The Rochester Checkers Player: Multi-Model Parallel Programming for Animate Vision

B. Marsh, C. Brown, T. LeBlanc, M. Scott,
T. Becker, P. Das, J. Karlsson, C. Quiroz

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

Technical Report 374

June 1991

Abstract

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. No single model of parallel programming is likely to serve for all tasks, however. Early vision algorithms are intensely data parallel, often utilizing fine-grain parallel computations that share an image, while cognition algorithms decompose naturally by function, often consisting of loosely-coupled, coarse-grain parallel units. 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 describe *Checkers*, a multi-model parallel program that implements a robot checkers player. Checkers runs on the BBN Butterfly multiprocessor under the Psyche operating system, which we designed to support multi-model programming. Checkers visually monitors a standard checkerboard, decides on a move in response to a move by a human opponent, and moves its own pieces. Different parallel programming models are used for vision, robot motion planning, and strategy. Tasks from different programming models are able to synchronize and communicate using a shared checkerboard data structure. The Checkers implementation, which required only two months of part-time effort by five people, demonstrates many of the advantages of multi-model programming (e.g., expressive power, modularity, code reuse, efficiency), and the importance of integration through shared data abstractions and customized communication protocols accessible from every parallel programming model.

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 is supported by a DARPA/NASA Graduate Research Assistantship in Parallel Processing. The government has certain rights in this material.

1 Introduction

Vision can be viewed as a passive, observation-oriented activity, or as one intimately related to action (e.g., manipulation, navigation). 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. In *passive vision* systems the camera providing the image input is immobile. In *active vision* systems observer-controlled input sensors are used [2]. 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 [9]. Reconstructionist vision appears to be nearing its limits however, without reaching its goal.

An alternative to reconstructionist vision derives from the observation that biological systems do not, in general, seem to 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. *Animate* vision researchers, inspired by successful biological systems [7; 23], seek to develop practical, deployable vision systems by discovering and exploiting principles that link perception and action. These systems, although technically possible due to recent advances in real-time image analysis and robotics hardware, require extremely complex software to integrate vision, planning, and action.

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. The type of parallelism we would like to exploit varies among tasks in the system however, and no single model of parallel computation is likely to suffice for *all* tasks.

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.

The central claim of this paper is that an integrated vision architecture must support multiple models of parallelism. We support this claim by describing a complex animate vision application, Checkers, constructed as a multi-model program. Checkers runs on the BBN Butterfly multiprocessor under the Psyche operating system, which was designed to support multi-model programming. Checkers visually monitors a standard checkerboard, decides on a move in response to a move by a human opponent, and moves its own pieces. Different parallel programming models are 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 a shared checkerboard data structure. The Checkers implementation, which required only two months of part-time effort by five people, demonstrates many of the advantages of multi-model programming (e.g., expressive power, modularity, code reuse,

- Vision does not function in isolation, but is instead a part of a complex behavioral system that interacts with the physical world.
- 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.
- Directed interaction with the physical world can permit information that is not readily available from static imagery to be obtained efficiently.
- Vision is dynamic; fast vision processing implies the world can serve as its own database, with the system retrieving relevant information by directing gaze or attention.
- Vision is adaptive; the functional characteristics of the system may change through interaction with the world.

Figure 1: Animate Vision Claims

efficiency), and the importance of integration through shared data abstractions and customized communication protocols accessible from every parallel programming model.

We first describe a typical architecture for animate vision systems, including both the hardware environment and software requirements. Next we present an overview of the Psyche operating system. We then describe the implementation of our checkers player in detail, emphasizing the use of several different parallel programming environments, and the integration of tasks in the implementation. We conclude with a discussion of our experiences with Checkers and multi-model programming.

2 An Architecture for Animate Vision Systems

Systems for animate vision have at least three components: sensor input, cognition, and action. The goal of our work is to facilitate the integration of these components, both at the hardware and software level.

2.1 Hardware Environment

Animate vision systems are likely to include movable, computer-configurable sensors, and sophisticated effectors or mobile vehicles. Our particular laboratory consists of five key components: a binocular head containing movable cameras for visual input, a robot arm that supports and moves the head, a Utah dextrous manipulator, a special-purpose parallel processor for high-bandwidth low-level vision processing, and a general-purpose MIMD parallel processor for high-level vision and planning.

The head has two movable greyscale 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-eye platform; separate motors control each camera's pan angle, providing independent vergence control. The controllers allow sophisticated velocity and position commands and data read-back.

Passive Vision	Animate Vision
A fixed camera may not have an object in view.	*Animate vision can use physical search*, by navigation or manipulation, changing intrinsic or extrinsic camera parameters.
Static camera placement results in nonlinear, ill-posed problems.	*Known, controlled camera movements* and active knowledge of camera placement provide self-generated constraints that simplify processing.
Stereo fusion is intractable.	*An actively verging system* simplifies stereo matching.
A single fixed camera imposes a single, possibly irrelevant, coordinate system.	*Animate 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.	*Variable camera parameters* can compensate for range, provide varying depth of field, and indirectly give information about the physical world.
Segmentation of static, single images is a known intractable problem.	*Gaze control helps segmentation:* active vergence or object tracking can isolate visual phenomena in a small volume of space, simplifying grouping.

Table 1: Computational Features of Passive and Animate Vision Systems.

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

Our dextrous manipulator is a 16 degrees-of-freedom Utah hand, which can require up to 96 parallel input channels and 64 parallel output channels for control and sensing. The Utah hand was not used in our implementation of Checkers; we used a passively compliant checker-pushing tool (referred to as the robot's *nose*) instead.

The MaxVideo system consists of several independent boards that can be connected together to achieve a wide range of frame-rate image analysis capabilities. The MaxVideo boards are all register programmable and can be controlled via VME bus. The ZEBRA [22] and ZED programming systems, developed at Rochester, make this hardware easily and interactively programmable.

An important feature of our laboratory is the use of a multiprocessor as the central computing resource. Our Butterfly Plus Parallel Processor has 24 nodes, each consisting of an MC68020 processor with hardware floating point and memory management units and 4 MBytes of memory. The Butterfly is a shared-memory multiprocessor; each processor may directly access any memory in the system, although local memory is roughly 12 times faster to access than non-local memory. The Butterfly has a VME bus connection that mounts in the same card cage as the MaxVideo and motor controller boards. The Butterfly also has a serial port on each board: we use the port to communicate directly with the VAL robot controller software on its dedicated LSI-11. A Sun4/330 workstation acts as a host for the system.

In the aggregate, our hardware provides a set of powerful peripherals, each of which may have intelligent controllers, attached to a single multiprocessor. The flexibility and power inherent in this architecture allow us to implement all the control functions needed in our Checkers example, as well as more complex applications.

2.2 Software Requirements

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

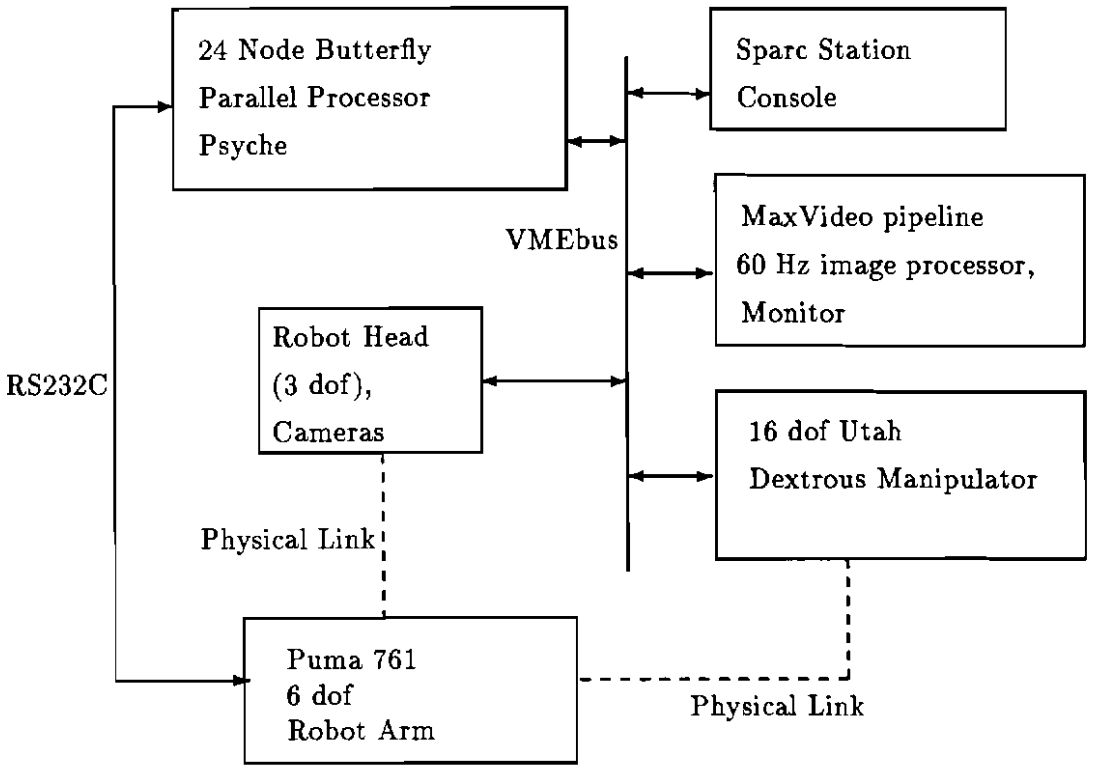


Figure 2: Rochester Hardware Architecture for Animate Vision.

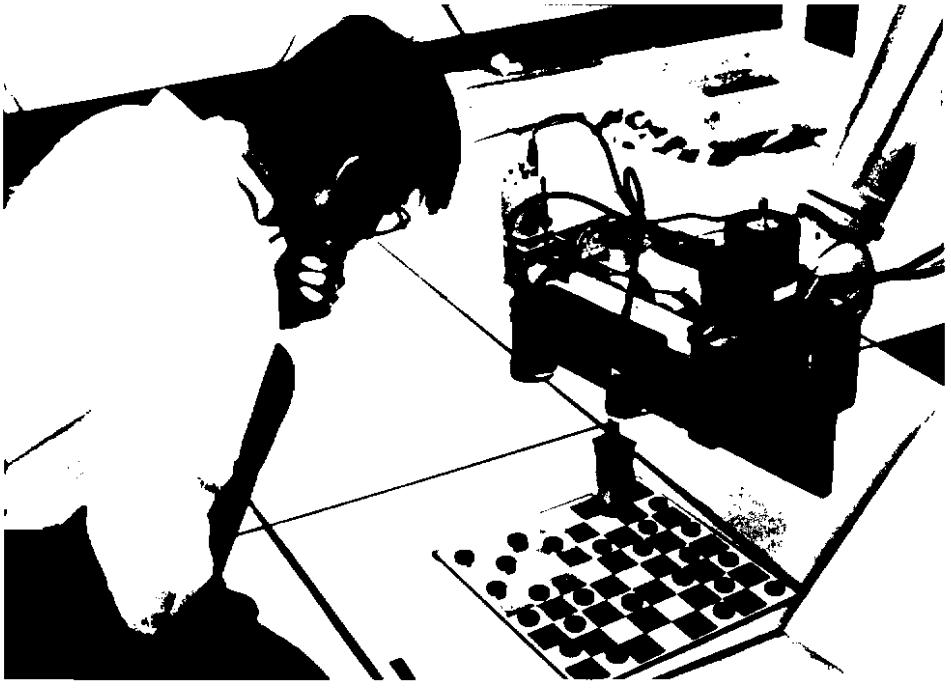


Figure 3: In this configuration the robot head has one large color camera and two small greyscale cameras, a single tilt motor, twin pan motors, and a passively compliant checker-pushing tool.

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 [6; 24] 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 benchmark contained a suite of noninteracting routines for low- and high-level vision tasks. The low-level vision routines required manipulation of 2-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 [21]. We were able to show that pixel-level data-parallel functionality could be implemented on the Butterfly with nearly linear speedup given additional processors [15]. (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 and communication than is provided by the Uniform System. To implement these functions we used two parallel programming environments developed at Rochester: a message-passing library (SMP) [11] and a parallel programming language (Lynx) [16]. 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 lead to the subsequent development of a new programming environment called Ant Farm [17].

Each of these models (Uniform System, SMP, Lynx, Ant Farm) was chosen because it made programming a 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 individual solutions were implemented using several different programming models and we lacked the system support necessary to integrate the various models. 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.

The DARPA benchmarks and our other applications experience showed the potential advantages of using a large-scale MIMD multiprocessor as the controlling architecture in integrated animate vision systems. Our experiences also demonstrated 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 Psyche Operating System Overview

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 have provided 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 have been 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 have all been used for communication and synchronization. These communication 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 frequency, 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 attempting 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 beneath many parallel programming models, in many 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 [3] as a low-level interface, we created several new programming libraries and languages, and ported several others [12]. 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 benchmarks, 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 [18; 19; 20] 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, 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 run-time 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 is 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 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 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 fairly among 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 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 represented in many different ways (e.g., as lightweight threads of various kinds or as requests on the queue of a server).

Within a protection domain, each process maintains a data structure shared with the kernel containing pointers to process management functions, such as *block* and *unblock* routines. The kernel never calls these operations itself, but identifies them in the kernel/user data area, so that user-level code can invoke them directly. Dissimilar types of processes can use these primitive operations to build scheduler-based synchronization mechanisms, such as semaphores.

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 in its domain at the current time, 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* and *process-independent communication* [13; 14]. First-class user-level threads enjoy the functionality of traditional kernel processes, while retaining the efficiency and flexibility of being implemented outside the kernel. Process-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. Different kinds of threads, in separate but overlapping address spaces, can synchronize access to shared data structures. 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 non-blocking 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 cheaply (in both directions). These data structures indicate (among other things) the state of the currently executing process, the address of a preallocated stack to be used when handling software interrupts, and a collection of variables for managing the behavior of software interrupts. User-writable data can be used to specify what ought to happen in response to kernel-detected events. By allowing the kernel and user-level code to share data, changes in desired behavior can occur frequently (for example when context switching in user space).

3.4 Process-Independent Communication

Shared memory is a viable communication medium between 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 extremely 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 one or more realms and is only accessible using realm invocations. Invocations (i.e., procedure calls) are a communication mechanism, providing for 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 parallel programming model provides an interface to its process management 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.

These mechanisms can be used to implement two different types of interactions between dissimilar programming models: (1) shared data structures and (2) direct invocations from one programming model to another. Shared data structures are typically passive; the associated management code is

executed only when a process invokes an operation on the data structure. A direct invocation from one programming model to another causes a process to move from one programming environment into the runtime environment of another programming model. Subject to certain limitations, the process can then perform operations available to processes of the new environment. We now describe how both varieties of interactions are used in our implementation of Checkers.

4 A Multi-Model Robot Checkers Player

Checkers is a multi-model vision application implemented on top of Psyche. A checkers–playing robot conducts a game of checkers against a human opponent, cyclically sensing the opponent's move, and then planning and executing its response.

An inexpensive, standard-sized checkers game is used. The board squares are 46mm on a side; the pieces, made of light plastic, are 30mm in diameter and 5mm in height. The position of the board is fixed in the coordinate system of the robot. Normal fluorescent lighting is used. The camera's internal parameters (aperture and focus) are manually adjusted before the game, and the external parameters (the exact positions of the pan and tilt motors, and the robot's position) are compensated by an initial calibration procedure (finding pixel coordinates of the corners of the board).

The human is given the black pieces and the first move. The normal rules of play are obeyed, including multiple captures, crowning, and the extended capabilities of kings. A king is not crowned by the robot — it remembers which pieces on the board are kings. The human can either flip promoted pieces to expose an embossed crown, or stack pieces. The robot pushes pieces around the board: it does not pick them up.

During play the human player modifies the board by moving a piece. The sensing task detects the change in the board configuration and interprets it symbolically in terms of the primitive moves of checkers. In a symbolic, strategic task, the robot considers the change to be a potentially legal move by the human. The move is validated, and, if valid, accepted by the robot. Once a valid move is made by the human, the robot runs a symbolic game-playing algorithm to find its response to the human move. The board position reported by the vision subsystem and the symbolic move computed by the robot are used to plan an optimal sequence of primitive, physically-realizable actions by the effector task. When this movement plan is available, the robot arm is engaged to execute the plan. Central control, communication, data representation, and synchronization are provided by a board maintenance task. The robot emits status information, error messages, and occasional gratuitous remarks through a voice-synthesis board.

In our current implementation, the robot watches the board from a home position directly overhead. When the program begins execution, the robot goes to the home position and waits for the human to make the first move. A few seconds after the human's move, the robot head descends, acquires a piece with its passively-compliant effector, and pushes it. During a capture it pushes the captured piece(s) off the board and completes the move. It then returns to its home position and waits for the board to assume a new legal state as a result of the next human move. The human interacts with the program simply by playing checkers.

4.1 Parallel Programming Environments

Checkers, 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 [8] 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

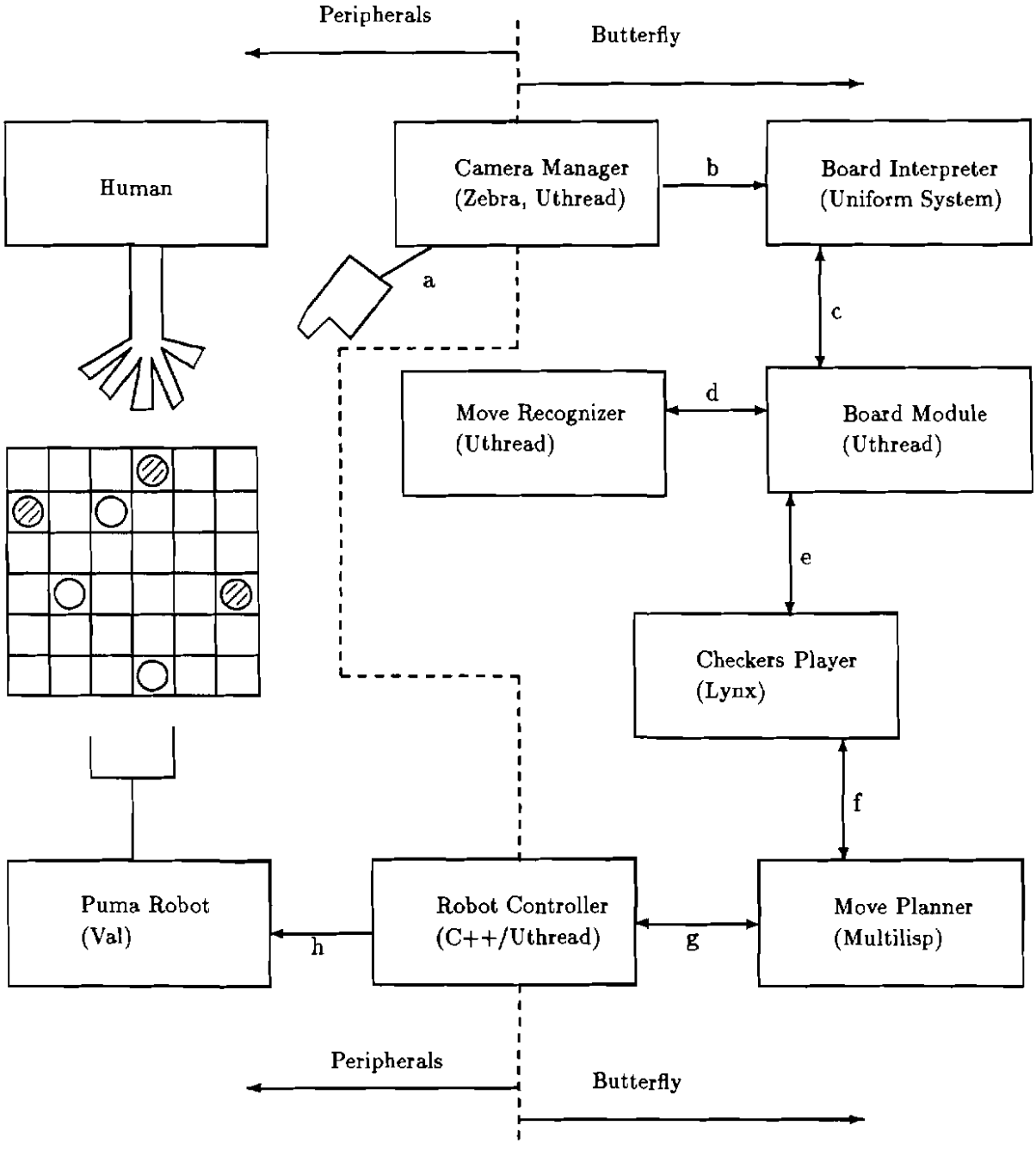


Figure 4: Functional modules and communication paths in the Checkers Player. Multiple models of parallelism (to the right of the dotted line) are implemented under Psyche on the Butterfly. Perceptual and motor modules (to the left of the dotted line) reside on the Butterfly and in peripherals.

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 [16] is a parallel programming language based on message passing; the language was designed and implemented by Michael Scott at Wisconsin, and subsequently ported to the Butterfly. Lynx programs consist of multiple heavyweight processes, each with their 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 [21] 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 that 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 tasks on all processors. Since tasks are not allowed to block, spinlocks are used for synchronization.

Uthread is a simple, lightweight thread package developed at Rochester 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.

We chose these environments for four reasons. First, each of these environments was specifically developed for a particular application domain that was a subset of our problem domain. Second, implementations of all four environments were either already available for our hardware or could be easily ported to our hardware. Third, the principals involved in the project had extensive experience with one or more of these implementations and would not have to learn a new system. Finally, we already had a software base for vision, planning, and checkers playing, composed of programs written in the Uniform System, Lisp, and Lynx, respectively.

4.2 Checkers Data Structures

The primary data structures used to implement checkers are the representations of the board and moves. A short pipeline of representations is needed to support backing up to legal or stable states. 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.

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 structured 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.

4.3 Checkers Tasks

Six different tasks cooperate to implement Checkers. Two manage the camera and robot devices; the remainder implement vision, move recognition, checkers strategy, and motion planning.

Camera Manager – a Uthread module that creates and initializes a Psyche realm for the VME memory used to control and access the MaxVideo hardware. This module registers the name and address of the realm with a name server. The Board Interpreter accesses this realm directly to retrieve an image from the MaxVideo framebuffer.

Board Interpreter – a Uniform System program that transfers an image from the Camera Manager (in VME memory) to local Butterfly memory, and produces a symbolic description of the checkers in the image. The data transfer of 0.25 Mbytes of image information over the VME bus takes 280ms. After transferring the image, the Board Interpreter segments the image into 64 board squares and analyzes each square in parallel. Each task attempts to recognize the color of its square, whether the square contains a piece, and if so, of what color. The pixels in the square are labeled as *Dark*, *White*, *Red*, or *Unrecognized* according to their grey level. (We cannot differentiate black pieces from the dark squares using grey levels, so we play on the light squares.) Each square is then labeled according to the color of the square and the color of the piece on that square (e.g., *BlackSquare*, *WhiteSquare*, *RedPieceOnWhiteSquare*, *BlackPieceOnWhiteSquare*, *Unrecognized*). For each square containing a piece, the centroid of the piece is calculated as the centroid of the relevant pixels. The center of each empty square is similarly calculated, first in pixel coordinates, and then in world coordinates using calibration information acquired at the start of execution. Once a complete interpretation containing no unrecognized squares is calculated, the Board Interpreter accepts the interpretation. If the new interpretation differs from the previous interpretation, the result is reported to the Board Module. Using four processors the Board Interpreter can interpret the image input a little more often than once every second.

Move Recognizer – a Uthread module that compares 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.

Checkers 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 multi-threaded master process manages the parallel evaluation of possible moves; slave processes perform work on behalf of the master. The master explores the first few levels of the game tree, while building the game tree data structure. At a fixed depth in the tree (typically four or five levels), the master enters board positions into the slaves' work queue. When the results come back, the master updates the game tree, performs any necessary pruning (i.e., to

throw away moves that are now known to be sub-optimal), and produces new entries for the work queue. If the work queue is ever full, the master blocks until slaves have time to remove entries. Both the quality and the speed of the game tree search are maximized when there is a slave process running on every available processor.

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. Two versions of this program have been written. The first version sequentially executes partial moves after converting the symbolic information to robot coordinates, pushing captured pieces off the board using a diagonal path through non-playing (dark) squares. The second version constructs, in parallel, artificial potential fields that have peaks reflecting square occupancies and bias reflecting the goal location (see Figure 5) [10]. 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. Since the algorithm allows pieces to be temporarily moved aside or swapped with the moving piece, it is a route-maker as well as a route-finder. The result is a set of plans, one of which is chosen based on some cost function, such as the total estimated time to complete the move, or the shortest distance to push the checker.

Robot Controller – a Uthread module that controls a serial line connection between the Butterfly and the Puma robot. The Robot Controller sends movement commands in the VAL language (equivalent to MoveTo (X,Y,Z,SPEED)) and waits for notification of successful completion.

4.4 The Implementation of Moves

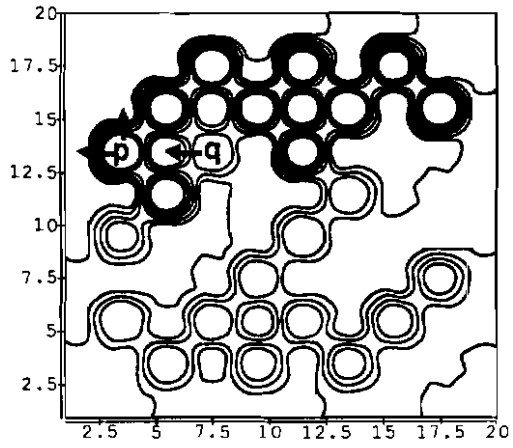
The execution of the program is implemented as a series of moves, each of which requires the cooperation of several modules and programming models. The following is a description of the events that comprise a move, including the initialization phase that occurs at the start of execution. (Control flow among the modules is indicated by the lettered arrows in Figure 4.)

An initialization procedure creates and initializes all the major modules, and enables the VME interrupts. The Checkers Player immediately invokes the Board Module to obtain the first human move in symbolic form. The Board Interpreter begins to analyze images produced by the low-level vision hardware in an attempt to recognize the first move.

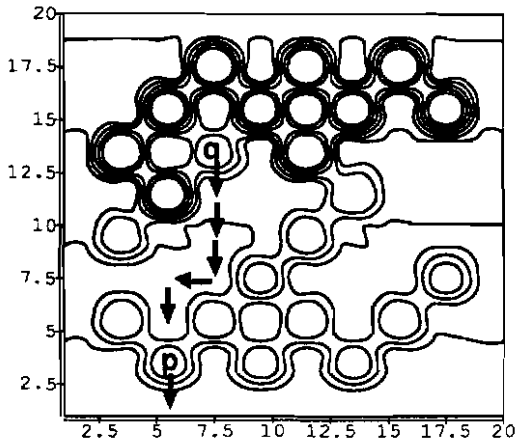
The Board Interpreter continuously receives an image from the camera (a-b) and analyzes it. If the image is the same as the previous image, or if interpretation is unsuccessful, the Board Interpreter tries again. If the board position has changed, the Board Interpreter invokes the Board Module (c) to update the board description, passing the symbolic and quantitative positions.

When the Board Module receives a new board position from the Board Interpreter, it invokes the Move Recognizer (d) to parse the difference between new and old board positions into partial checkers moves. These partial moves are stored in the Board Module to be retrieved by the Checkers Player. After a successful return from the Move Recognizer, the original invocation from the Board Interpreter to the Board Module returns, which causes the Board Interpreter to then wait for a new image to analyze.

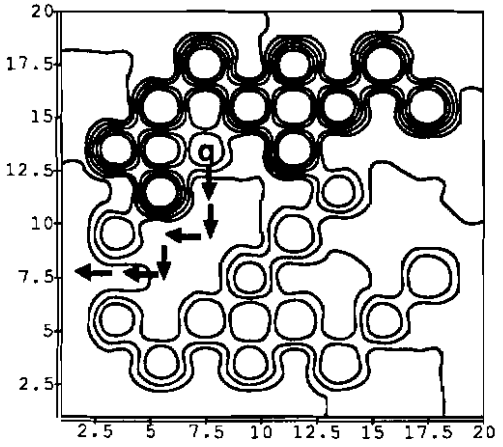
When the invocation from the Checkers Player to the Board Module (e) discovers that a new valid list of partial moves has appeared in the Board Module, it returns the first partial move to the checkers game tree evaluation program. If several partial moves are needed to complete the move, additional invocations from the Checkers Player to the Board Module (e) follow. If any of the partial moves represents an illegal move, the Checkers Player resets its internal state to the beginning of the move sequence, and flushes the current state information and list of partial moves in the Board Module. It also synchronizes with the Board Interpreter (e-c), which informs the human and produces a new board state.



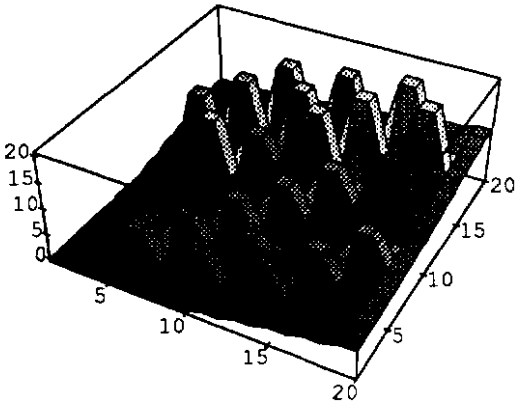
(a)



(b)



(c)



(d)

Figure 5: (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 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.

As long as the incoming list of partial moves is legal, the Checkers Player will wait for moves to appear in the Board Module. As a result, board interpretation can occur several times while the human makes a move, particularly if the move is a complex jump. The Checkers Player and Board Module interact (e) until a complete move is input. At this point the checkers-playing program runs and generates its reply to the human's move in the form of a symbolic board position. This board position is passed to the Board Interpreter, which generates a list of partial moves required to implement the differences between the updated board position and the current position.

Once the Board Interpreter has produced a set of partial moves that define the robot's response, the Checkers Player invokes the Move Planner (f) with the partial move sequence. Partial moves are passed to the Move Planner one at a time, and each one causes a sequence of low-level move commands and acknowledgements to flow back and forth between the Move Planner, the Robot Controller, and the robot (g-h).

4.5 Inter–Model Communication

The implementation of a single move illustrates two distinct styles of interactions among programming models: data structures shared between models and direct procedure calls (or invocations) between models. Both styles of interaction require synchronization between processes of different types.

The Board Module must synchronize access to data structures shared by processes from the Multilisp, Lynx, Uniform System, and Uthread environments. To access these data structures, processes call directly into the Board Module and execute the associated code. When a process must block within the Board Module, the code uses the pointer provided by the kernel to find the correct **block** and **unblock** routines for the currently executing process type. A process that must block on a semaphore first places the address of its **unblock** routine in the semaphore data structure, and then calls its **block** routine. When another process wants to release a process that is blocked on a semaphore, it simply retrieves the address of the appropriate **unblock** routine from the semaphore data structure and calls the routine. If protection between process types is desired, the appropriate rights can be stored with the address of the routines, and protected invocations can be required.

There are several advantages to using shared data abstractions for communication between models. First, since we use a simple procedural interface to access shared data, there is a uniform interface between models, regardless of the number or type of programming models involved. Second, communication is efficient because processes can use shared memory to communicate directly. Third, synchronization across models is efficient due to the underlying mechanisms for implementing synchronization (a kernel pointer to user-level process management routines, and a procedural interface to routines that block and unblock a process). Finally, although the Board Module resembles a blackboard communication structure, shared data abstractions between models can be used to implement a wide variety of communication mechanisms, including message channels and mailboxes.

A different type of interaction occurs between the Checkers Player and the Move Planner, wherein a Lynx thread calls directly into the Multilisp environment of the Move Planner. Since the Move Planner already provides exactly the functionality required by the Checkers Player, an intervening data structure would simply add unnecessary generality and overhead (such as the cost of extra invocations). Instead, every entry point exported by the Move Planner refers to a stub routine designed for invocation by processes outside the Multilisp world. This stub routine copies parameters into the Multilisp heap and dispatches a Multilisp future to execute the Lisp function associated with the invocation. After the future executes the correct Multilisp function, the Multilisp runtime calls the Lynx scheduler directly to unblock the Lynx thread.

Direct calls between arbitrary environments are often complicated by the fact that the code in each environment makes many assumptions about the representation and scheduling of processes.

Psyche facilitates direct calls between modules by separating the code that depends on the semantics of processes from the code used as an external interface. As a result, an application like Checkers can be constructed from a collection of self-contained modules, without regard to the programming model used within each module.

5 Conclusions

Several complete games have been played with Checkers. The robot plays a competent game; the quality and speed of its moves are a function of the number of processors devoted to strategy. More important than the quality of the play however, is that Checkers demonstrates the advantages of decomposing animate vision systems by function, and independently selecting an appropriate parallel programming model for each function. 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 parallel programming systems and applications. These properties add significantly to our ability to build complex animate vision applications.

The entire Checkers implementation required only two months of part-time effort by five people. Our use of multiple parallel programming models 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.

We were able to resurrect a Lynx checkers-playing program (Scott) that had been implemented years ago as a stand-alone program. The Uniform System image-analysis library was plugged into Checkers after several years of disuse. The vision tasks (Das), Move Planner (Karlsson), Board Module (Becker), and Move Recognizer (Quiroz), as well as necessary Psyche support for the particular models we used (Marsh), were all developed in parallel by people that had expertise in a particular problem domain and the related software environment. Coding these modules was a part-time activity extending over several weeks.

Integration was a full-time activity that took only a few days. During integration, we made (and subsequently changed) many decisions about which modules would communicate directly with each other, and which should use the 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.

Many improvements are possible and some are ongoing. The board interpretation module is now being modified to use color instead of grey-scale input for more robust and discriminating classification. We have written a calibration routine that locates the board precisely in the image, and thereby automatically accounts for nonrepeatability in robot positioning from day to day. We would like to investigate hand-eye coordination and recovery from imperfectly performed moves. We would also like to use our newly acquired dextrous manipulator to make moves by picking up the pieces.

We have not yet made extensive experiments to determine whether there are bottlenecks in the system, either in hardware or software. If software bottlenecks are found, we will have several options to improve performance. The allocation of processors to tasks can be changed easily, as can the communication protocols between tasks. Our ability to build the stylized data abstractions and communication protocols used in Checkers suggests that we will have little difficulty experimenting with alternatives. It is precisely this type of flexibility that is required in animate vision systems, and our experiences suggest that multi-model programming in general, and the Psyche mechanisms in particular, can provide the needed flexibility.

References

[1] Accetta, M., R. Baron, W. Bolosky, D. Golub, R. Rashid, A. Tevanian, and M. Young, "Mach: A New Kernel Foundation for UNIX Development," *PROC of the Summer 1986 USENIX Technical Conference and Exhibition*, pp. 93–112, June 1986.

[2] Aloimonos, Y. and D. Shulman, *Integration of Visual Modules*, Academic Press, 1989.

[3] BBN Advanced Computers Inc., "Chrysalis Programmers Manual Version 4.0," Cambridge, MA, February 1988.

[4] Ballard, D.H., "Animate Vision," *Artificial Intelligence*, 48(1):57–86, February 1991.

[5] Brooks, R.A., "A Robust Layered Control System for a Mobile Robot," *IEEE Journal of Robotics and Automation*, RA-2:14–23, 1986.

[6] Brown, C.M., R.J. Fowler, T.J. LeBlanc, M.L. Scott, M. Srinivas, L. Bukys, J. Costanzo, L. Crowl, P. Dibble, N. Gafter, B. Marsh, T. Olson, and L. Sanchis, "DARPA Parallel Architecture Benchmark Study," Butterfly Project Report 13, Computer Science Department, University of Rochester, October 1986.

[7] Coombs, D.J., T.J. Olson, and C. M. Brown, "Gaze Control and Segmentation," *Proc. of the AAAI-90 Workshop on Qualitative Vision*, Boston, MA, July 1990.

[8] Halstead, R., "Multilisp: A Language for Concurrent Symbolic Computation," *ACM TOPLAS*, 7(4):501–538, October 1985.

[9] Klinker, G., S. Shafer, and T. Kanade. "A Physical Approach to Color Image Understanding," *Intl. Journal of Computer Vision*, 4(1):7–38, January 1990.

[10] Latombe, J.C., *Robot Motion Planning*, Kluwer Academic Publishers, Boston 1990.

[11] LeBlanc, T.J., "Structured Message Passing on a Shared-Memory Multiprocessor," *Proc. Twenty-First Annual Hawaii Intl. Conf. on System Science*, pp. 188-194, January 1988.

[12] LeBlanc, T.J., M. L. Scott, and C. M. Brown, "Large-Scale Parallel Programming: Experience with the BBN Butterfly Parallel Processor," *Proc. of the First ACM Conf. on Parallel Programming: Experience with Applications, Languages and Systems*, pp. 161–172, July 1988.

[13] Marsh, B.D., *Multi-Model Parallel Programming*, Ph.D. Dissertation, Computer Science Department, University of Rochester, July 1991.

[14] Marsh, B.D., M.L. Scott, T.J. LeBlanc, and E.P. Markatos, "First-Class User-Level Threads," *Proc. of the 13th Symp. on Operating Systems Principles*, October 1991 (to appear).

[15] Olson, T.J., L. Bukys, and C.M. Brown, "Low Level Image Analysis on an MIMD Architecture," *Proc. First Intl. Conf. on Computer Vision*, pp. 468-475, June 1987.

[16] Scott, M.L., "Language Support for Loosely-Coupled Distributed Programs," *IEEE Transactions on Software Engineering*, SE-13(1):88–103, January 1987.

[17] Scott, M.L. and K.R. Jones, "Ant Farm: A Lightweight Process Programming Environment," Butterfly Project Report 21, Computer Science Department, University of Rochester, August 1988.

[18] Scott, M.L., T. J. LeBlanc, and B. D. Marsh, "Design Rationale for Psyche, a General-Purpose Multiprocessor Operating System," *Proc. of the 1988 Intl. Conf. on Parallel Processing*, Vol. II — Software, pp. 255–262, August 1988.

[19] Scott, M.L., T. J. LeBlanc, and B. D. Marsh, "Evolution of an Operating System for Large-Scale Shared-Memory Multiprocessors," Technical Report 309, Computer Science Department, University of Rochester, March 1989.

[20] Scott, M.L, T.J. LeBlanc, and B.D. Marsh, "Multi-Model Parallel Programming in Psyche," *Proc. of the 2nd ACM Conf. on Principles and Practice of Parallel Programming*, pp. 70–78, March 1990.

[21] Thomas, R.H. and A.W. Crowther, "The Uniform System: An Approach to Runtime Support for Large Scale Shared Memory Parallel Processors," *Proc. of the 1988 Intl. Conf. on Parallel Processing*, Vol. II — Software, pp. 245-254, August 1988.

[22] Tilley, D.G., "Zebra for MaxVideo: An Application of Object-Oriented Microprogramming to Register Level Devices," Technical Report 315, Computer Science Department, University of Rochester, November 1989.

[23] Waxman, A.M., W.L. Wong, R. Goldenberg, S. Bayle, and A. Baloch, "Robotic Eye-Head-Neck Motions and Visual Navigation Reflex Learning Using Adaptive Linear Neurons," *Neural Networks Supplement: Abstracts of 1st INNS Meeting*, 1:365, 1988.

[24] Weems, C.C., A.R. Hanson, E.M. Risemsn, and A. Rosenfeld, "An Integrated Image Understanding Benchmark: Recognition of a 2 1/2 - D Mobile," *Proc. IEEE Conf. on Computer Vision and Pattern Recognition*, June 1988.