduling queue are potentially effected by changes to the blackboard made by executing KSIs. Queue latency has been addressed by ignoring it, by recalculating all priority ratings at the end of each KSI execution, by recalculating all ratings "every so often," and by recording the objects and blackboard areas used to determine the priority and retriggering the calculation if changes are made to them.

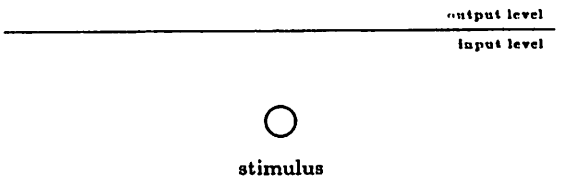


Figure 3: Synthesis Actions (Initial State).

KSs complicate the simple system-level synchronization because their computations require:

1. examining dynamically determined blackboard objects and values;
2. significant time to complete.

Consider a simple synthesis KS that is triggered by the creation of a *stimulus hypothesis* (Figure 3). The KS attempts to find other *supporting hypotheses* on the blackboard that, in conjunction with the stimulus, support the creation of a higher-level *output hypothesis*. To perform this activity the KS might execute the following steps:

1. Using the attributes of the stimulus hypothesis, determine the allowable range of attributes for each of the potential supporting hypotheses. Each such range forms an *input context* for the KSI (Figure 4).
2. Retrieve from the blackboard all hypotheses contained within the input context (Figure 5).
3. Analyze each "legal" combination of stimulus and support hypotheses for computability. For those with sufficient computability, compute the attributes (including belief) of an output hypothesis to be created based on those supports (Figure 6).
4. For each output hypothesis, check the blackboard to see if a hypothesis that is semantically "identical" to the output hypothesis already exists (Figure 7).[9]
5. If so, appropriately update the attributes of the existing hypothesis (Figure 8).
6. If an existing hypothesis does not exist, create a new hypothesis using the output hypothesis values (Figure 9).

[9] A desirable property in many applications is that "equivalent" blackboard objects are not created. Instead, the new object's values are *merged* with those of the existing object. Therefore, when creating a new blackboard object, a KS must first look for an existing object.

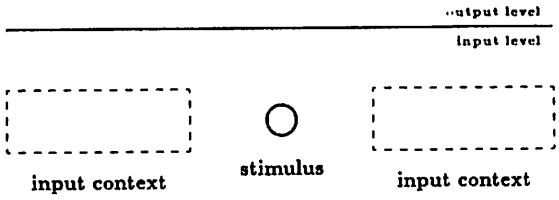


Figure 4: Synthesis Actions (Step 1).

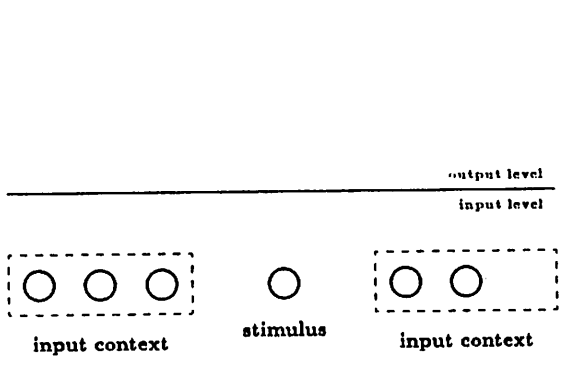


Figure 5: Synthesis Actions (Step 2).

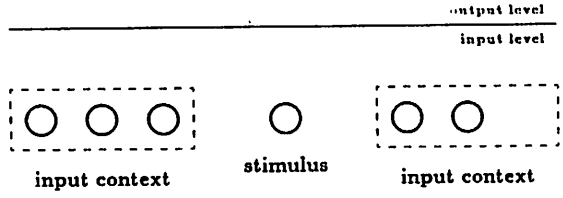


Figure 6: Synthesis Actions (Step 3).

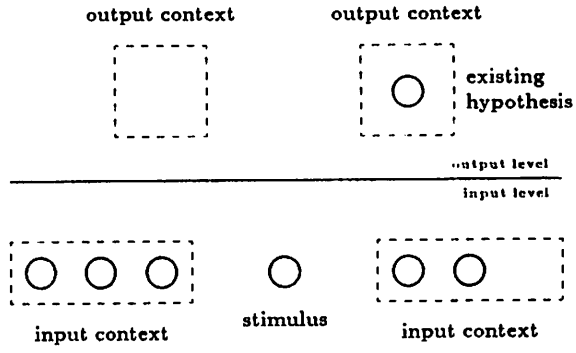


Figure 7: Synthesis Actions (Step 4).

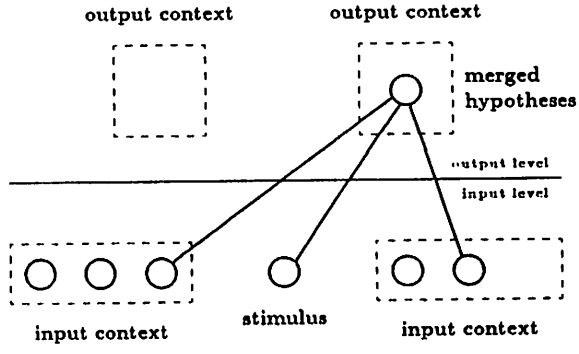


Figure 8: Synthesis Actions (Step 5).

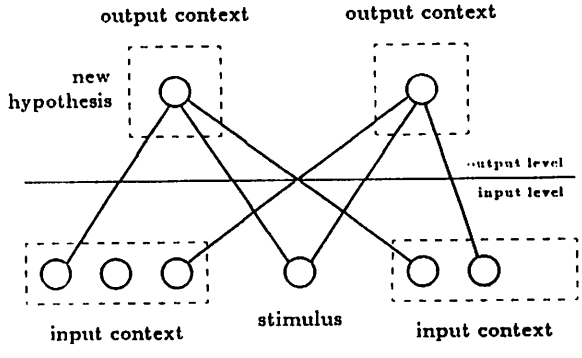


Figure 9: Synthesis Actions (Step 6).

Considerable time may pass between the time the KSI finds input objects on the blackboard (step 2) and the time the KSI writes the resulting output objects (steps 5 & 6). What if another executing KSI modifies an input context (either by changing the attributes an existing object or by creating a new object within the input context) during this time interval? Similarly, what if another KSI creates a hypothesis within an output context during steps 3–6? Simple system-level synchronization of slot access does not address such *semantic* race conditions. However, these potential interactions must be taken into account.[10]

One means of providing semantic synchronization is to postpone the execution of KSIs that might semantically interact with another executing KSI. As noted by Fennell and Lesser, avoiding interference purely through intelligent scheduling is unreasonable and likely to seriously reduce parallel activity. Avoiding interaction purely by scheduling is in conflict with the KS independence touted for blackboard systems. If the control component is to schedule activity to reduce blackboard interference, it must be aware of the existence and blackboard interaction patterns of every KS (or at least every precondition procedure). When the behavior of KSs is highly dependent on the data on which they operate, predicting the potential interactions is difficult, leading to over-conservative scheduling.

Another approach is to structure the blackboard and the problem solving so that blackboard objects are never modified. Instead new versions are created to reflect changes.[11] This "dataflow" style of problem solving can be appropriate in certain situations, but in general memory and processing costs can make it an inappropriate choice.

Yet another approach is to treat the entire computation as an atomic operation. Doing so requires treating existing blackboard objects and arbitrary regions of the blackboard (the input and output contexts) as allocated resources. Locking mechanisms for blackboard objects and for blackboard regions were implemented in Fennell and Lesser's Hearsay-II multiprocessing simulation. We summarize their mechanisms here.

Object locking is straightforward. Each blackboard object is locked for either exclusive access or for read-only access. Access to the slots of an object requires obtaining access rights to the entire object. Locking at the level of an entire object rather than the individual slots of an object provides a reasonable balance between the cost of acquiring access to the blackboard and the granularity of locking machinery. Since objects were the unit of creation on the blackboard, and since multiple slots of the same object tend to be accessed as a group, this level of granularity appears appropriate.

The need for region locking arises out of several situations. First, if a blackboard object is to be placed on the blackboard, the blackboard itself needs to be treated as a resource. (There may not be any existing objects within the blackboard region to lock.) Obviously, the entire blackboard or even a single blackboard level is an inappropriately large resource for allocation. In the Hearsay-II speech understanding system, Fennell and Lesser used the utterance time metric in conjunction with the blackboard level to define a two-dimensional ($level \times time$) coordinate space for region locking. A region could be defined as a level value and a range of utterance time values ($time_{min}$ .. $time_{max}$). As with object locking, exclusive and read-only access was available.

An important issue when combining object and region locking, is avoiding deadlock. Since object and region locks both apply to the blackboard, their locking mechanisms must be coordinated. Fennell and Lesser's simulation avoided deadlock by using a non-preemptive, linearly ordered lock acquisition strategy that encompassed both objects and regions. A non-preemptive strategy is important, as any premature forced release of a blackboard resource could invalidate the computations of the releasing process.

To implement a linearly-ordered lock acquisition strategy, the Hearsay-II simulation required generating a total ordering of all blackboard objects and another of all blackboard regions. Blackboard objects were given a unique identifier implementing this ordering that was generated when the unit was created.[12]

Regions posed a more difficult ordering problem. Because regions can overlap arbitrarily, the actual region locks were implemented using a fixed set of primitive region locks. Each region was mapped onto the set of primitive regions that overlapped the desired region, and the lock request was treated as if those primitive regions had been specified. Thus, the number of primitive regions as well as the registration of lock requests onto these primitive regions are important influences on the granularity of region locking.

The node and region orderings were integrated by insuring that node locking did not interfere with any

[10] Semantic interference from concurrent KSI executions is present whether or not parallelism is used to implement the KS itself.

[11] Agora's *write-once* objects are an example of this approach [4].

[12] GBB requires that units within a class be uniquely named. By ordering the classes and using the name within the class, a similar total ordering can be established.

region lock and by insuring that region locking did not interfere with any node lock.

The locking approach used by Fennell and Lesser is highly-compatible with GBB and we are implementing their basic approach in each of our parallel/distributed frameworks. GBB's notion of space dimensionality[13] is particularly suited to region locking, providing a direct metric for specifying blackboard regions. For example, a space used for tracking aircraft might contain five dimensions: *x*, *y*, *z*, *time*, and *classification*. Region locking allows precise, five-dimensional regions within the space to be locked for exclusive or read-only access. However, providing unit and region locking mechanisms does not prescribe how best to use the machinery in a particular application. In the remainder of this section, we continue looking at the synchronization issues associated with concurrent KSI execution.

Returning to the simple synthesis KS model, we could use the locking machinery to implement the entire KSI execution as an atomic action:

1. Lock the stimulus hypothesis.
2. Using the stimulus hypothesis, determine the input contexts and lock them using region locking.
3. Retrieve from the blackboard all hypotheses contained within the input contexts.
4. Compute the output contexts and lock them using region locking.
5. Perform merging and hypothesis creation, as appropriate.
6. Release the region locks.

This approach is subject to deadlock in a nonpreemptive strategy because the locks used in steps 1, 2, and 4 are not acquired as a unit.

One way out of the deadlock situation is to have the KS attempt to determine the potential output contexts directly from the stimulus. This allows locks for both the input and output contexts to be acquired at once. Since the KS must insure that its region locks cover all possible contingencies, the locks may cover significantly more of the blackboard than is required, increasing the possibility of overlapping locks held by executing KSIs. Also note that the stimulus hypothesis is unprotected until the locks are acquired. It is possible that another KSI could modify the stimulus during this interval.

A different approach is to give up treating the entire KSI execution as atomic, opting for atomic pieces of the entire KSI execution. For example, we could reimplement the simple synthesis KS as:

1. Using the stimulus hypothesis, determine the input contexts.
2. Reverify that the stimulus hypothesis has not been changed, if it has repeat step 1.
3. Lock the stimulus hypothesis and region lock the input contexts.
4. Retrieve from the blackboard all hypotheses contained within the input contexts and copy their relevant values to a local context.
5. Unlock the stimulus hypothesis and input contexts.
6. Compute the output contexts.
7. Lock the output contexts and all supporting hypotheses.
8. Reverify that none of the supporting hypotheses have changed. If they have, restart at step 2.
9. Perform merging and hypothesis creation, as appropriate.
10. Release the locks.

This approach releases all locks before a new set is required, thus avoiding deadlock. Each KSI essentially voluntarily preempts its use of input locks. Reverifying that data has not changed requires additional effort. Either all data values have to be reinspected and compared with saved copies, or *tags* have to be associated with the values indicating that the KSI is to be informed if any tagged value is changed by another KSI.[14] In addition, the need to repeat computations can lead to repeated recalculations if interactions are severe.

One means of reducing the granularity of recomputation is to further divide the KSI into smaller atomic pieces. An example of this approach is the *transactional blackboard* approach of Ensor and Gabbe [11]. Although their work was concerned with system-level synchronization of blackboard modifications in a distributed system, a transactional approach can be applied to primitive KSI pieces. If an input context associated with a transaction is modified, the transaction is aborted and restarted.

Instead of the executing KSI redoing its computations when unprotected blackboard objects are modified, it is possible to allow *subsequent* KSIs to perform the computations using the new values. This approach is at the heart of the FA/C problem solving approach. In this case, no attempt is

[13] GBB's primitive blackboard "levels" are termed *spaces*.

[14] Such a tagging scheme was used in the Hearsay-II simulation study.

made to verify the input state. The KSI proceeds to write output hypotheses based on its original input data. The assumption behind this approach is that other KSIs will be stimulated by the blackboard modifications that invalidated the data of the executing KSI, and that those KSIs will rectify any inconsistencies. The developing solution will tend to be "out of synch," but if KSIs continue converging the blackboard database faster than new data (or high-level goals) are arriving, the solution will eventually become consistent.

This approach does not require any detection of invalidated data or of input contexts. If transient, semantically "identical" hypotheses can be tolerated, even output context locking can be eliminated. All the FA/C approach requires is system-level consistency. The appeal of this approach stems from the possibility that reperformed calculations are roughly equivalent to those performed using a more rigorous approach, so that the elimination of semantic synchronization overhead results in a win for the FA/C approach.

Although the FA/C approach appears attractive, with the sole exception of Fennell and Lesser's simulation studies using Hearsay-II, use of the FA/C approach in a parallel environment remains speculative. Lesser, *el. al.*, have reported favorable results with the FA/C approach in distributed settings [12,13].

These discussions have focused on the problem of semantic consistency of blackboard operations. The queue latency problem has not be addressed. The ratings of pending KSIs is likely to be based on blackboard object values. If these change, the ratings become inconsistent. Obviously, the application would quickly deadlock if all KSIs pending execution locked the contexts used to compute their priority ratings. Again, recomputation (either at regular intervals or via tagging) is one approach. The FA/C-style of allowing inconsistent ratings is another extreme. The appropriate choice here appears heavily dependent on both the application and the particular control scheme being used.

In summary, system-level synchronization is the easy part of concurrent KSI execution. On shared-memory machines, it can be performed using available language and hardware facilities. On distributed networks, it is performed by treating messages as atomic actions. Semantic synchronization is the difficult problem facing parallel and distributed blackboard implementations, and it is a problem that must be addressed no matter what underlying architecture or blackboard design configuration is used.

The appropriate approach to semantic synchronization is highly application dependent, and the solutions range from attempting to prevent all forms of semantic interference through locking to ignoring (and later repairing) the damage caused by semantic interference after it occurs.

5 Three Design Alternatives for Parallel and Distributed Blackboard Architectures

In this section we present three different design alternatives for parallel and distributed blackboard architectures. These designs are based on concurrent execution of KSIs (option 3), yet each can be easily extended to include the other levels of parallelism as well. Each design has different overhead costs and implementation complexities, and each is best suited to a different architectural and application environment. Following that section, we present an overview of how we are implementing each design in GBB and the issues associated with each implementation.

5.1 The Shared-memory Blackboard Approach

The *shared-memory blackboard approach* (SMBB) allows each processor to directly access the blackboard (Figure 10). This approach is the most direct mapping of the cooperating KS paradigm onto multiprocessing hardware. To provide efficient access to the blackboard, the SMBB approach requires a *shared-heap.*[15]

Two different KS-to-processor mappings can be implemented using the SMBB approach: a *functional* mapping and a *computational server* mapping. In the functional mapping, each KS is bound to a specific processor. That processor is responsible for executing all instances of its KS. If more than one instance of a KS needs to be executed, the additional KSIs are queued at that processor until they can be executed. A functional mapping directly models the interactions between a group of experts solving a problem, and it is particularly appropriate when different specialized hardware is required to execute different KSs. The main drawback with a functional mapping arises when processing

[15] The distinction between a *shared-heap* and a *shared-address-space* is important when considering AI applications written in Lisp or similar heap-allocated languages. It is crucial that each processor be able to read and write directly into the shared heap. A shared-address-space with separate heaps requires translating inter-heap pointers.

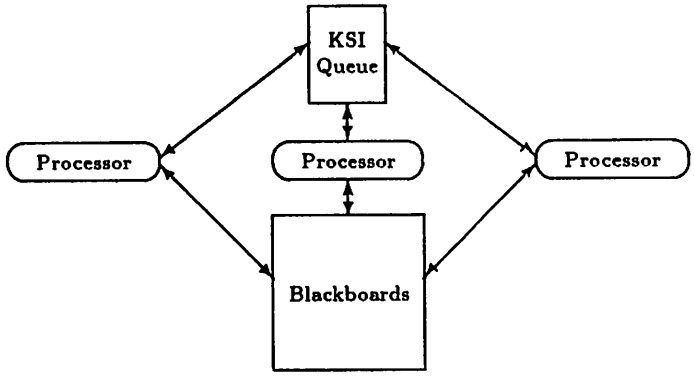


Figure 10: The Shared-memory Blackboard Approach.

demands are not uniform among the KSs. If a number of plausible instances of one KS should be executed, all available processors should be used to execute those KSIs. Instead, the functional mapping results in idle processors and processors working on relatively unimportant KSIs while more important KSIs remain queued at a busy processor.

The computational server KS-to-processor mapping implements the multithreaded KSI scheduling approach discussed previously. Each processor is an execution resource for all pending KSIs. When a processor completes a KSI, it selects the highest-rated instance of any KS from the scheduling queue for execution. The computational server mapping is well-suited to non-uniform demands on the KSs. However, it does require that every processor be capable of executing any KSI. Specializations of the computational resource mapping are also possible. For example, *preferential* scheduling can be used to execute KSIs with special needs on processors with hardware suited to those needs.[16] Due to its flexibility, we will restrict discussion of the GBB implementation of the SMBB approach to the computational server KS-to-processor mapping.

5.2 The Distributed Blackboard Approach

In the *distributed blackboard approach* (DBB), each node[17] has a separate local blackboard containing units created locally as well as copies of units received from other processing nodes (Figure 11). When communication among processing nodes is limited by bandwidth and communication delay, a central blackboard becomes infeasible. Instead, each node is given a local blackboard for its problem solving activities, and "important" changes to these local blackboards are communicated to other nodes for incorporation into their local blackboards. Because each node can directly access its local blackboard, without communication delay or direct interference from another node, this approach is very appropriate for (but not limited to) low-bandwidth distributed network environments.

An important design issue with the DBB approach is deciding what overlap should exist among the blackboards and whether or not complete consistency among any overlapping portions is required. One extreme is to have every node maintain a complete copy of the entire blackboard. The other extreme is to divide responsibility for maintaining the blackboard into non-overlapping areas, assigning responsibility for each area to a particular node. Nodes needing blackboard objects outside their areas of responsibility must receive a copy of the data from the appropriate node. This copy can be obtained by explicitly requesting the information from the other node or by having the other node volunteer the information under the assumption that it might be used by the local node.[18]

In the first extreme, all blackboard objects are available locally. However, a massive amount of communication can be required to transmit all blackboard modifications to all nodes. If complete consistency among the copies is required, the synchronization messages and delays further impede system performance. The latter extreme requires

[16] The functional mapping is, in effect, an extreme preferential specialization of the computational server mapping.

[17] In this paper, "node" is used loosely to indicate a processing and memory resource. A node might be a processor connected to a loosely-coupled network, a loosely-coupled multiprocessor, or a processor on a shared memory machine. As an example, each node in a DBB approach could be a complete SMBB system.

[18] Lesser and Erman present a more detailed taxonomy of DBB characteristics [12].

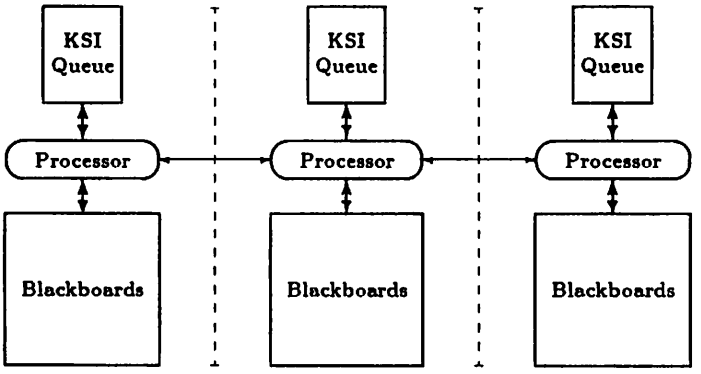


Figure 11: The Distributed Blackboard Approach.

communication only when a node's KSIs need to access non-local portions of the blackboard. In this case internode communication is required to obtain copies of the data. If the copying is performed as needed by the node's KSIs, transmission latency can significantly delay continued processing. Anticipating the need for copying objects avoids this latency, at the potential cost of copying unused objects. Thus allowing KSIs or the control component of a node to prefetch potentially needed objects before they are needed can be important in the DBB approach.

A compromise between these two extremes is to provide some overlap among the individual node's blackboard responsibility areas, but only enough so that the KSIs executing on that node access primarily local information. The degree of overlap is dependent upon the characteristics of the application and the decomposition of problem solving knowledge into KSs. As before, anticipating the need for objects can reduce the processing delays associated with communication latency. Another means of reducing the delay is to have nodes voluntarily communicate the data in anticipation of their use by other nodes. Unless synchronized, these techniques can allow KSs to operate on out-of-date copies. This many not be a problem, since it may be possible to write KSs that can function effectively with objects that are out-of-date with versions present at another node. Several experimental investigations have simulated this type of approach (including experiments with a 3-node Hearsay-II speech understanding system [12] and the Distributed Vehicle Monitoring Testbed [14]).

In any event, knowing which node to ask for needed information or which nodes should receive locally stored information requires meta-level network control information.

5.3 The Blackboard Server Approach

As with the DBB approach, the *blackboard server approach* (BBS) does not allow multiple access paths to the blackboard. Unlike the DBB approach, however, the BBS approach maintains only a single version of each blackboard object. The overall blackboard is partitioned into *blackboard responsibility areas*, each allocated to *blackboard server* nodes. A node can directly access objects within its blackboard responsibility area. Objects outside this area are indirectly accessed via the appropriate blackboard server node. Unlike the DBB approach, no local copies of non-local units are placed on the local blackboard. Instead, a *virtual unit* is created. A virtual unit serves as a placeholder for the actual object, and can be used in blackboard interaction functions as if it were the actual object.

As in the DBB approach, two partitioning extremes are possible. In the first extreme, a single server node is responsible for maintaining the entire blackboard for the other *client* nodes (Figure 12). At the other extreme, each node acts as both a server for a disjoint piece of the overall blackboard and as a client in executing KSI activities.[19] Efficiently and appropriately switching roles is an important issue with uniprocessor nodes performing such dual roles. When multiprocessing nodes are used, multiple processes can be assigned to service incoming client activities.

The DBB approach is best suited to hardware architectures with relatively fast communication, but which lack shared-heap address spaces.[20] It can also be appropriate for more loosely-coupled networks

[19] This is the approach taken in Boeing's *virtual blackboard* framework [15].

[20] Without shared-heap capabilities, shared-memory only provides a fast communication channel.

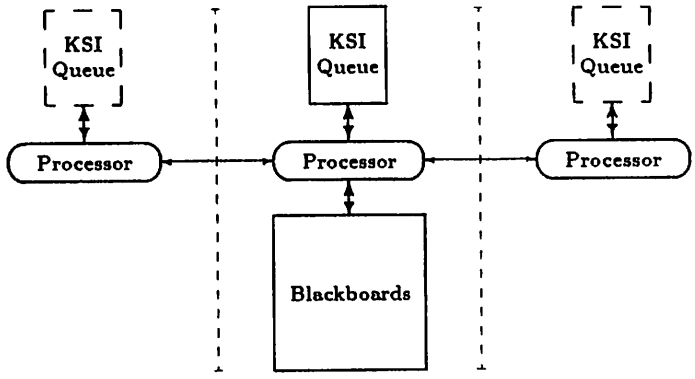


Figure 12: The Blackboard-server Approach.

when:

- the application can be partitioned so that most of the references are to local objects; or
- when the application mandates that all copies of an object be consistent and objects are updated frequently.

As with the DBB approach, anticipating needs and prefetching information is important when significant communication latency exists.

Client nodes communicate all blackboard interaction requests to the appropriate server node, which queues them for service. For example, instead of creating a blackboard object, a client node transmits a creation request (along with all attribute values) to the appropriate server. The server actually creates the object. Similarly, modifying the value of a remote object's slot is performed by the server. Finally, retrieval requests are performed by the server, and the results are communicated back to the client. When the client and server processes do not execute within the same Lisp heap, each interaction requires translating values as they cross node boundaries.

6 The GBB Implementations

In this section, we overview the implementation issues associated with developing each design as a GBB extension.

6.1 The Shared-memory Blackboard Approach

In the SMBB implementation, the control shell is responsible for creating the processes associated with multithread scheduler model. For example, the control shell could be instructed to create 16 KS execution processes and to associate each process with a particular processor. To hide the particulars of process creation from the control shell writer, GBB/SMBB provides the form:

```
(create-process {form}*)
```

which creates a new Common Lisp process and begins executing *forms*. Typically, *forms* invoke a basic scheduling loop in a multiprocessing control shell.

Object locking is provided using the form:[21]

```
(lock-unit unit &KEY (read-access nil))
```

Executing this form does not actually acquire the lock, it merely records that this lock is to be included in the next *lock set* used by the KSI. A lock set is simply a set of lock specifications that are maintained until the form `(acquire-locks)` is evaluated. At this point all unit (and region) locks are ordered and acquired. If a desired lock cannot be obtained, the executing process blocks until that lock is released. Then the lock acquisition process continues. The lock set mechanism provides a convenient way for the KS author to determine the desired locks and then to subsequently acquire these locks in a single action.

Regions locks are specified using the form:

```
(lock-region path region
             &KEY (read-access nil))
```

Again, this form does not directly acquire the lock. *Path* indicates the space(s) on which the region is to apply.[22] *Region* specifies the particular

[21] GBB's blackboard objects are called *units*.

[22] In GBB spaces are specified using a *path* object: a special GBB object encoding the path from the root of a blackboard hierarchy to a particular space or a relative modification of another path.

dimensional region(s) of the space to lock. The *region* specification is identical to the *pattern-object* syntax used in GBB's blackboard search/retrieval function, **find-units**. In particular, regions can be expressed as transformations on existing blackboard objects.

Locks are freed using the form `(release-locks)` which releases all locks held by the executing process.

These constitute the primitive extensions to GBB for developing SMBB applications. GBB/SMBB does not provide any machinery for tagging units or contexts nor does it directly support other synchronization primitives. These can be built as required using GBB's blackboard event machinery and system-level synchronization primitives provided by the underlying multiprocessing Common Lisp implementation.

One major issue with SMBB/GBB implementation is the algorithm used in blackboard search/retrieval. **Find-units** uses a tagging technique to convert $O(n^2)$ operations to $O(n)$ operations [16]. Each GBB unit contains a tag slot that is used by the find algorithm to mark units during the pattern-match and filtering phases of unit retrieval. Maintaining this algorithm using this tag slot requires **find-units** to lock the entire retrieval region throughout its operation. In some applications, the time spent in **find-units** can be significant—even when retrieval operations are tuned for efficiency [17].

We could use $O(n^2)$ operations. These would not require any locking, but would result in increased computational cost. We could also dynamically allocate tags as needed by a process, at the cost of increased storage and computation time to locate the appropriate tag value. In both cases, if KSIs blocked by **find-units** operations result in idle processors, the extra computational cost could still result in increased overall throughput.

Is locking the entire retrieval region reasonable? In many cases the retrieval region is related to the input and output contexts of the KS. If the KS itself is locking these regions, then is no possibility for interference from another executing KSI and we can use the existing algorithm without modification.[23] Each of these choices is appropriate in particular situations. We are pursuing all three until we have sufficient experience to narrow the field.

[23] The actual region required by **find-units** is dependent upon the implementation structure of the blackboard, and is likely to be somewhat larger than the region representing an input or output context. In the worst case, where a space is specified as a list of objects, the entire space must be locked during **find-units**.

Integrating blackboard monitoring and control into the SMBB approach involves determining which processes are responsible for which control activities. Our approach is to have the process causing the event handle the initial duties associated with the event. In GBB, this translates to evaluating any event handlers associated with the particular event.

GBB's control shells define event handlers that perform control activities. These event handlers have to be defined to interact appropriately with a multiprocessing control shell. For example, if a shell implements parallel control and domain processing, the event handlers might simply push the event onto a buffer of events associated with the executing KSI. When the KSI completes, the base scheduler for that processor might insert the buffer onto a control blackboard for processing and create a new buffer for the next KSI execution. The control KSIs would take over from that point.

In summary, the GBB/SMBB implementation provides primitives for implementing a range of problem solving and control options. As experience is gained, the GBB/SMBB implementation will be augmented with higher-level facilities that have proven useful. This is in keeping with the basic philosophy of GBB.

6.2 The Distributed Blackboard Approach

The GBB/DBB implementation involves augmenting GBB with primitive communication capabilities as well as abilities to initialize and terminate remote processing nodes.

The first requirement is invoking and connecting to a GBB process on each node from an initial *user interface* node. DBB/GBB provides a form:

`(initialize-node` *network-name* {*form*}*`)`

which initializes a Common Lisp process on node *network-name* and creates a bidirectional communication link with that node. All initialization *forms* are then evaluated on the remote node. A node object is returned which can be used in subsequent communication requests. Typically, the *forms* are used to invoke an instance of a distributed control shell at each node.

Blackboard units are communicated using the form:

`(send-unit` *unit node*`)`

which sends an ASCII representation of *unit* to *node*. The decision to use an ASCII representation stems from the desire to support loosely-coupled

heterogeneous networks. Point-to-point communication overhead any two nodes can be reduced by replacing the ASCII representation with a specialized encoding (such as a FASL representation).

Send-unit immediately transmits *unit* to *node*, where it is buffered until requested by *node*. This allows the application or control shell writer to introduce new blackboard units at controllable intervals. The form:

```
(process-messages node &KEY (wait nil))
```

flushes the reception buffer of all messages received from *node*, creating whatever units were transmitted. This form also supports a keyword `:WAIT` argument, which if non-`nil` causes the executing process to wait for a message if the buffer is presently empty.

Often, it is desirable to place a transmitted unit on different spaces at the receiving node than the spaces it resided upon at the transmitting node. Such path translation could be performed at either the transmitting or receiving node. Because the receiving node has direct knowledge of the structure of its blackboards, we have opted to perform path translation at the receiving end. The form:

```
(define-path-translation
   node path-specification
   {path}*)
```

adds a new translation for units received from node *node*. Normally communicated units are placed on spaces corresponding to the spaces the unit resided on at the transmitting node. However, if a *path-specification*[24] matches one of the spaces of a received unit, that space is replaced by the spaces represented by the *paths* value. The *paths* value can also be `nil`, meaning not to place the unit on the matched space.

When the value of a slot is modified, that change can be communicated using the form:

```
(send-slot unit slot node)
```

which sends an ASCII representation of the update of slot *slot* in *unit* to *node*. As with unit creation messages, slot update messages remain queued until processed by the receiving node.

GBB's event handling capabilities allow easy implementation of automatic communication protocols. If every created instance of a unit class is to be transmitted to a particular node, a creation event handler for the unit can be written which uses **send-unit** to send the newly created unit to the desired node. Similarly, if updates to particular slots or links are to be automatically communicated, a handler for that slot can be added which uses **send-slot**.

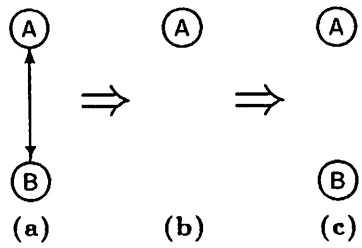


Figure 13: Separate Transmissions Disconnects Linked Units.

A KSI may want to know when a node has processed all the messages it has transmitted. GBB/DBB provides a general mechanism that can be used for this purpose. The form:

```
(send-form form node)
```

sends *form* to *node* for evaluation. *Form* is evaluated in the null environment. As with other messages, evaluation messages remain queued in order of reception until the receiving node processes the messages. Thus, sending a *form* to *node* that generates an acknowledgment message is an easy way to ensure that all messages sent prior to sending *form* have been processed by *node*.

When the typical unit of communication is a GBB unit, recreating the structure of linked units is important. For example, suppose a node contains two linked units A and B (Figure 13 a). If this node sends A to a second node (Figure 13 b) the link information cannot be reproduced, as unit B does not exist at the remote node.[25] If the node later sends B, the receiving node would not know that A and B were originally linked together (Figure 13 c).

Although this behavior might be desired in certain situations, it is reasonable to assume that sending all units to a remote node would recreate the structure present in the local node. GBB provides the expected behavior through the use of *dummy units*. GBB requires that units of the same unit class be uniquely named. In GBB/DBB we extend this requirement to require a unique network name for instances of the same unit class. When a particular unit instance is received, GBB attempts to recreate its unit link structure using existing local units of the same name. For each link referencing

[24] A *path-specification* is the unencoded representation of a path.

[25] GBB insures that unit links are consistent: that they indeed point to an existing unit and that there is a proper inverse link. Without unit B at the remote node, GBB cannot create the link.

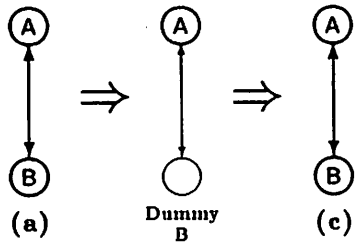


Figure 14: Reconnecting Units Using Dummy Units.

a non-existing unit, GBB creates a special dummy unit and links the transmitted unit to the dummy unit (Figure 14). Dummy units are not placed on any space and are invisible to the GBB's unit retrieval functions. They are visible as link values, however. This property allows transitive communication of units involving links to dummy units to recreate the original structure.[26]

When a transmitted unit is created, a check is first made to see if a dummy unit for that unit already exists at the local node. If so, the created unit replaces the dummy unit and inherits all the dummy unit's link structure.

Although the use of dummy units is important in the GBB/DBB implementation, the machinery for handling them was already in place in GBB. The same link-recreating property is required if partial portions of the blackboard can be saved separately to disk. GBB's blackboard save and restore machinery uses dummy units to appropriately merge partial blackboard structures to recreate the original structure.

GBB/DBB implements the buffering of incoming messages using a separate reader process for each incoming message channel. This allows a special "immediate evaluation" message type. The form:

`(send-priority-form` *form node*`)`

causes *form* to be evaluated by the reader process as soon as it is received. By convention, this form is not allowed to make modifications to the blackboard database (other than acquiring locks). An example of the use of **send-priority-form** is in acquiring locks for a remote node.

System-level synchronization is a simple matter in the DBB approach. Since only one KSI is executing at a time at the local node, the only blackboard interference is from received messages.[27] One strategy is to buffer all incoming messages during the execution of a KSI, only inserting messages between executions of KSIs. Another strategy is to immediately create and modify objects resulting from received messages, subject to object and region locking. This second strategy allows for slightly faster response to incoming messages, but if currently executing KSIs have locked the blackboard regions on which they are operating, the effective increase in responsiveness may be minor. A compromize strategy is to buffer messages, but allow messages with particular characteristics to interrupt the ongoing KSI.

Similarly, the queue-latency problem is identical to a single processor, centralized implementation and existing techniques can be used.

Node and region locking facilities are provided in the GBB/DBB implementation, but the cost of using them on remote nodes is substantially higher than in the SMBB approach. (Locking a remote unit or region requires more than twice the communication latency.) In GBB/DBB the **lock-node** and **lock-region** functions accept an additional `:NODE` keyword argument, specifying the unit or region is to be locked on a remote node. Because pointers to units are not available, the name and class of a unit can be used to specify a unit residing at a remote node.[28]

The cost of locking in loosely-coupled networks is likely to limit applications to transaction-based and FA/C-style approaches to semantic synchronization among the nodes.

6.3 The Blackboard Server Approach

The abstraction of the blackboard implementation provided by GBB is well-suited to the BBS approach. The blackboard structure is partitioned into non-overlapping areas of responsibility using GBB's **instantiate-blackboard-database** function at each node. A global check is made to insure that no overlap is specified. A space is the atomic level of responsibility partitioning; dividing responsibility for portions of a space between two nodes is not supported. This restriction avoids the problem of determining responsibility for units that could span a partitioned space.

In GBB, all references to particular spaces are made using *paths* created using **make-paths**. In the GBB/BBS implementation, **make-paths** uses the node responsibility partitioning to return a path

[26] This requires that dummy units contain all the link slots associated with the unit class.

[27] It is also possible to implement a DBB approach where each local node is a SMBB (or a BBS, discussed later) structure. In such situations the synchronization issues addressed in the previous section also pertain to local synchronization.

[28] Recall that units must be uniquely named within a class.

that points at the appropriate server node.

Then a unit is created in GBB/BBS, the path(s) associated with the unit are analyzed to determine which node is responsible for the unit. Since GBB supports units that reside on multiple spaces, there is an ambiguity when more than one space is indicated in the path.[29] When this situation arises, one of the spaces is selected as the responsible space and the unit is created at the node responsible for that space.[30] Once the unit has been created on a space, any remaining spaces included in the path are handled as follows.

If responsibility for a remaining space is with the same node as the responsible space, the unit is inserted on that space in the same manner as the uniprocessing GBB system. However, if responsibility is with a different node, a special *virtual unit* is created at that node. This virtual unit is placed on all remaining spaces in the path for which that node is responsible.

This procedure insures that:

- no copies of a unit exist among the nodes;
- at most one virtual unit exists at a node for a specific unit;
- either the unit (on spaces at the responsible node) or a virtual unit representing the unit (on spaces at client nodes) resides on each space specified in the path.

Virtual units serve as indirect unit references in GBB operations. When a node performs an operation on a virtual unit, a request to perform the operation is sent to the server node.

Blackboard retrieval by **find-units** requires transmitting a request to a server node if any retrieval space is not within the executing node's responsibility. Virtual units are returned indicating any found units.

GBB's design has several properties that complicate the BBS approach. GBB allows units to be added to and removed from multiple spaces dynamically. This means that a non-virtual unit can be deleted from all spaces on which it resides while still remaining on remote spaces as a virtual space. In this case, the server node retains responsibility for the node, even though it does not reside on any of its spaces. Of course, such nodes can only be access via links from other virtual or non-virtual nodes.

[29] GBB allows units to be created that do not reside on any space, responsibility for such "floating" units is allocated to the creating node.

[30] The responsible space determination is arbitrary from the programmer's viewpoint. In practice, the creating node is used, if possible.

GBB also allows units to be deleted. In the BBS implementation, bidirectional pointers must be maintained between the actual unit and any virtual unit representing it so that if a request is made to delete it (or its virtual representative) the entire structure can be deleted.

Finally, GBB allows *dynamic units*: units that move within a space as their *dimensional values* are modified. Supporting this property requires potentially repositioning virtual units (as well as the original) if a dimensional value is modified. Again, the bidirectional links are useful here. Fortunately, it is possible to tell at blackboard initialization time which unit classes (and which slots within the class) might cause units to move on particular nodes.

Virtual units can also be used to cache values. Since a unit and its virtual units are bidirectionally linked. Updates to the original unit can be migrated to the virtual copies. Whether or not caching is beneficial is dependent upon the architectural and application environments. In any event, we are not implementing caching in our initial GBB/BBS implementation.

7 Summary and Status

We have discussed the issues associated with parallel and distributed multiprocessing blackboard architectures. We considered concurrency at a number of levels, particularly emphasizing concurrent execution of KSIs. By examining details of KSI interactions, we have considered a number of implementation options, each with different performance characteristics and applicability. We also presented three general parallel/distributed blackboard design approaches and described the characteristics and requirements of each.

None of these approaches provide a "magic" means of converting a serial blackboard application to a multiprocessing application. Achieving effective blackboard multiprocessing is hard, and until a number of diverse applications are attempted using different multiprocessing approaches, it will be difficult to predict the performance of a particular application using a particular approach. It is our hope that the extensions we are building for GBB will assist pioneering applications in experimenting with different multiprocessing approaches, gathering the performance measures that will guide future multiprocessing application developers.

The GBB/DBB approach has been implemented for Texas Instruments Explorer workstations and is undergoing tests and performance measures. The SMBB and BBS approaches remain in the design stage. Initial work on the GBB/SMBB implemen-

tation will start in September 1988 using the UMass Parallel Implementation of Common Lisp (PICL) system. The GBB/BBS implementation will not begin until we have developed considerable experience with the other two implementations.

Acknowledgments

Kevin Gallagher is the principal implementer of GBB and is a partner in GBB's ongoing refinement and extension. John Brolio implemented the GBB/DBB approach. Victor Lesser made helpful suggestions that were incorporated into the paper.

References

[1] Victor R. Lesser and Lee D. Erman. A retrospective view of the Hearsay-II architecture. In *Proceedings of the Fifth International Joint Conference on Artificial Intelligence*, pages 790–800, Tiblisi, Georgia, USSR, August 1977.

[2] Daniel D. Corkill, Kevin Q. Gallagher, and Kelly E. Murray. GBB: A generic blackboard development system. In *Proceedings of the National Conference on Artificial Intelligence*, pages 1008–1014, Philadelphia, Pennsylvania, August 1986.

[3] R. Bisiani. A software and hardware environment for developing AI applications on parallel processors. In *Proceedings of the National Conference on Artificial Intelligence*, pages 742–747, Philadelphia, Pennsylvania, August 1986.

[4] R. Bisiani, F. Alleva, F. Correrini, A. Forin, F. Lecouat, and R. Lerner. Heterogeneous parallel processing: The Agora shared memory. Technical Report CMU-CS-87-112, Computer Science Department, Carnegie-Mellon University, Pittsburgh, Pennsylvania, March 1987.

[5] L. Gasser, C. Braganza, and N. Herman. Implementing distributed artifical intelligence systems using MACE. In *Proceedings of the 3rd IEEE Conference on Artificial Intelligence Applications*, pages 315–320, 1987.

[6] 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*, pages 875–883, Milan, Italy, August 1987.

[7] Lee D. Erman, Frederick Hayes-Roth, Victor R. Lesser, and D. Raj Reddy. The Hearsay-II speech-understanding system: Integrating knowledge to resolve uncertainty. *Computing Surveys*, 12(2):213–253, June 1980.

[8] 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.

[9] Richard D. Fennell and Victor R. Lesser. Parallelism in Artificial Intelligence problem solving: A case study of Hearsay II. *IEEE Transactions on Computers*, C-26(2):98–111, February 1977.

[10] Richard Dean Fennell. *Multiprocess Software Architecture for AI Problem Solving.* PhD thesis, Carnegie-Mellon University, May 1975.

[11] J.R. Ensor and J.D. Gabbe. Transactional blackboards. *International Journal for AI in Engineering*, 1(2):80–84.

[12] Victor R. Lesser and Lee D. Erman. Distributed interpretation: A model and experiment. *IEEE Transactions on Computers*, C-29(12):1144–1163, December 1980.

[13] Victor Lesser, Daniel Corkill, Jasmina Pavlin, Larry Lefkowitz, Eva Hudlická, Richard Brooks, and Scott Reed. A high-level simulation testbed for cooperative distributed problem solving. Technical Report 81-16, Department of Computer and Information Science, University of Massachusetts, Amherst, Massachusetts 01003, June 1981. (Revised and shortened versions of this report appeared in *Proceedings of the Distributed Sensor Networks Workshop*, MIT Lincoln Laboratory, Lexington, Massachusetts, pages 247–270, January 1982, and in *Proceedings of the Third International Conference on Distributed Computer Systems*, pages 341–349, October 1982.).

[14] Victor R. Lesser and Daniel D. Corkill. The Distributed Vehicle Monitoring Testbed: A tool for investigating distributed problem solving networks. *AI Magazine*, 4(3):15–33, Fall 1983. (Also to appear in *Readings from AI Magazine 1980–1985*, in press, 1988).

[15] V. Jagannathan. Realizing the concurrent blackboard model. Technical Report BCS-G2010-61, Boeing Advanced Technology Center, P.O. Box 24346, MS 7L-64, Seattle, Washington 98124, March 1988.

[16] Kevin Q. Gallagher and Daniel D. Corkill. Blackboard retrieval strategies in GBB. Technical Report 88-39, Department of Computer and Information Science, University of Massachusetts, Amherst, Massachusetts 01003, August 1988.

[17] Daniel D. Corkill and Kevin Q. Gallagher. Tuning a blackboard-based application: A case study using GBB. In *Proceedings of the National Conference on Artificial Intelligence*, St. Paul, Minnesota, August 1988. (To appear.).