JOURNAL OF PARALLEL AND DISTRIBUTED COMPUTING 15, 103–117 (1992)

Operating System Support for Animate Vision*

B. MARSH, C. BROWN, T. LEBLANC, M. SCOTT, T. BECKER, P. DAS, J. KARLSSON, AND C. QUIROZ

Computer Science Department, The University of Rochester, Rochester, New York 14627

Animate vision systems couple computer vision and robotics to achieve robust and accurate vision, as well as other complex behavior. These systems combine low-level sensory processing and effector output with high-level cognitive planning—all computationally intensive tasks that can benefit from parallel processing. A typical animate vision application will likely consist of many tasks, each of which may require a different parallel programming model, and all of which must cooperate to achieve the desired behavior. These *multi-model* programs require an underlying software system that not only supports several different models of parallel computation simultaneously, but which also allows tasks implemented in different models to interact. This paper describes the Psyche multiprocessor operating system, which was designed to support multi-model programming, and the Rochester Checkers Player, a multi-model robotics program that plays checkers against a human opponent. Psyche supports a variety of parallel programming models within a single operating system by according first-class status to processes implemented in user space. It also supports interactions between programming models using model-independent communication, wherein different types of processes communicate and synchronize without relying on the semantics or implementation of a particular programming model. The implementation of the Checkers Player, in which different parallel programming models are used for vision, robot motion planning, and strategy, illustrates the use of the Psyche mechanisms in an application program, and demonstrates many of the advantages of multi-model programming for animate vision systems.

1. ANIMATE VISION

Vision can be viewed as a passive observational activity, or as one intimately related to action (e.g., manipulation, navigation). In *passive vision* systems the camera providing the image input is immobile. *Active vision* systems use observer-controlled input sensors [1]. Active vision results in much simpler and more robust vision algorithms for several reasons. A fixed camera may not have an object in view, whereas active vision can use physical search, through navigation or manipulation, and can change intrinsic or extrinsic camera parameters. Static camera placement results in nonlinear, ill-posed problems, whereas known, controlled camera movements and knowledge of camera placement provide self-generated constraints that simplify processing. Stereo fusion is intractable, whereas an actively verging system simplifies stereo matching. A solitary fixed camera imposes a single, possibly irrelevant, coordinate system; active vision can generate and use exocentric coordinate frames, yielding more robust quantitative and qualitative algorithms, and serving as a basis for spatial memory. Fixed spatial resolution limits imaging effectiveness, whereas variable camera parameters can compensate for range, provide a varying depth of field, and indirectly give information about the physical world. Segmentation of static, single images is a known intractable problem, whereas gaze control helps segmentation: active vergence or object tracking can isolate visual phenomena in a small volume of space, simplifying grouping.

Another aspect of active vision is its behavioral character; that is, intelligent activity, including perception, can be structured as vertically integrated skills (or *behaviors*) that are applied in particular contexts. Table I shows some visual capabilities in their behavioral contexts.

Another dimension for classifying computer vision approaches is reconstructive versus animate. In the *reconstructionist* or *general-purpose* paradigm, the vision task is to reconstruct physical scene parameters from image input, to segment the image into meaningful parts, and ultimately to describe the visual input in such a way that higher-level systems can act on the descriptions to accomplish general tasks. During the last decade, substantial progress in reconstructionist vision has been made using both passive and active systems that exploit physical and geometric constraints inherent in the imaging process [17]. However, reconstructionist vision appears to be nearing its limits without reaching its goal.

An alternative to reconstructionist vision derives from the observation that biological systems do not, in general, perform goal-free, consequence-free vision [4]. This observation suggests that vision may, of necessity, be a more interactive, dynamic, and task-oriented process than is assumed in the reconstructionist approach. *Ani-*

* This research was supported by the National Science Foundation under Grants IRI-8920771, CDA-8822724, CCR-9005633, and IRI-8903582. Support also came from ONR/DARPA Contract N00014-82-K-0193. Brian Marsh was supported by a DARPA/NASA Graduate Research Assistantship in Parallel Processing. The government has certain rights in this material.

0743-7315/92 $5.00

TABLE I
Computations Simplified by Behavioral Assumptions

Visual task	Behavioral context
Shape from shading	Light source not directly behind viewer [25]
Time to adjacency	Rectilinear motion; gaze in the direction of motion [21]
Kinetic Depth	Lateral head motion while fixating a point in a stationary world [4]
Color Homing	Target object is distinguished by its color spectrum [36]
Optic Flow	Texture-rich environment [16]
Stereo Depth	System can fixate environmental points [37]
Edge Homing	Target position can be described by approximate directions from texture in its surround [24]
Binocular Object Tracking	Vergence and tracking operate simultaneously [13]

mate vision researchers, inspired by successful biological systems, seek to develop practical, deployable vision systems using two principles: the active, behavioral approach (Table I), and *task-oriented* techniques that link perception and action. Table II summarizes the key differences between the reconstructionist vision paradigm and the task-oriented approach.

Animate vision thus needs cooperation between complex high-level symbolic algorithms and intensive low-level, real-time processing. The computations required by animate vision systems are so extensive that a parallel implementation is necessary to achieve the required performance. Fortunately many of the tasks in an animate vision system are inherently parallel. Inputs from multiple sensors can be processed in parallel. Early vision algorithms are intensely data-parallel. Planning and strategy algorithms frequently search a large state space, which can be decomposed into smaller spaces that are searched in parallel. Thus, there is no problem finding parallelism in the application. However, the type of parallelism we would like to exploit varies among tasks in the system—no single model of parallel computation is likely to suffice for all tasks.

TABLE II
Key Differences between Passive Vision and Task-Oriented Vision

Passive vision	Task-oriented vision
Use all vision modules	Use a subset of vision modules
Process entire image	Process areas of the image
Maximal detail	Sufficient detail
Extract representation first	Ask question first
Answer question from representation data	Answer question from scene data
Unlimited resources	Limited resources

The difficulty arises because parallelism can be applied in many ways, using different programming constructs, languages, and runtime libraries for expressing parallelism. Each of these environments can be characterized by the process model it provides: the abstraction for the expression and control of parallelism. The process model typically restricts the granularity of computation that can be efficiently encapsulated within a process, the frequency and type of synchronization, and the form of communication between processes. A typical animate vision application will likely consist of many tasks, each of which may require a different parallel programming model, and all of which must cooperate to achieve the desired behavior. These *multi-model* programs require an underlying software system that not only supports several different models of parallel computation simultaneously, but which also allows tasks implemented in different models to interact.

In this paper we argue that an architecture for animate vision systems must support multi-model programming. We describe an operating system, called Psyche, that was designed to support multi-model programming. We illustrate the use of Psyche, and the general concept of multi-model programming, by describing the implementation of the Rochester Checkers Player, a multi-model robotics application. The Checkers Player visually monitors a standard checkerboard, decides on a move in response to a move by a human opponent, and moves its own pieces. We describe the implementation of our Checkers Player in detail, emphasizing the use of several different programming models, and the integration of tasks in the implementation.

2. SOFTWARE REQUIREMENTS FOR ANIMATE VISION

Animate vision systems are inherently parallel. The hardware devices they use provide one source of parallelism. The algorithms used for device control and for combining perception and action provide another source. The real issue is how to harness the parallelism inherent in the application without being overwhelmed by the complexity of the resulting software. Our experiences with two DARPA benchmarks for parallel computer vision [11, 34] and a recent goal-oriented system [26, 27] illustrate the utility of multiple parallel programming environments for implementing computer vision algorithms, and the difficulty of successfully integrating the components of an animate vision system.

The first DARPA benchmark contained a suite of non-interacting routines for low- and high-level vision tasks. The low-level vision routines required manipulation of

two-dimensional pixel arrays using data parallelism, best accomplished using an SIMD style of computation. To implement these functions we used BBN's Uniform System library package [33]. We were able to show that pixel-level data-parallel functionality could be implemented on a shared-memory multiprocessor with nearly linear speedup given additional processors [12]. (Although we now use pipelined hardware for most low-level vision tasks, those functions not available in hardware can be implemented reasonably in software.)

The functions for high-level vision required coarser-grain parallelism than is provided by the Uniform System. To implement these functions we used two parallel programming environments developed at Rochester: a message-passing library [19] and a parallel programming language [28]. These examples demonstrated the utility of the message-passing paradigm on a shared-memory machine.

Several of the tasks in the first benchmark suite called for graph algorithms that are naturally implemented with many independent, lightweight processes, one per node in the graph. The lack of such a programming model was a major impediment in the development of the graph algorithms, and led to the subsequent development of a new programming environment [29].

Each of these vision tasks naturally suggested a particular model of parallelism that made programming that particular application easier in some way. Clearly, having multiple programming models to chose from was a benefit of our software environment.

The second benchmark called for an integrated scene-describing system. This benchmark emphasized integration of several levels of image understanding to describe a scene of polygons at various discrete depths. It thus underscored the usefulness of a unified approach to multi-model parallelism. Unfortunately, our previous individual solutions were implemented using several different programming models and we lacked the system support necessary to integrate them. The data structures produced by our SIMD computations could not be accessed directly by processes in our MIMD computations. Processes of different types could not synchronize. Ironically, the very diversity that facilitated our success in the first benchmark prevented a successful implementation of the second.

A more recent example of the difficulties in constructing a real-time, flexible task-oriented application is the TEA system [26, 27], in which Bayes nets, planning, and a maximum-expected-utility decision rule provide a knowledge base and control structure to choose and apply visual actions for information acquisition. Visual actions involve camera movements, imagery selection (foveal, peripheral, color, grey-scale), and operator selection. TEA is designed to embody all the characteristics of a task-oriented system (Table II). Currently TEA uses hardware pipelined parallelism for some low-level vision tasks, but does not exploit parallelism elsewhere. As a result, the system is too slow to deal with a dynamic environment. We hope to achieve real-time performance (thus producing a real-time planning and acting system) by exploiting parallelism (both data parallelism and functional parallelism) in the implementation. Image processing and analysis can be parallelized easily, speeding up individual modules and allowing several visual modules to run together on a scene. Propagation of belief through the network can proceed in parallel with motor control for object tracking or changing of viewpoint. Recent work with an eight-node transputer configuration has demonstrated the practicality of a multi-model approach, and has shown that sufficient image input bandwidth is available to support real-time operation.

The DARPA benchmarks, the TEA system, and other applications experience illustrate the potential advantages of using a large-scale MIMD multiprocessor as the controlling architecture in integrated animate vision systems. Our experiences also demonstrate the importance of matching each application, or parts of a large application, to an appropriate parallel programming environment, and the importance of integrating functions across environment boundaries. We will now describe a multiprocessor operating system designed to facilitate both the construction and integration of multiple parallel programming environments.

3. THE PSYCHE MULTIPROCESSOR OPERATING SYSTEM

3.1. Background

The widespread use of distributed and multiprocessor systems in the last decade has spurred the development of programming environments for parallel processing. These environments provide many different notions of processes and styles of communication. Coroutines, lightweight run-to-completion threads, lightweight blocking threads, heavyweight single-threaded processes, and heavyweight multi-threaded processes are all used to express concurrency. Individually routed synchronous and asynchronous messages, unidirectional and bidirectional message channels, remote procedure calls, and shared address spaces with semaphores, monitors, or spin locks are all used for communication and synchronization. These communications and process primitives, among others, appear in many combinations in the parallel programming environments in use today.

A parallel programming environment defines a model of processes and communication. Each model makes assumptions about communication granularity and fre-

quency, synchronization, the degree of concurrency desired, and the need for protection. Successful models make assumptions that are well matched to a large class of applications, but no existing model has satisfied all applications. Problems therefore arise when we attempt to use a single operating system as the host for many different models, because the traditional approach to operating system design adopts a single model of parallelism and embeds it in the kernel. The operating system mechanisms are seldom amenable to change and may not be well matched to a new parallel programming model under development, resulting in awkward or inefficient implementations in some parallel applications. For example, although the traditional Unix interface has been used to implement many parallel programming models, in most cases the implementation has needed to compromise on the semantics of the model (e.g., by blocking all threads in a shared address space when any thread makes a system call) or accept enormous inefficiency (e.g., by using a separate Unix process for every lightweight thread of control).

Since 1984 we have explored the design of parallel programming environments on shared-memory multiprocessors. Using the Chrysalis operating system from BBN [5] as a low-level interface, we created several new programming libraries and languages, and ported several others [20]. We were able to construct efficient implementations of many different models of parallelism because Chrysalis allows the user to manage memory and address spaces explicitly, and provides efficient low-level mechanisms for communication and synchronization. As in most operating systems, however, Chrysalis processes are heavyweight (each process resides in its own address space), so lightweight threads must be encapsulated inside a heavyweight process, and cannot interact with the processes of another programming model.

Each of our programming models was developed in isolation, without support for interaction with other models. Our experiences with the implementation of these individual models, coupled with our integration experiences in the DARPA benchnmarks, convinced us of the need for a single operating system that would provide both an appropriate interface for implementing multiple models and conventions for interactions across models. The Psyche multiprocessor operating system [30–32] was designed to satisfy this need.

Rather than establish a high-level model of processes and communication to which programming environments would have to be adapted, Psyche adopts the basic concepts from which existing environments are already constructed (e.g., procedure calls, shared data, address spaces, and interrupts). These concepts can be used to implement, in user space, any notion of process desired. These concepts can also be used to build shared data structures that form the basis for interprocess communication between different types of processes.

3.2. *Psyche Kernel Interface*

The Psyche kernel interface provides a common substrate for parallel programming models implemented by libraries and language runtime packages. It provides a low-level interface that allows new packages to be implemented as needed and implementation conventions that can be used for communication between models when desired.

The kernel interface is based on four abstractions: *realms*, *protection domains*, *processes*, and *virtual processors*.

Each realm contains code and data. The code provides a protocol for accessing the data. Since all code and data are encapsulated in realms, computation consists of invocation of realm operations. Interprocess communication is effected by invoking operations of realms accessible to more than one process.

To facilitate the sharing of arbitrary data structures at run time, Psyche arranges for every realm to have a unique system-wide virtual address. This *uniform addressing* allows processes to share pointers without worrying about whether they might refer to different data structures in different address spaces.

Depending on the degree of protection desired, invocation of a realm operation can be as fast as an ordinary procedure call (*optimized* invocation), or as safe as a remote procedure call between heavyweight processes (*protected* invocation). The two forms of invocation are initiated in exactly the same way, with the native architecture's jump-to-subroutine instruction. In some cases this instruction generates a page fault, allowing the kernel to intervene when necessary during protected invocations.

A process in Psyche represents a thread of control meaningful to the user. A virtual processor is a kernel-provided abstraction on top of which user-defined processes are implemented. There is no fixed correspondence between virtual processors and processes. One virtual processor will generally schedule many processes. Likewise, a given process may run on different virtual processors at different points in time. On each physical node of the machine, the kernel time-slices the virtual processors currently located on that node.

As it invokes protected operations, a process moves through a series of protection domains, each of which embodies a set of access rights appropriate to the invoked operation. Within each protection domain, the representations of processes are created, destroyed, and scheduled by user-level code without kernel intervention. As a process moves among domains, it may be rep-

resented in many different ways (e.g., as lightweight threads of various kinds or as requests on the queue of a server).

Asynchronous communication between the kernel and virtual processors is based on signals, which resemble software interrupts. User-level code can establish interrupt handlers for wall clock and interval timers. The interrupt handlers of a protection domain are the entry points of a scheduler for the processes of the domain, so protection domains can be used as boundaries between distinct models of parallelism. Each scheduler is responsible for the processes executing within its domain, managing their representations, and mapping them onto the virtual processors of the domain.

These Psyche kernel mechanisms support multi-model programming by facilitating the construction of *first-class user-level threads* [23] and *model-independent communication* [22]. First-class user-level threads enjoy the functionality of traditional kernel processes, while retaining the efficiency and flexibility of being implemented outside the kernel. Model-independent communication allows different types of processes to communicate and synchronize using mechanisms that are not tied to the semantics or implementation of a particular parallel programming model.

3.3. First-Class User-Level Threads

In a multi-model programming system most programmers do not use the kernel interface directly; user-level thread packages and language runtime environments provide the functionality seen by the programmer. This means that the kernel is in charge of coarse-grain resource allocation and protection, while the bulk of short-term scheduling occurs in user space. In according first-class status to user-level threads, we intend to allow threads defined and implemented in user space to be used in any reasonable way that traditional kernel-provided processes can be used. For example, first-class threads can execute I/O and other blocking operations without denying service to their peers. Also, time-slicing implemented in user space can be coordinated with preemption implemented by the kernel.

Our general approach is to provide user-level code with the same timely information and scheduling options normally available to the kernel. Software interrupts are generated by the kernel when a scheduling decision is required of a parallel programming environment implemented in user space. Examples include timer expiration, imminent preemption, and the commencement and completion of blocking system calls. Timer interrupts support the time-slicing of threads in user space. Warnings prior to preemption allow the thread package to coordinate synchronization with kernel-level scheduling. Every system call is nonblocking by default; the kernel simply delivers an interrupt when the call occurs, allowing the user-level scheduler to run another thread.

The kernel and the runtime environment also share important data structures, making it easy to convey information in both directions. These data structures indicate the state of the currently executing process, the address of a preallocated stack to be used when handling software interrupts, and a collection of variable for managing the behavior of software interrupts. User-writeable data can be used to specify what ought to happen in response to kernel-detected events. When the kernel and user-level code are allowed to share data, changes in desired behavior can occur frequently (for example, when context switching in user space).

3.4. Model-Independent Communication

In Psyche, a multi-model program can be constructed as a set of modules (groups of realms), each of which may implement a (potentially different) programming model. Each module defines a set of interface procedures that are used to access the code and data encapsulated by the module. To communicate between modules, and hence between different programming models, processes invoke interface procedures to access the memory associated with a module.

Shared memory is a viable communication medium between programming models, but by itself is insufficient to implement a wide range of communication styles. Interprocess communication requires several steps, including data transfer, control transfer and synchronization. While shared memory is sufficient to implement data transfer, both control transfer and synchronization depend on the precise implementation of processes. For this reason processes of different types usually communicate using simple, low-level mechanisms (e.g., shared memory and spin locks, with no protection mechanisms in place) or generic, high-level communication primitives (e.g., remote procedure calls requiring kernel intervention for protection).

The Psyche approach to interprocess communication, especially when the communicating processes are of different types, is based on two concepts:

- *A procedural interface for control and data transfer*—Each shared data structure is encapsulated within a module and can only be accessed by invoking the appropriate interface procedures. Invocations (i.e., procedure calls to realm operations) implement control and data transfer. Either optimized or protected invocations may be appropriate, depending on whether the shared data structure resides within its own protection domain.
- *A kernel-supported interface for process management*—Each module that implements a parallel programming model provides an interface to a set of process man-

agement routines. These routines, which are typically used to block and unblock a thread of control implemented within the programming model, can be invoked from within shared data structures, providing a means for synchronization among dissimilar process types. A data structure shared between the kernel and user contains pointers to the current set of process management routines; these pointers are updated during each context switch.

These mechanisms can be used to implement two distinct types of interactions between dissimilar programming models: shared data structures and direct invocations from one programming model to another.

Shared data structures are typically passive; the code associated with a data structure is executed only when a process invokes an operation on the data structure. When a process needs to wait for some condition while executing in an interface procedure, the code (shared by all processes that access the data structure) can follow the pointers to the process management routines for the currently executing process. It can then save the address of the *unblock* routine in the shared data structure, and call the *block* routine. At a later point in time, when another process establishes the condition, that process can retrieve the pointer to the *unblock* routine, and call into the module that manages the waiting process, causing the process to unblock. The call to *unblock* may itself be a protected invocation if the two processes are from different programming environments.

A direct invocation from one programming model to another causes a process to move from its native programming environment into the runtime environment of another programming model. This type of invocation can be very efficient (allowing a process to execute in another environment at the cost of single procedure call), but poses several problems for the implementation. In particular, differences in process representation and behavior in the two environments can lead to *process interference*, wherein implicit assumptions about the nature of processes are embedded in the implementation of a programming model, and then violated by a calling process.

The simplest example of process interference occurs when a process enters a nonreentrant runtime environment. Many programming models use a run-until-block scheduling policy, and the underlying implementation typically assumes only one process can execute at a time in the environment. These implementations do not require explicit synchronization, relying instead on the implicit assumption of mutually exclusive execution. This assumption is violated if a process is allowed to enter the runtime environment at any time using a procedure call.

There are many examples of process interference, but in each case an assumption about the nature of processes is embedded in the implementation of a runtime environment, and then violated by a process that calls into that environment from outside. One way to avoid process interference is to disallow procedure calls between programming models, requiring instead that a native process execute on behalf of each caller. Rather than introduce the overhead of process creation and scheduling on every interaction between programming models, we can avoid process interference by placing two constraints on the structure of programming environments:

- The code in an environment must be organized so as to isolate process dependencies. Code and data that depend on the native process model are placed into a *full-model* portion of the environment. Code and data that only depend on the process management interface of the currently executing process are placed into a *semi-model* portion. Only native processes should execute code in the full-model portion of the environment; any process can execute code in the semi-model portion.
- Every process that enters a runtime environment from outside must pass through an interface procedure that schedules a process for execution. If the required operation is located in the semi-model portion of the environment, the calling process may be allowed to execute the operation immediately, or may be scheduled for execution at a later time. If the required operation is located in the full-model portion of the environment, a local representative must be created and scheduled for execution on behalf of the calling process.

By dividing a programming environment into full-model and semi-model portions, we separate those operations that require a native process for execution from the operations that can be performed by any process. Semi-model operations can be especially efficient, since they do not add process management overhead (such as process creation or scheduling) to each operation. Full-model operations, on the other hand, have access to all the resources of the host environment, including the power and flexibility of the host programming model.

3.5. RELATED WORK

Many researchers have addressed some aspect of multi-model programming, including operating system support for user-level implementations of programming models, and communication mechanisms for use across programming models. We will describe the work most closely related to our approach on both of these issues.

Implementing Multiple Models

Several systems have addressed the need for cooperation between the kernel and user-level thread package to facilitate scheduling. Like Psyche, these systems allow user-level software to control the impact of general-

purpose, kernel-level strategies on model-specific implementations.

As part of the Symunix project at New York University, Edler *et al.* [14] proposed a set of parallel programming extensions to the Unix kernel interface, including an asynchronous interface for the existing synchronous system calls in UNIX, and a quantum-extending mechanism designed to avoid preemption during critical sections. The temporary nonpreemption mechanism employs a counter in user space at a location known to the kernel. When entering a critical section, user-level code can increment the counter. Within reason, the kernel will refrain from preempting a process when the counter is nonzero. In contrast, the Psyche mechanism notifies the user level when preemption is imminent, without affecting the kernel policy.

At the University of Washington, Anderson *et al.* [2] have explored user-level scheduling in the context of the Topaz operating system on the DEC SRC Firefly multiprocessor workstation. For each address space, they maintain a pool of virtual processors (called scheduler activations) in the kernel. When a scheduler activation is preempted or blocks in the kernel, the kernel freezes its state and sends a new activation from the pool up into user space. The new activation (and any other running activations in the same address space) can examine the state of all processes in the address space and decide which ones to execute. The most important difference with Psyche is that scheduler activations notify the user level *after* an event has occurred, with the expectation that another activation on another processor will respond to the event. Psyche always notifies the virtual processor associated with an event, so that the event can be handled on the same processor on which it occurred, and by the virtual processor most affected by the event.

A somewhat different approach was proposed by Black for use in the Mach operating system [10]. Instead of having applications control their scheduling behavior by responding to events generated in the kernel, he added a collection of system calls to Mach so that threads may give hints to the kernel scheduler. These calls allow a thread to indicate that it should be descheduled, or to identify a particular thread that should be executed instead.

Integrating Multiple Models

Integrating multiple models within a single application requires that processes from each of the modules be able to communicate and synchronize. Previous work has considered how to cross traditional boundaries between programming models, such as machine or address space boundaries, but the problem of process interference has not been addressed.

Remote procedure call (RPC) [8] is a well-known communication mechanism that allows processes to invoke procedures located on other machines or in other address spaces. Most RPC systems avoid the problem of process interference by requiring that a native process execute the called procedure. LRPC [6], a lightweight remote procedure call facility for communication between address spaces in a shared-memory multiprocessor, allows the calling process to execute in the called environment, because the implementation assumes a single process model. Without this assumption, process interference could arise, and important optimizations used in LRPC for thread management might not be possible.

HRPC (heterogeneous remote procedure call) is a remote procedure call facility designed to accommodate hardware and software heterogeneity [7]. HRPC defines an interface for thread management similar to the process management interface in Psyche. Any thread package meeting the interface can be implemented easily using the HRPC runtime. Unlike Psyche, the HRPC thread management interface is not intended for use by other programming models to access model-specific functions.

The Portable Common Runtime (PCR) [35] supports multiple models within a single application by providing a common substrata of low-level abstractions, including threads, memory, and I/O. Processes from different models can interact by making "intercalls" between environments, but the process model (based on PCR threads) is the same in all environments.

Agora [9] is one the few systems designed for a heterogeneous, distributed environment that does not use some form of remote procedure call for communication. Agora provides a distributed shared memory for interprocess communication. Access to this shared memory is provided through customized access functions written in a Lisp-like specification language. The access functions coordinate the behavior of processes within the shared memory using local process management routines provided by the host operating system (i.e., Mach), much as a semi-model coordinates the behavior of processes that enter a new programming environment in Psyche. The primary difference between the two systems is that Agora defines new abstractions for use by all processes (a distributed shared memory and specific synchronization events), while Psyche allows processes to interact using their native abstractions for communication and synchronization.

4. A MULTI-MODEL PROGRAM FOR CHECKERS

The Rochester Checkers Player is a multi-model vision application implemented on top of Psyche. A checkers-playing robot conducts a game of checkers (draughts) against a human opponent, cyclically sensing the opponent's move, and then planning and executing its re-

sponse, all within about 5 s. The robot uses a voice synthesizer to issue status information, error messages, and occasional gratuitous remarks.

4.1. Hardware Environment

A modern computer vision laboratory is likely to include sophisticated effectors or mobile vehicles, and movable, computer-configurable sensors. The work described in this paper was performed in such a laboratory, which currently consists of six key components (Fig. 1): a binocular head containing movable cameras for visual input; a robot arm that supports and moves the head; a special-purpose parallel processor for high-bandwidth, low-level vision processing; and several choices of general-purpose MIMD parallel processors for computations from high-level vision and planning to motor control. Two components not relevant to this paper are a 16-degree-of-freedom Utah dextrous manipulator (hand) and a Dataglove for input of manipulation configurations.

The head has two movable grey-scale CCD television cameras and a fixed color camera providing input to a MaxVideo pipelined image-processing system. One motor controls the tilt angle of the two-camera platform, and separate motors control each camera's pan angle, providing independent vergence control (Fig. 2). The controllers allow sophisticated velocity and position commands and data read-back.

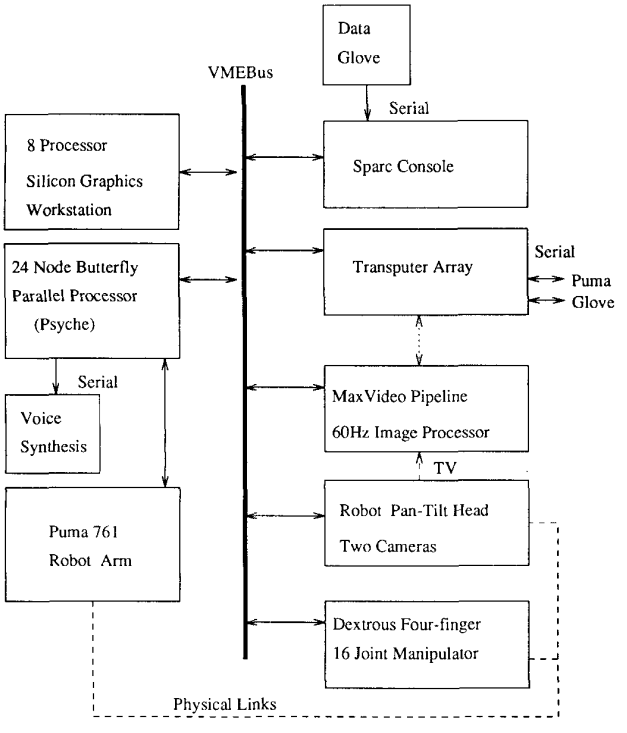

FIG. 1. Vision and robotics laboratory.

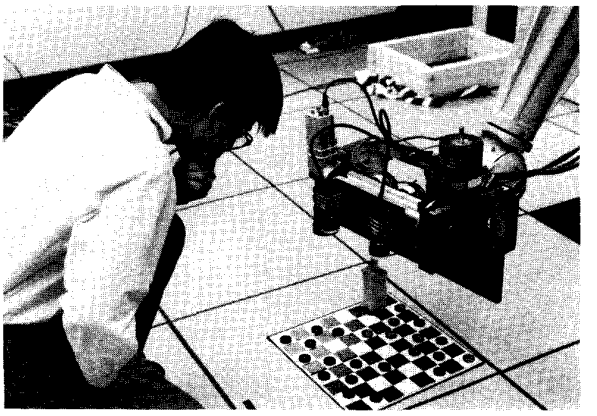

FIG. 2. In this configuration the robot head has one large color camera, two small grey-scale cameras, a single tilt motor, twin pan motors, and a passively compliant checker-pushing tool.

The robot body is a PUMA761 six-degree-of-freedom arm with a 2-m radius workspace and a top speed of about 1 m/s. It is controlled by a dedicated LSI-11 computer implementing the proprietary VAL execution monitor and programming interface.

The MaxVideo system consists of several independent boards that can be cabled together to achieve a wide range of frame-rate image analysis capabilities. The MaxVideo boards are all register programmable and are controlled by a host processor via the VME bus. The ZEBRA and ZED programming systems, developed at Rochester, make this hardware easily and interactively programmble.

A characteristic feature of our laboratory is the capability to use a multiprocessor as the central computing resource and host. Our BBN Butterfly Plus Parallel Processor has 24 nodes, each consisting of an MC68020 processor, MC68851 MMU, MC68881 FPU, and 4 MBytes of memory. The Butterfly is a shared-memory multiprocessor with nonuniform memory access times; local memory is roughly 12× faster to access than nonlocal memory. The Butterfly has a VME bus connection that mounts in the same card cage as the MaxVideo and motor controller boards. The Butterfly has a serial port on each board; we use the port to communicate directly with the VAL robot control software. A Sun 4/330 workstation acts as a host terminal system.

4.2. Parallel Programming Environments

The Checkers Player, like many animate vision applications, consists of tasks to implement sensing, planning, and action. In our implementation, each of these functions is implemented using a different parallel programming environment: Multilisp, Lynx, the Uniform System, or Uthread.

Multilisp [15] is a Lisp extension for parallel symbolic programming developed at MIT. The unit of parallelism in Multilisp is the *future*, which is a handle for the evaluation of an arbitrary s-expression that is evaluated in parallel with the caller. Although the value of the s-expression is *undetermined* until the expression has been completely evaluated, the handle for that value can be passed around and referenced as needed. Futures are evaluated last-in–first-out to avoid the combinatorial growth in the number of futures that would otherwise result from the extensive use of recursion in Lisp. Any attempt to reference a future before the value is determined causes the caller to block. These two mechanisms, parallel execution via futures and synchronization via references to futures, suffice to build parallel programs in Lisp.

Lynx [28] is a parallel programming language based on message passing. Lynx programs consist of multiple heavyweight processes, each with its own address space, that exchange messages using named communication channels (*links*). Each heavyweight process consists of multiple lightweight threads of control that communicate using shared memory. Thread creation occurs as a side-effect of communication; a new thread is automatically created to handle each incoming message. Condition variables are used for synchronization between threads in the same process; synchronous message passing provides synchronization between processes.

The Uniform System [33] is a shared-memory, data-parallel programming environment developed at BBN for the Butterfly. Within a Uniform System program, task generators are used to create a potentially large number of parallel tasks, each of which operates on some portion of a large shared address space. Task descriptors are placed on a global FIFO work queue, and are removed by processors looking for work. Each task must run to completion, at which time another task is removed from the task queue. Each processor maintains processor-private data, which may be shared by all tasks that execute on that processor. The primary mechanism for communication is the globally shared memory, which is accessible to all Uniform System tasks on all processors. Since tasks are not allowed to block, spinlocks are used for synchronization.

Uthread is a simple, lightweight thread package developed for Psyche that can be called from C++ programs. Uthread is the general-purpose programming environment of choice in Psyche, and is frequently used to implement single-threaded servers.

4.3. Data Structures

The primary data structures used in the Checkers Player are the representations of the checkerboard and the moves. There are four different board representations, each used for different tasks:

1. A digitized image of the board from the TV camera ($512 \times 512 \times 8$ bits).
2. Calibration information that locates the squares of the board in the robot's workspace.
3. A quantitative description of the (X, Y, Z) location of the centroids of pieces on the board and their color.
4. A symbolic description of the board, denoting which squares contain pieces of which color.

Three different representations for moves are used, depending on the context in which a move is considered. One representation is simply the new board state that results from the move. A move may also be represented as a sequence of physical coordinates for the robot motion commands. A third representation is the list of partial moves (i.e., a push or a sequence of jumps) needed to execute a move. A short pipeline of moves is maintained to support backing up to legal or stable states.

The various representations for the board and move data structures are encapsulated within the *Board Module*, which provides synchronized access to the data structures, and translation routines between the various representations. The Board Module is implemented using the Uthread package; a single thread of control is created to initialize the data structures, after which the module becomes a passive data structure shared by tasks from other programming models. The synchronization routines provided by the Board Module use the Psyche conventions for process management to implement semaphores that can be called by any model.

4.4. Modules

The execution of the program is implemented as a series of moves, each of which requires the cooperation of several modules and programming models. Control flow among the modules is indicated by the arrows in Fig. 3.

In addition to the Board Module, there are seven other modules in the Checkers Player implementation. Three of these modules manage the robot devices; the remainder implement vision, checkers strategy, and motion planning.

Camera Manager—a Uthread module that initializes the VME memory used to control and access the MaxVideo hardware. The Board Interpreter accesses this memory directly to retrieve an image from the MaxVideo framebuffer.

Board Interpreter—a Uniform System program that transfers an image from VME memory to local Butterfly memory, and produces a symbolic description of the checkers in the image.

Move Recognizer—a Uthread module that compares

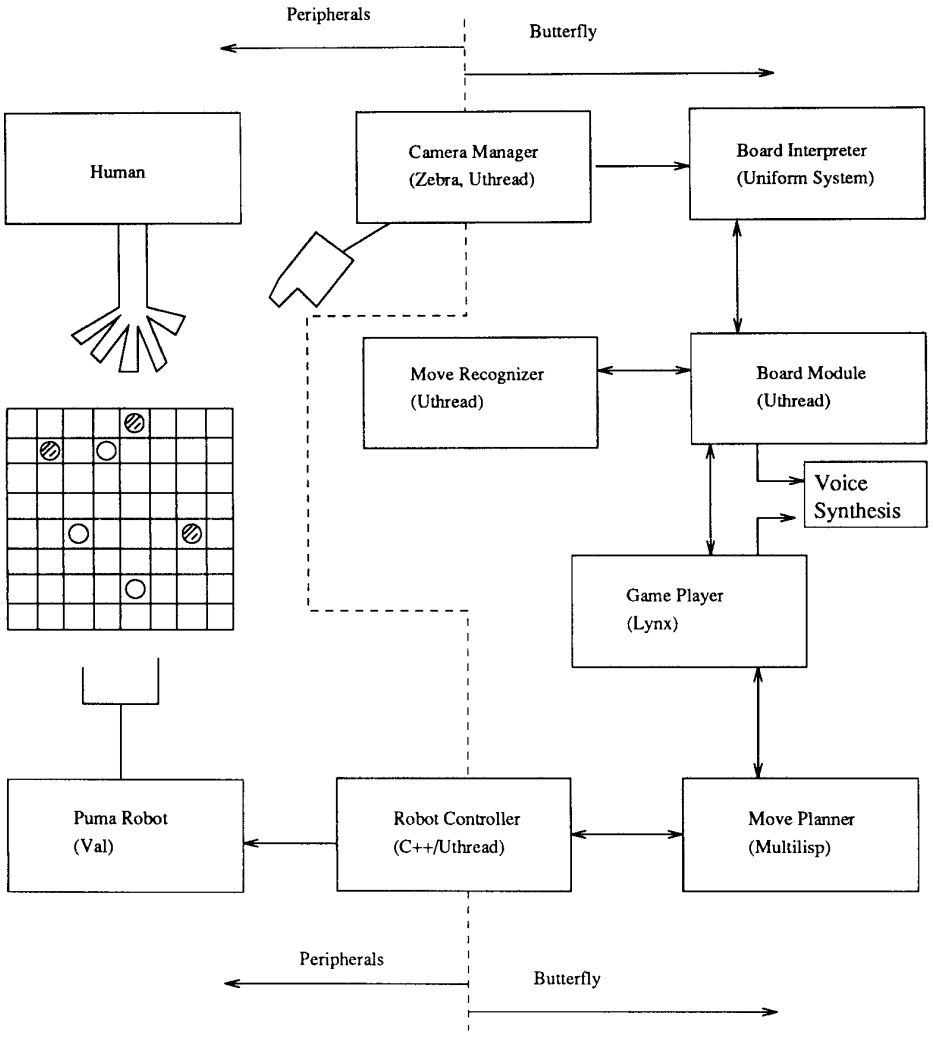

FIG. 3. Functional modules and communication paths in the Checkers Player. Multiple models of parallelism (to the right of the dotted line) are implemented under Psyche on the Butterfly. Perceptual and motor modules (to the left of the dotted line) reside on the Butterfly and in peripherals.

two successive symbolic board interpretations produced by the Board Interpreter, and recursively decomposes the differences into a sequence of legal partial moves (i.e., single jumps or moves) that transforms the first interpretation into the second.

Game Player—a checkers game-playing program written in Lynx. It takes as input the list of partial moves describing the human's move and produces as output the list of partial moves to be made in response. A single multithreaded master process manages the parallel evaluation of possible moves; slave processes perform subtree exploration on behalf of the master.

Move Planner—a trajectory calculation and planning program written in Multilisp. This program transforms the information gained from vision into a form useful for planning the robot's actions in the world. It constructs, in parallel, artificial potential fields that have peaks reflecting square occupancies and bias reflecting the goal location (see Fig. 4) [18]. For individual moves, the goal location is a particular square; when removing pieces, the goal location is one of eight goal areas off the board. These potential fields are considered in parallel, using a local search procedure that yields a gradient-descent path along which a checker can be pushed. The algorithm allows pieces to be temporarily moved aside or swapped with the moving piece. Candidate plans can be ranked by speed, effort, or other metrics.

Speech Controller—a Uthread module that manages

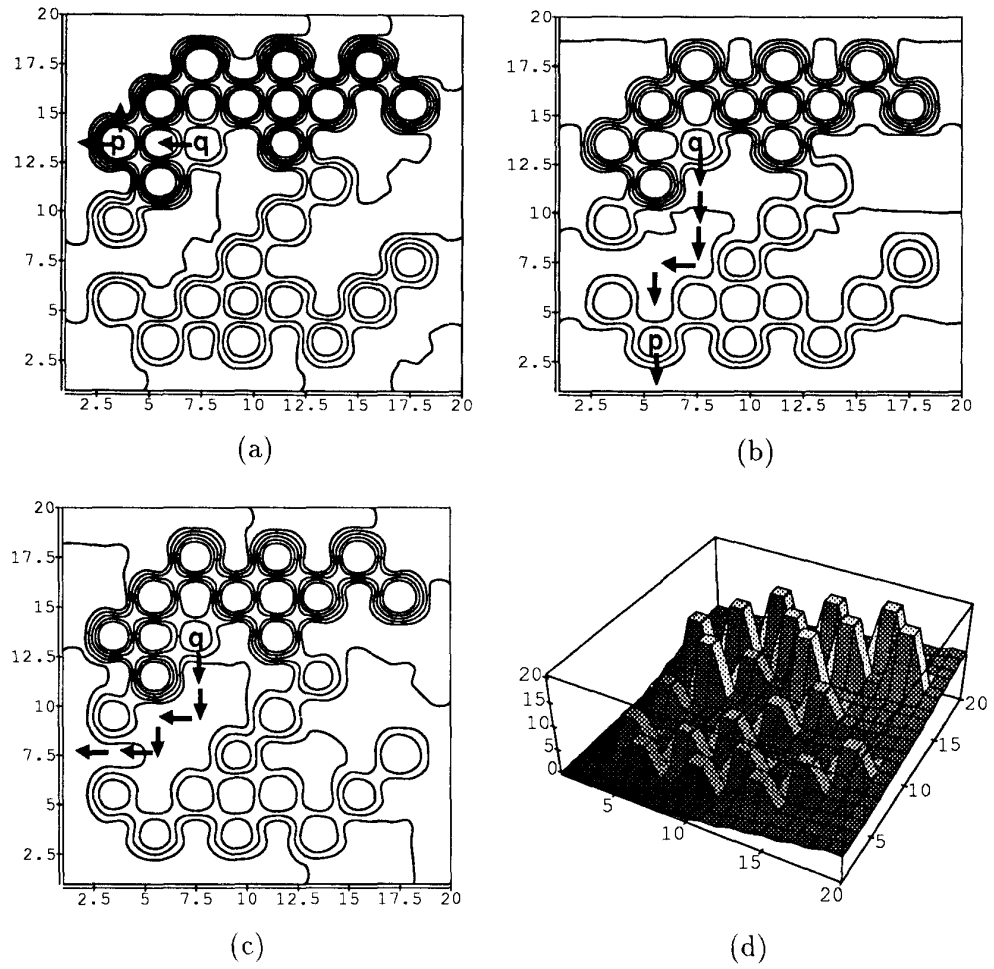

FIG. 4. (a) A contour map of the path through the potential field resulting when piece q is moved to the left side of the board by first removing and then replacing a differently colored piece p. (b) A longer alternative path through a piece p that is the same color as q, allowing p to be moved off the bottom of the board, to be replaced by q. (c) The actual path selected minimizes both the path through the potential field and the cost of manipulating pieces. (d) A 3-D representation of the potential field for the path selected, which moves q off the bottom-left corner of the board.

the voice synthesizer using a serial line connection. This module handles invocation requests for speech generation. Since interactions with the speech board must take place on the node to which the board is attached, a Uthread process in the Speech Controller serves as a local proxy for interactions with the board.

Robot Controller—a Uthread module that controls a serial line connection between the Butterfly and the robot. This module sends movement commands to the robot (equivalent to MoveTo (X,Y,Z, SPEED)), and awaits notification of successful completion. Once again, a Uthread process serves as a local proxy for interactions with the board.

4.5. Intermodel Communication

There are two kinds of interactions between modules in the Checkers Player: semi-model operations executed by a calling process, and full-model operations that require a native process for execution. Table III lists the operations exported by the various modules, and indicates whether an operation is a semi-model or a full-model operation. (Some modules export no operations, and therefore are not listed in the table.)

The Board Module is a passive data structure, and therefore only exports semi-model operations. After initialization, all the functions within the Board Module are

TABLE III
Exported Operations Used in Checkers Player

Module name	Operation	Type
Board Module	set_checkers_desc	semi
	get_checkers_desc	semi
	get_human_move	semi
	set_human_move_status	semi
	analyze_board	semi
	update_vision	semi
	update_robot	semi
Move Recognizer	find_partial_move	semi
Move Planner	do_partial_move	full
Robot Controller	move_robot	full
Speech Controller	speak	full

executed by a calling process from another environment. Uniform System processes from the Board Interpreter call into the Board Module to register a new board configuration. Lynx processes from the Game Player call into the Board Module to wait for a new human move. If the new board configuration represents a legal move, the Uniform System process from the Board Interpreter signals the waiting Lynx process using a semaphore for condition synchronization. The semaphore is implemented using the process management interface for the currently executing process. When the Lynx process must wait for the semaphore, a call is made back to the Lynx runtime to block the process. However, before doing so the semaphore implementation records the process identifier and the address of the *unblock* routine for later use by the Uniform System process that registers a new move.

Lynx processes from the Game Player call the Move Planner to get a partial move executed. Process interference between the Lynx process and the Multilisp environment is avoided by creating a Multilisp future to perform the operation. On entry to `do_partial_move` the Lynx process enqueues its arguments to the called function, and attempts to wake up a Multilisp server process. When the operation is completed by the server process, the *unblock* routine in the Lynx runtime environment is called to reschedule the Game Player's process.

The Robot Controller and Speech Controller both export full-model operations, since they require access to serial lines associated with a particular node in the machine. The operations provided by these modules use server processes located on the appropriate node to implement communication with the serial line.

4.6. *The Benefits of Multi-model Programming*

The Checkers Player implementation demonstrates the advantages of decomposing animate vision systems by function, and independently selecting an appropriate parallel programming model for each function. Programmers were free to choose the most appropriate programming model for each module, without placing constraints on how modules would be integrated. The Board Interpreter was written in the Uniform System because the fine-grain parallelism and shared memory of that programming model made it easy to implement the data parallel algorithm used for checker identification. The Game Player module was written in Lynx several years ago, and was easily incorporated into the Checkers Player without significant modifications. The Motion Planner module was written in Multilisp because the symbolic features of Lisp and the parallelism provided by the future construct made it easy to express the planning algorithm. The Board Module, the Robot Controller, and the Speech Controller were written in Uthread because their functions were both low-level and simple; none required the special features of a sophisticated programming model.

Each of the modules was developed independently by a different person, and then integrated into the whole. Integration was simplified by the use of model-independent communication: each module exports a model-independent communication interface for use by other modules. Process interference was avoided within each module by separating operations into semi-model and full-model portions of the module.

Source code statistics for the Checkers Player are presented in Table IV. For each module this table lists the number of lines of code in the application, the underlying runtime environment, and the interface procedures (stubs) for the exported operations. Implementation required about 2900 lines of new application-level code, and 550 lines of stub interface code (which were produced by hand, but which could easily be produced automatically by a stub generator). We were able to reuse 1800 lines of Lynx code, and all of the code in the runtime environments. We had to add almost 900 lines of code to the Lynx and Multilisp runtime environments to implement the semi-model interface used by processes from other models, but this code is not application specific, and can be reused in future applications.

Only a small percentage of the code written specifically for the Checkers Player required expertise with system software. Application programmers implemented the Board Interpreter, Move Planner, and Board Module. Each of these programmers had expertise in one programming model, but none required experience with the other models in use. Even though the Board Module was written in Uthread, and is accessed by Uniform System processes and Lynx processes, there was no need for the implementor to understand the details of all three programming models; the code contains no provisions for the specific types of processes that call it. The ease with which the Board Module was built demonstrates how model-independent communication simplifies the con-

TABLE IV
Source Code Statistics for the Checkers Player

Module	Model	Application		Runtime		Stubs
		Full	Semi	Full	Semi	
Board Module	Uthread	36	687	2436	20	105
Camera Manager	Uthread	84	0	2436	20	0
Board Interpreter	Uniform System	380	0	8057	80	90
Move Recognizer	Uthread	0	547	2436	20	13
Game Player	Lynx	1800	0	8838	354	150
Move Planner	Multilisp	900	0	13127	512	60
Robot Controller	Uthread	27	0	2436	20	90
Speech Controller	Uthread	211	0	2436	20	14

struction and use of modules that are shared between different process types.

The implementation required two months of part-time effort by five people. There are two reasons for the timely success of this effort: each programmer was free to choose the best available programming model for each module, and the integration effort did not require intimate knowledge of the other programming models in use. Most of the effort was devoted to application concerns, such as high-level and low-level vision, parallel potential field minimization, and a novel algorithm for determining a sequence of partial moves from two board configurations. Relatively little effort was required for integration. We were able to change many decisions about which modules would communicate directly with each other, and which should use certain shared data structures. Our experiences have convinced us of the importance of integration through shared data abstractions, and customized communication protocols accessible from every parallel programming model.

5. CONCLUSIONS

We can summarize the tenets of active, behavioral, task-oriented (in short, *animate*) vision as follows:

1. Vision does not function in isolation, but is instead a part of a complex behavioral system that interacts with the physical world.
2. General-purpose vision is a chimera. There are simply too many ways in which image information can be combined, and too much that can be known about the world for vision to construct a task-independent description.
3. Directed interaction with the physical world can permit information that is not readily available from static imagery to be obtained efficiently.
4. Vision is dynamic; fast vision processing means that the world can serve as its own database, with the system retrieving relevant information by directing gaze or attention.
5. Vision is adaptive; the functional characteristics of the system may change through interactions with the world.
6. Vision requires several concurrently cooperating layers of functionality with different degrees of potential parallelism.

It has been our experience that an integrated architecture for animate vision must support multi-model parallel programming. That is, it should be possible to select the best parallel programming model for each task, and then to integrate those tasks into a single application easily.

In this paper we described how the Psyche operating system supports multi-model programming through mechanisms for first-class user-level threads and model-independent communication. We then illustrated the use of those mechanisms in the implementation of a checkers playing robot. In the Checkers Player, different programming models were used for vision (fine-grain processes using shared memory), robot motion planning (Multilisp futures), and strategy (coarse-grain processes using message passing). Tasks from different programming models are able to synchronize and communicate using shared data structures.

A major conclusion of this work is that multi-model parallel programming has significant software engineering advantages. Our use of multiple models in the Checkers Player was not an artificial constraint, but instead was a reasoned choice based on the tasks to be performed, the expertise of the people involved, the available software, and the available programming environments. By extending the well-known software engineering principle of modularity to include different parallel programming environments, we increase the expressive power, reusability, and efficiency of the resulting code, and thereby simplify the construction of complex, animate vision systems.

REFERENCES

1. Aloimonos, Y., and Shulman, D. *Integration of Visual Modules.* Academic Press, New York, 1989.
2. Anderson, T. E., Bershad, B. N., Lazowska, E.D., and Levy, H. M. Scheduler activations: Effective kernel support for the user-level management of parallelism. *ACM Trans. Comput. Systems,* **10,** 1 (Feb. 1992), 53–79.
3. Ballard, D. H., and Ozcandarli, A. Real-time kinetic depth. In *Proc. Second International Conference on Computer Vision,* Nov. 1988, pp. 524–531.
4. Ballard, D. H. Animate vision. *Artif. Intell.* **48,** 1 (Feb. 1991), 57–86.
5. BBN Advanced Computers, Inc. *Chrysalis Programmers Manual, Version 4.0.* Cambridge, MA, Feb. 1988.
6. Bershad, B. N., Anderson, T. E., Lazowska, E. D., and Levy, H. M. Lightweight remote procedure call. *ACM Trans. Comput. Systems,* **8,** 1 (Feb. 1990), 37–55.
7. Bershad, B. N., Ching, D. T., Lazowska, E. D., Sanislo, J., and Schwartz, M. A remote procedure call facility for interconnecting heterogeneous computer systems. *IEEE Trans. Software Engrg.* **SE-13,** 8 (Aug. 1987), 880–894.
8. Birrell, A. D. and Nelson, B. J. Implementing remote procedure calls. *ACM Trans. Comput. Systems,* **2,** 1 (Feb. 1984), 39–59.
9. Bisiani, R., and Forin, A. Multilanguage parallel programming of heterogeneous machines. *IEEE Trans. Comput.* **37,** 8 (Aug. 1988), 930–945.
10. Black, D. L. Scheduling support for concurrency and parallelism in the Mach Operating System. *Computer* **23,** 5 (May 1990), 35–43.
11. Brown, C. M., Fowler, R. J., LeBlanc, T. J., Scott, M. L., Srinivas, M., Bukys, L., Costanzo, J., Crowl, L., Dibble, P., Gafter, N., Marsh, B., Olson, T., and Sanchis, L. DARPA parallel architecture benchmark study. Butterfly Proj. Rep. 13, Computer Science Department, University of Rochester, Oct. 1986.
12. Brown, C. M., Olson, T., and Bukys, L. Low-level image analysis on a MIMD architecture. *Proc. of the First IEEE International Conference on Computer Vision,* London, June 1987, pp. 468–475.
13. Coombs, D. J., and Brown, C. Cooperative gaze holding in binocular vision. *IEEE Control Systems,* **11,** 4 (June 1991), 24–33.
14. Edler, J., Lipkis, J., and Schonberg, E. Process management for highly parallel UNIX systems. *Proc. of the USENIX Workshop on UNIX and Supercomputers,* 1988.
15. Halstead, R. H., Jr. Multilisp: A language for concurrent symbolic computation. *ACM Trans. Programming Languages Systems,* **7,** 4 (Oct. 1985), 501–538.
16. Heeger, D. J. Optical flow from spatiotemporal filters. *Proc. First International Conference on Computer Vision,* June 1987, pp. 181–190.
17. Klinker, G., Shafer, S., and Kanade, T. A physical approach to color image understanding. *Internat. J. Comput. Vision,* **4,** 1 (Jan. 1990), 7–38.
18. Latombe, J. C. *Robot Motion Planning.* Kluwer Academic, Boston, 1990.
19. LeBlanc, T. J. Structured message passing on a shared-memory multiprocessor. *Proc. of the 21st Hawaii International Conference on System Science,* Kailua-Kona, HI, January 1988, pp. 188–194.
20. LeBlanc, T. J., Scott, M. L., and Brown, C. M. Large-scale parallel programming: Experience with the BBN Butterfly parallel processor. *Proc. ACM/SIGPLAN PPEALS 1988,* New Haven, CT, July 1988, pp. 161–172.
21. Lee, D. N., and Lishman, J. R. Visual control of locomotion. *Scand. J. Psych.* **18** (1977), 224–230.
22. Marsh, B. D. Multi-model parallel programming. Ph.D. thesis, Computer Science Department, University of Rochester, July 1991.
23. Marsh, B. D., Scott, M. L., LeBlanc, T. J., and Markatos, E. P. First-class user-level threads. *Proc. 13th Symposium on Operating Systems Principles.* Pacific Grove, CA, Oct. 1991, pp. 110–121.
24. Nelson, R. C. and Aloimonos, J. Obstacle avoidance using flow field divergence. *IEEE Trans. Pattern Anal. Mach. Intell.* **11,** (1989), 1102–1106.
25. Pentland, A. P. Shape from shading: A theory of human perception. *Proc. Second International Conference on Computer Vision,* Nov. 1988.
26. Rimey, R. D., and Brown, C. M. Where to look next using a Bayes net: An overview. *DARPA Image Understanding Workshop Proceedings,* Feb. 1992.
27. Rimey, R. D., and Brown, C. M. Where to look next using a Bayes net: Incorporating geometric relations. *European Conference on Computer Vision,* May 1992.
28. Scott, M. L. The Lynx distributed programming language: Motivation, design, and experience. *Comput. Languages,* **16,** 3/4 (1991), 209–233.
29. Scott, M. L., and Jones, K. R. Ant Farm: A lightweight process programming environment. Butterfly Proj. Rep. 21, Computer Science Department, University of Rochester, August 1988.
30. Scott, M. L., LeBlanc, T. J., and Marsh, B. D. Design rationale for Psyche, a general-purpose multiprocessor operating system. *Proc. of the 1988 International Conference on Parallel Processing, Vol. II–Software,* August 1988, pp. 255–262.
31. Scott, M. L., LeBlanc, T. J., and Marsh, B. D. Evolution of an operating system for large-scale shared-memory multiprocessors. TR 309, Computer Science Department, University of Rochester, Mar. 1989.
32. Scott, M. L., LeBlanc, T. J., and Marsh, B. D. Multi-model parallel programming in Psyche. *Proc. of the Second ACM/SIGPLAN Symposium on Principles and Practice of Parallel Programming.* Seattle, WA, March 1990, pp. 70–78.
33. Thomas, R. H., and Crowther, W. The Uniform System: An approach to runtime support for large scale shared memory parallel processors. *Proc. of the 1988 International Conference on Parallel Processing, Vol. II–Software.* Aug. 1988, 245–254.
34. Weems, C. C., Hanson, A. R., Riseman, E. M., and Rosenfeld, A. An integrated image understanding benchmark: Recognition of a $2\frac{1}{2}$-D Mobile. *Proc. of the IEEE Conference on Computer Vision and Pattern Recognition,* June 1988.
35. Weiser, M., Demers, A., and Hauser, C. The Portable Common Runtime approach to interoperability. *Proc. 12th Symposium on Operating Systems Principles,* Litchfield, AZ, Dec. 1989, pp. 114–122.
36. Wixson, L. E., and Ballard, D. H. Real-time detection of multi-colored objects, *SPIE Symp. on Advances in Intelligent Robotics Systems.* Nov. 1989, pp. 435–446.
37. Yeshurun, Y., and Schwartz, E. L. Cepstral filtering on a columnar image architecture: A fast algorithm for binocular stereo segmentation. Robotics Research Report 286, Courant Institute, New York, 1987.

BRIAN MARSH is a research scientist at the Matsushita Information Technology Laboratory in Princeton, New Jersey. His research interests include multiprocessor and distributed operating systems. Marsh

received his M.S. and Ph.D. degrees in computer science from the University of Rochester in 1988 and 1991.

CHRIS BROWN is a professor in the computer science department of the University of Rochester. His research interests include geometric invariance, cognitive and reflexive gaze control, and integration of computer vision, robotics, and parallel computation into active intelligent systems. Brown received his Ph.D. in information sciences from the University of Chicago in 1972.

TOM LEBLANC is an associate professor in the computer science department at the University of Rochester. His research interests include parallel programming environments and multiprocessor operating systems. LeBlanc received his Ph.D. in computer science from the University of Wisconsin at Madison in 1982.

MICHAEL SCOTT is an associate professor in the computer science department at the University of Rochester. His research focuses on programming languages, operating systems, and program development tools for parallel and distributed computing. Scott received his Ph.D. in computer science from the University of Wisconsin at Madison in 1985.

Received April 25, 1991; revised December 9, 1991; accepted January 1992

TIM BECKER is on the technical staff of the computer science department at the University of Rochester. His recent work has included projects in computer vision and robotics, and the implementation of the Psyche multiprocessor operating system. Becker received his B.S. in industrial engineering from the Pennsylvania State University in 1980.

CESAR QUIROZ is a software engineer with EXELE Information Systems, East Rochester, NY. His research interests are centered around the study of parallelism in programming language implementation, especially the parallelization of imperative code. Quiroz received his Ph.D. in computer science from the University of Rochester in 1991.

PRAKASH DAS is a system designer at Transarc Corporation in Pittsburgh. His research interests include multiprocessor operating systems. Das received his M.S. in computer science from the University of Rochester in 1991.

JONAS KARLSSON is a graduate student in the computer science department at the University of Rochester. His research interests include multiagent planning and robot motion planning. Karlsson received his B.S. in computer science from Stanford University in 1990.