The Rochester Checkers Player

Multimodel Parallel Programming for Animate Vision

Brian Marsh, Chris Brown, Thomas LeBlanc, Michael Scott, Tim Becker, Cesar Quiroz, Prakash Das, and Jonas Karlsson
University of Rochester

Checkers, a complex real-time application, demonstrates the advantages of decomposing animate vision systems by function and independently selecting an appropriate parallel-programming model for each function.

Vision can be viewed as a passive, observational activity, or as one intimately related to action (for example, manipulation, navigation). In *passive vision* systems the camera providing the image input is immobile. In *active vision* systems observer-controlled input sensors are used.[1] Active vision results in much simpler and more robust vision algorithms, as outlined in Table 1.

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. 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.[2] 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, seek to develop practical, deployable vision systems by discovering and exploiting principles that link perception and action. Animate systems use active vision and are structured as vertically integrated skills or behaviors, rather than as visual modules that try to reconstruct different aspects of the physical world.

Despite the computational simplifications of the animate vision paradigm, 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. Low-level-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

0018-9162/92/0200-0012$03.00 © 1992 IEEE

searched in parallel. Thus, finding parallelism in the application is easy. However, the type of parallelism we would like to exploit varies among tasks in the system, 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 to express it. Each environment 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 task may require a different parallel-programming model, and all must cooperate to achieve the desired behavior. These *multimodel* programs require an underlying software system that supports several different models of parallel computation simultaneously and also allows tasks implemented in different models to interact.

We believe that to exploit fully the parallelism inherent in animate vision systems, an integrated vision architecture must support multiple models of parallelism. To support this claim, we first describe the hardware base of a typical animate vision laboratory and the software requirements of applications. We then briefly overview the Psyche operating system, which we designed to support multimodel programming. Finally, we describe a complex animate vision application, Checkers, constructed as a multimodel program under Psyche.

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 provide mechanisms for the efficient integration of these components at both the hardware and software levels.

Hardware environment. Animate vision systems require movable, computer-configurable sensors, sophisticated effectors or mobile vehicles, and several high-bandwidth computing devices. Our hardware currently consists of six key components:

- 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,
- general-purpose MIMD (multiple instruction, multiple data) parallel processors,
- a dextrous manipulator, and
- a Data Glove input device.

Table 1. Computational features of passive and active vision systems.

Passive Vision	Active Vision
A fixed camera may not have an object in view.	Active 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.	Active vision can generate and use exocentric coordinate frames, which yield more robust quantitative and qualitative algorithms and serve 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.

Tenets of animate vision

- Vision does not function in isolation, but is instead part of a complex behavioral system that interacts with the physical world.
- General-purpose vision is a chimera. There are simply too many ways to combine image information and too much to know about the world for vision to construct a task-independent description.
- Directed interaction with the physical world can permit information not readily available from static imagery to be obtained efficiently.
- Vision is dynamic. Fast vision processing implies that 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.

The head, shown in Figure 1, has two movable gray-scale CCD (charge-coupled device) 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-eyed platform. Separate motors control each camera's pan angle, providing independent vergence control. The controllers allow sophisticated velocity and position commands and data readback.

The robot body is a Puma 761 six-degrees-of-freedom arm with a 2-meter-radius workspace and a top speed of about 1 meter per second. It is controlled by a dedicated Digital Equipment Corp. LSI-11 computer implementing the proprietary Val execution monitor and programming interface.

The MaxVideo system consists of several independent boards that can be connected to achieve a wide range of frame-rate image-analysis capabilities. The MaxVideo boards are all register programmable and can be controlled via a VMEbus. The Zebra and Zed programming systems, developed at the University of Rochester, make this hardware easily and interactively programmable.

An important feature of our laboratory is the use of a shared-memory multiprocessor as the central computing resource. Checkers, our animate vision application, uses a 32-node BBN Butterfly Plus parallel processor. Each node contains a Motorola MC68020 processor with floating-point hardware and 4 Mbytes of memory. The Butterfly is a shared-memory multiprocessor. Each processor can directly access any memory in the system, although local memory is roughly 12 times faster to access than nonlocal memory. The Butterfly has a VMEbus connection that mounts in the same card cage as the MaxVideo and motor controller boards, replacing a processor in the physical address space of the multiprocessor. 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 Sun 4/330 workstation acts as a host for the system.

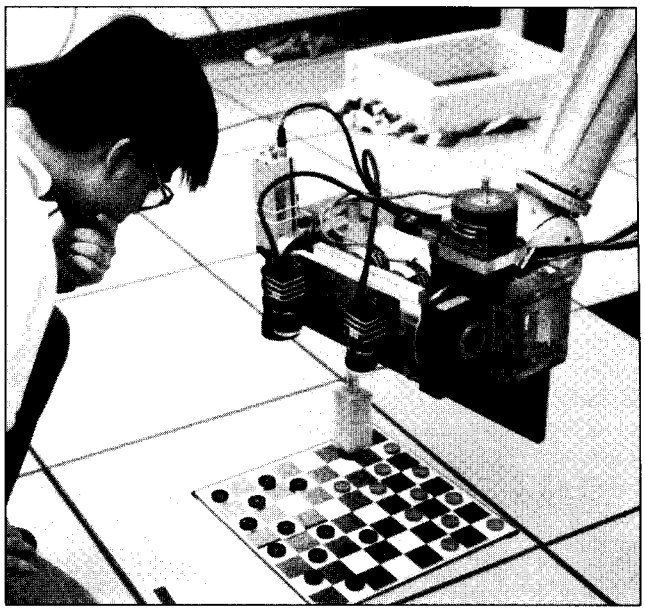


Figure 1. In this configuration, the robot head has one large color camera and two small gray-scale cameras, a single tilt motor, twin pan motors, and a passively compliant checker-pushing tool.

Several components have only recently been installed in our laboratory and therefore were not used in Checkers. These include an array of eight transputers for real-time control, a 16-degrees-of-freedom Utah hand, and a Data Glove used to gather manipulation data from humans and for teleoperation of the Utah hand.

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 action provide another source. The real issue is how to harness the application's inherent parallelism without being overwhelmed by the complexity of the resulting software. Our experiences with two DARPA benchmarks for parallel computer vision[3] 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 contains a suite of noninteracting routines for low- and high-level-vision tasks. The low-level-vision routines require manipulation of two-dimensional pixel arrays using data parallelism, conveniently accomplished using the SIMD (single instruction, multiple data) style of computation provided by the Uniform System library from BBN.[4] The functions for high-level vision require coarser grain parallelism and communication, for which we used two parallel-programming environments developed at Rochester: a message-passing library and a parallel-programming language (Lynx[5]). Each environment made programming a particular application easier in some way.

The second benchmark calls for an integrated scene-describing system. This benchmark emphasizes integration of several levels of image understanding to describe a scene of polygons at various discrete depths. It thus underscores the usefulness of a unified approach to multimodel parallelism. Unfortunately, we implemented the parts of our scene-describing system using several different programming models, and we lacked the system support necessary to integrate the various models. Processes in our MIMD computations could not directly access the data structures produced by our SIMD computations, and 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.

Multimodel programming in Psyche

Psyche is a multiprocessor operating system designed to support multimodel

programming.[6] The abstractions provided by the Psyche kernel allow user-level runtime libraries to implement customized programming models, with each library retaining full control over how its processes are represented and scheduled, and how they synchronize with the processes implemented by other libraries.[7] The kernel is responsible for coarse-grain resource allocation and protection, while runtime libraries implement short-term scheduling and process management.

A multimodel program consists of a set of modules, each of which may implement a (potentially different) programming model. Each module defines a set of interface procedures used to access the code and data encapsulated by the module. In addition, each module that implements a programming model defines a set of process-management routines used to control the behavior of the model's processes. To communicate between modules, and hence between different programming models, processes call interface procedures, which may in turn call process-management routines. The two key aspects of this approach are

• *Procedural access to shared data.* The interface procedures of a module implement a communication protocol. Shared data in the module represents the state of the protocol and is manipulated by the procedures defined in the interface. By invoking interface procedures, processes take on the communication style provided by the called module. Interface procedures allow processes to gain access to arbitrarily complex styles of communication through a mechanism — the procedure call — found in every model.

• *Dynamically bound process management.* The process-management routines of a module allow interface procedures to tailor their behavior to different programming models. Interface procedures call process-management routines whenever they must create, destroy, block, or unblock a process. They need not embody assumptions about any particular model. Among other things, process-management routines are required for scheduler-based synchronization.

To facilitate the use of shared data and procedures, Psyche arranges for every module 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. It also allows processes to call interface procedures regardless of the current address space. Depending on the degree of protection desired, a call to an interface procedure can be as fast as a normal 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 in Psyche, with the native architecture's jump-to-subroutine instruction. In some cases this instruction generates a page fault, allowing the kernel to intervene.

To implement a programming model, the user-level library must manage a set of virtual processors, which are kernel-provided abstractions of the physical processors. The kernel delivers a software interrupt to a virtual processor whenever it detects an event that might require the library to take immediate action. Events of this sort include program faults, the imminent end of a kernel-scheduling quantum, the need to block during a system call, and the initiation (via page fault) of a protected invocation. Data structures shared between the kernel and the user (writable in part by the user) allow the library code to control the behavior of the software interrupt mechanism. They also provide a location in which to store the addresses of process-management routines.

When creating a virtual processor, the user-level library specifies the location of a data structure describing that virtual processor. The kernel maintains a pointer to this descriptor (distinct on every physical processor) among the data structures shared between the kernel and the user. By convention, the descriptor in turn contains the address of a vector of pointers to process-management routines. When multiprogramming virtual processors on top of a single physical processor, the kernel changes the pointer to the virtual processor descriptor on every context switch. When multiprogramming user-level processes on top of a single virtual processor (or on a collection of virtual processors), the user changes the address of the vector of process-management routines on every context switch. As a result, an interface procedure can always find the process-management routines of the currently executing process without knowing the origin or representation of the process.

Using interface procedures and process-management routines, Psyche programmers have developed two distinct idioms for interaction between dissimilar programming models. Both of these idioms appear in Checkers. In the first, a module encapsulates a passive shared data structure that is accessed by other modules containing different process types. In the second, a process from one model calls directly into a module that implements a different model. This second approach occurs both when calling an interface procedure directly and when implementing synchronization inside an interface procedure.

When a process needs to wait for some condition while executing in an interface procedure, the code can follow pointers in well-known locations to find the addresses of the process-management routines of the process's programming model. It can then save the address of the *unblock* routine in a shared data structure and call the *block* routine. Later, when another process establishes the condition, it can retrieve the pointer to the unblock routine and call into the module that manages the waiting process, causing that process to unblock.

When calling an interface procedure or process-management routine, a process must obey certain well-defined constraints to avoid interfering with the correct operation of the host environment. In the general case, it may be necessary to create a native process to obtain the full capabilities of that environment.

Multimodel robot checkers player

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

An inexpensive, standard-size checkers game 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 adjusted by an ini-

tial calibration procedure (finding pixel coordinates of the corners of the board).

The normal rules of play are obeyed, including multiple captures, crowning, and the extended capabilities of kings. 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 robot validates the move and, if it is valid, accepts it. Once the human makes a valid move, the robot runs a symbolic game-playing algorithm to find its response to the human move. The effector task uses the board position reported by the vision subsystem and the computed symbolic move to plan a sequence of primitive, physically realizable actions. When this movement plan is available, the robot arm is engaged to execute the plan. A board maintenance task provides central control, communication, data representation, and synchronization. The robot emits status information, error messages, and occasional gratuitous remarks through a voice-synthesis board. A complete robot move, including sensing and commentary, takes about 6 seconds.

Parallel-programming environments. Checkers, like many animate vision applications, consists of tasks to implement sensing, planning, and action. We implemented each of these functions using a different parallel-programming environment: Multilisp, Lynx, the Uniform System, or Uthread.

The unit of parallelism in Multilisp is the future,[8] which is a handle for the future evaluation of an arbitrary S-expression. 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 — are used to build parallel Lisp programs.

Lynx programs consist of multiple heavyweight processes, each with its own address space.[5] The processes exchange messages using named communication channels (*links*). Each heavyweight process consists of multiple lightweight threads of control that communicate using shared memory. Condition variables are used for synchronization between threads in the same process. Synchronous message passing provides synchronization between threads in different processes.

The Uniform System[4] is a shared-memory, data-parallel programming environment. Task generators 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. Spin locks are used for synchronization.

Uthread is a simple, lightweight thread package 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:

(1) Each was specifically developed for a particular application domain that was a subset of our problem domain.

(2) Implementations of all four environments were either already available for our hardware or could be easily ported to our hardware.

(3) 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.

(4) We already had a software base for vision, planning, and checkers playing, composed of programs written in the Uniform System, Lisp, and Lynx, respectively.

Checkers tasks. The primary data structures used to implement Checkers are the representations of the board and the moves. A short pipeline of representations is needed to support backing up to legal or stable states. There are four different board representations: digitized image, calibration information, (x, y, z) location of the piece centroids, and a symbolic description of the board. Each is used for different tasks.

Three different representations for moves are used: the new board state that results from the move, a sequence of physical coordinates for the robot motion commands, and the list of partial moves (that is, a push or a sequence of jumps) needed to execute a move.

These representations for the board and the moves are encapsulated in the *board module*, which provides synchronized access to the data structures and translation routines between the various representations. We implemented the board module 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 board module synchronization routines use the Psyche conventions for process management to implement semaphores that any model can call.

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

The *camera manager* is a Uthread module that maps and initializes a memory segment for the VME memory used to control and access the MaxVideo hardware. This module registers the name and address of the memory segment with a name server. The board interpreter (discussed below) accesses this segment directly to retrieve an image from the MaxVideo frame buffer. The frame buffer is filled at 30 Hz by the digitizer but is read out only when the board interpreter is ready to analyze another image.

The *board interpreter* is 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 Mbyte of image information over the VMEbus takes 280 milliseconds. After transferring the image, the board interpreter segments the image into 64 board squares and analyzes each square in parallel. Each task attempts to determine the color of its square, whether the square contains a piece, and, if so, the piece's color. Each square is then labeled according to the square's color and the piece's color. Piece centroids are calculated, as are centers of empty squares, in image and world coordinates. 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 about once every second.

The *move recognizer* is a Uthread module that compares two successive symbolic board interpretations produced by the board interpreter. It recursively decomposes the differences into a sequence of legal partial moves (single jumps or moves) that transforms the first interpretation into the second.

The *checkers player* is a 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 the work, implementing a parallel α-β game-tree search.

The *move planner* is a trajectory calculation and planning program written in Multilisp. It constructs, in parallel, artificial potential fields that have peaks reflecting square occupancies and a global bias reflecting off-board goal locations. 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. The program considers these potential fields 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. The algorithm chooses one plan on the basis of some cost function, such as the total estimated time to complete the move or the shortest distance to push the checker.

The *robot controller* is 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.

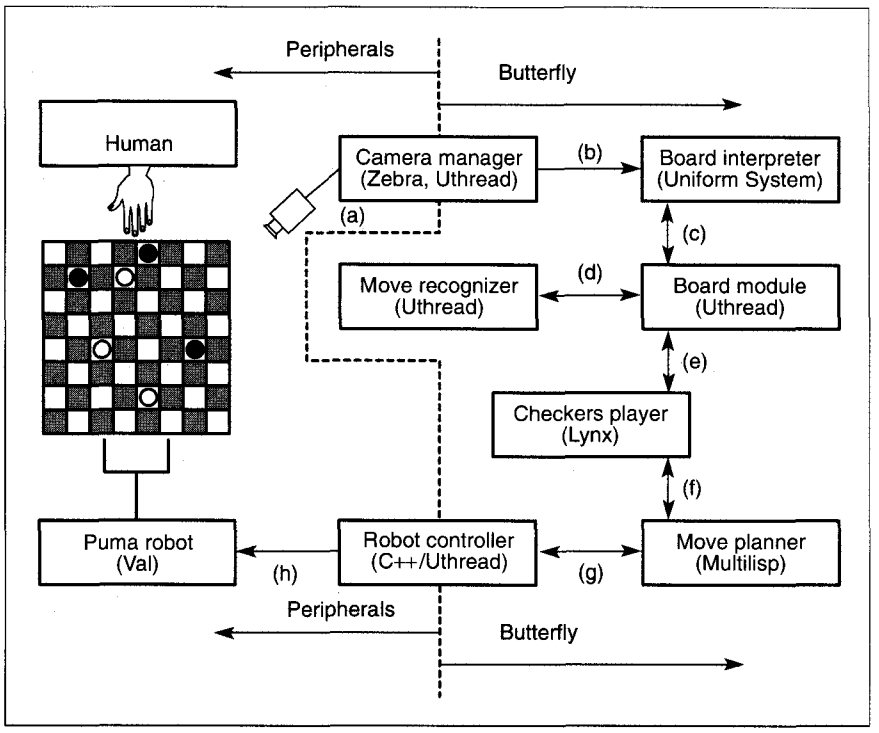


Figure 2. Functional modules and communication paths in Checkers. Multiple models of parallelism (to the right of the dotted line) are implemented under Psyche on the Butterfly. Perceptual and motor modules (to the left of the dotted line) reside on the Butterfly and in peripherals.

Implementation of moves. Program execution is implemented as a series of moves, each of which requires the cooperation of several modules and programming models. Lettered arrows in Figure 2 show control flow among the modules.

The board interpreter continuously receives an image from the camera (a, b) and analyzes it. When the board position changes, the board interpreter invokes the board module (c) to update the board description, passing the symbolic and quantitative positions of the checkers.

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, causing the board interpreter to resume evaluation of raw image data.

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 player module. If several partial moves are needed to complete the move, additional invocations from the checkers player to the board module (e) follow. If any partial move 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.

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 made. At this point the checkers player module 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 list of partial moves that define the robot's response, the checkers player invokes the move planner (f) with the partial move sequence. Partial moves

Table 2. Functional modules and source code statistics (number of lines) for Checkers. The runtime environment code and the Lynx game player were ported from existing systems; 4,482 lines of new code were written, including 2,902 lines of application code and 1,580 lines of interface code.

Function	Model	Application	Runtime	Interface
Game player	Lynx	1,800	8,838	504
Board interpreter	Uniform System	380	8,057	170
Move planner	Multilisp	900	13,127	572
Board module	Uthread	1,300	2,436	170
Robot controller	Uthread	27	2,436	110
Speech controller	Uthread	211	2,436	34
Camera manager	Uthread	84	2,436	20

are passed to the move planner one at a time, and each one causes a sequence of low-level move commands and acknowledgments to flow back and forth between the move planner, the robot controller, and the robot (g, h).

Intermodel communication. The implementation of a single move illustrates two distinct styles of interaction 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 pointers 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 communicating between models via shared data structures:

- Because 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.
- Communication is efficient because processes can use shared memory to communicate directly.
- Synchronization across models is efficient because of 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).
- The board module resembles a blackboard communication structure, but we can use shared data abstractions between models 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: 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 environment 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.

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; it 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 resurrected a Lynx checkers-playing program 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 board interpreter, move planner, board module, and move recognizer, as well as necessary Psyche support for the particular models we used, were all developed simultaneously by people who 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 inte-

gration, 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. Table 2 shows the relatively small amount of coding we had to do to integrate the various Checkers subsystems.

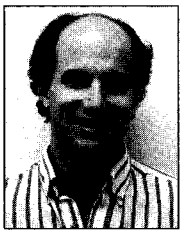
Our ability to build the stylized data abstractions and communication protocols used in Checkers suggests that we will have little difficulty experimenting with alternative communication protocols or processor assignments. This is precisely the type of flexibility required in animate vision systems, and our experiences suggest that multimodel programming in general, and the Psyche mechanisms in particular, can provide the needed flexibility. ■

Acknowledgments

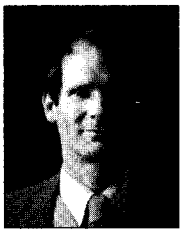
This research was supported by the National Science Foundation under grants IRI-8920771, CDA-8822724, CCR-9005633, and IRI-8903582. Support also came from ONR/DARPA Contract N00014-82-K-0193. Brian Marsh was supported by a DARPA/NASA graduate research assistantship in parallel processing.

References

1. Y. Aloimonos and D. Shulman, *Integration of Visual Modules*, Academic Press, New York, 1989.
2. D.H. Ballard, "Animate Vision," *Artificial Intelligence*, Vol. 48, No. 1, Feb. 1991, pp. 57-86.
3. C.C. Weems et al., "IU Parallel Processing Benchmark," *Proc. Computer Society Conf. Computer Vision and Pattern Recognition*, CS Press, Los Alamitos, Calif., Order No. 862, 1988, pp. 673-688.
4. R.H. Thomas and W. Crowther, "The Uniform System: An Approach to Runtime Support for Large-Scale Shared-Memory Parallel Processors," *Proc. 1988 Int'l Conf. Parallel Processing, Vol. II — Software*, CS Press, Los Alamitos, Calif., Order No. 889, 1988, pp. 245-254.
5. M.L. Scott, "The Lynx Distributed Programming Language: Motivation, Design, and Experience," *Computer Languages*, Vol. 16, No. 3/4, 1991, pp. 209-233.
6. M.L. Scott, T.J. LeBlanc, and B.D. Marsh, "Multi-Model Parallel Programming in Psyche," *Proc. Second ACM Conf. Principles and Practice of Parallel Programming*, ACM, New York, 1990, pp. 70-78.
7. B.D. Marsh et al., "First-Class User-Level Threads," *Proc. 13th Symp. Operating Systems Principles*, ACM, New York, 1991, pp. 110-121.
8. R. Halstead, "Multilisp: A Language for Concurrent Symbolic Computation," *ACM Trans. Programming Languages and Systems*, Vol. 7, No. 4, Oct. 1985, pp. 501-538.



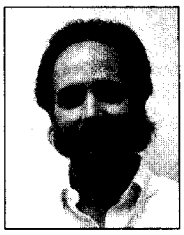
Brian Marsh is a research scientist at the Matsushita Information Technology Laboratory in Princeton, N.J. His research interests include multiprocessor and distributed operating systems. Marsh received his MS and PhD in computer science from the University of Rochester in 1988 and 1991, respectively.



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 the integration of computer vision, robotics, and parallel computation into active intelligent systems. Brown received his PhD in information sciences from the University of Chicago in 1972.

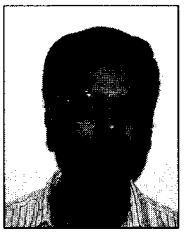
Thomas LeBlanc is an associate professor and chairman of the Computer Science Department at the University of Rochester. His research interests include parallel-programming environments and multiprocessor operating systems. LeBlanc received his PhD in computer sciences from the University of Wisconsin – Madison in 1982. He is a member of the IEEE Computer Society.

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 PhD in computer sciences from the University of Wisconsin – Madison in 1985. He is a member of the IEEE and the IEEE Computer Society.

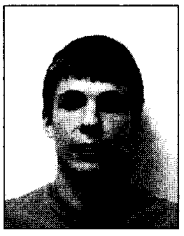


Tim Becker is on the technical staff of the Computer Science Department at the University of Rochester. His recent work has included projects in computer vision and robotics, and the implementation of the Psyche multiprocessor operating system. Becker received his BS in industrial engineering from Pennsylvania State University in 1980.

Cesar Quiroz is a software engineer with EXELE Information Systems, East Rochester, New York. His research interests center on the study of parallelism in programming-language implementation, especially the parallelization of imperative code. Quiroz received his PhD in computer science from the University of Rochester in 1991. He is a member of the IEEE Computer Society.



Prakash Das is a system designer at Transarc Corporation in Pittsburgh. His research interests include multiprocessor operating systems. Das received his MS 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 BS in computer science from Stanford University in 1990.

Readers may contact Chris Brown at the Computer Science Department, University of Rochester, Rochester, NY 14627-0226, e-mail brown@cs.rochester.edu.