RECENT DEVELOPMENTS IN BUTTERFLY™ LISP

Donald C. Allen, Seth A. Steinberg, and Lawrence A. Stabile
BBN Advanced Computers, Inc.
10 Fawcett Street
Cambridge, Massachusetts 02238

Abstract

This paper describes recent enhancements to the Common Lisp system that BBN is developing[1] for its Butterfly multiprocessor. The BBN Butterfly is a shared memory multiprocessor that contains up to 256 processor nodes. The system provides a shared heap, parallel garbage collector, and window based I/O system. The 'future' construct is used to specify parallelism.

I. Introduction

BBN has been actively involved in the development of shared-memory multiprocessor computing systems since the early 1970s. The first such machine was the Pluribus, a bus-based architecture employing commercially-available minicomputers. The Butterfly™ system is BBN's second generation parallel processor and builds on BBN's experience with the Pluribus. In 1986, BBN formed a wholly-owned subsidiary, BBN Advanced Computers Inc., to further develop the Butterfly architecture. More than 70 Butterfly systems are currently used for applications such as complex simulation, symbolic processing, image understanding, speech recognition, signal processing and data communications.

For approximately two years, we have undertaken (with DARPA support) the development of a Lisp programming environment for the Butterfly. At AAAI '86 [Steinberg *et al.*, 1986] we reported on the basic design of the system. In this paper, we discuss a variety of developments resulting from the work of the past year.

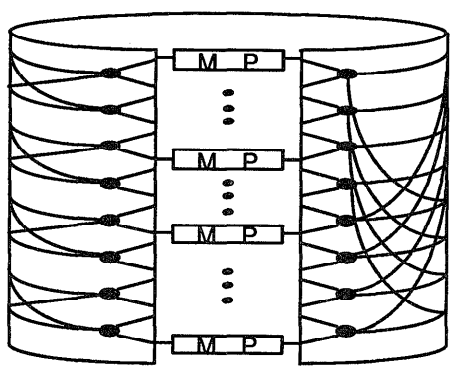

Figure 1: The Butterfly Multiprocessor

[1]The work described herein was done at BBN Advanced Computers, Inc. under contract to the Defense Advanced Research Projects Agency of the Department of Defense

II. The Butterfly Architecture

The Butterfly parallel processor includes from one to 256 processor nodes interconnected by a high performance logarithmic switching network (figure 1). Each processor node contains a Motorola 16-MHz MC68020 microprocessor; a Motorola MC68881 floating point coprocessor; up to four megabytes of memory; a general I/O port; and an interface to the Butterfly switch (see figure 2). Each node also contains a microcoded coprocessor, called the Processor Node Controller (PNC), that performs switch and memory management functions, as well as providing extensions to the 68020 instruction set in support of multiprocessing.

The memory management hardware, combined with the small latency of the Butterfly switch, permit the memories of the individual processor nodes to be treated as a pool of shared memory that is directly accessible by all processors. Remote memory references are made through the Butterfly switch and take approximately four microseconds to complete regardless of configuration size. This shared-memory capability is crucial to the implementation of Butterfly Lisp.

The Butterfly is designed to maintain a balance of processing power, memory capacity, and switch bandwidth over a wide range of configurations. The largest Butterfly system consists of 256-processor nodes and executes 250 million instructions per second, has a gigabyte of memory and provides eight gigabits per second of switch bandwidth.

The Butterfly system's expandability is due predominantly to the design of the Butterfly switch network. The switch is built from intelligent four-by-four crossbars configured in a serial decision network. The cost of the switch and switch bandwidth grow almost linearly, preserving price-performance from the smallest to the largest configuration.

III. Butterfly Lisp Overview

Butterfly Lisp is a shared-memory, multiple-interpreter system. Rather than implementing a loosely coupled set of separate Lisps that communicate via some external message-passing protocol, we have chosen to capitalize on the Butterfly's ability to support memory-sharing by providing a single Lisp heap, mapped identically by all interpreters. This approach preserves the shared-memory quality that has always been characteristic of the Lisp language: data structures of arbitrary complexity are easily communicated from one context to another by simply transmitting a pointer, rather than by copying. We believe this approach has significant ease-of-programming and efficiency advantages.

Butterfly Lisp uses the "future" mechanism (first implemented at MIT by Professor Robert Halstead and students in Evaluating the form

```
(future <lisp-expression>)
```

causes the system to record that a request has been made for the evaluation of <lisp-expression> and to commit resources to that evaluation when they become available. Control returns immediately to the caller, returning a new type of Lisp object called an "undetermined future". The "undetermined future" object serves as a placeholder for the value that the evaluation of <lisp-expression> will ultimately produce. An "undetermined future" may be manipulated as if it were an ordinary Lisp object: it may be stored as the value of a symbol, consed into a list, passed as an argument to a function, etc. If, however, it is subjected to an operation that requires the value of <lisp-expression> prior to its arrival, that operation will automatically be suspended until the value becomes available. The "future" mechanism provides an elegant abstraction for the synchronization required between the producer and consumer of a value. This preserves and encourages the applicative programming style so integral to Lisp programming and so important in a parallel machine, where carelessness with side-effecting operations can result in (difficult to diagnose) probabilistic bugs.

The Butterfly Lisp implementation is based on MIT CScheme, modified to support the future construct and other multiprocessing primitives (e.g., primitives for mutual exclusion). The system also includes a parallel stop-and-copy garbage collector. In addition to the Scheme dialect of Lisp, Butterfly Lisp also provides Common Lisp language support, implemented largely on top of Scheme. This implementation, which we discuss in detail below, uses significant portions of CMU's Spice Lisp.

The Butterfly Lisp compiler will be based on LIAR (LIAR Imitates Apply Recursively), a Scheme compiler recently developed at MIT by members of the Scheme Team.

The Butterfly Lisp User Interface (which leads to the rather unfortunate acronym BLUI) is implemented on a Symbolics 3600-series Lisp Machine and communicates with the Butterfly using Internet protocols. This system provides a means for controlling and communicating with tasks running on the Butterfly, as well as providing a continuously updated display of the overall system status and performance. Special Butterfly Lisp interaction windows, associated with tasks running on the Butterfly, may be easily selected, moved, resized, or folded up into task icons.

There is also a Butterfly Lisp mode provided for the ZMACS editor, which connects the various evaluation commands (e.g., evaluate region) to an evaluation service task running in the Butterfly Lisp system.

Each task is created with the potential to create an interaction window on the Lisp machine. The first time an operation is performed on one of the standard input or output streams a message is sent to the Lisp machine and the associated window is created. Output is directed to this window and any input typed while the window is selected may be read by the task. This multiple window approach makes it possible to use standard system utilities like the trace package and the debugger.

A pane at the top of the screen is used to display the system state. The system state information is collected by a Butterfly process that is separate from the Butterfly Lisp system, but has shared-memory access to important interpreter data structures. The major feature of this pane is a horizontal rectangle broken vertically into slices. Each slice shows the state of a particular processor. If the top half of the slice is black, then the processor is running, if gray, it is garbage collecting, and if white, it is idle. The bottom half of each slice is a bar graph that shows how much of each processor's portion of the heap is in use. The status pane also shows, in graphical form, the number of tasks awaiting execution. This display makes such performance problems as task starvation easy to recognize.

A recently developed User Interface facility provides a display of a combined spawning and data-dependency graph, which is created from metering data collected during the running of a Lisp program on the Butterfly. This facility is described in detail below.

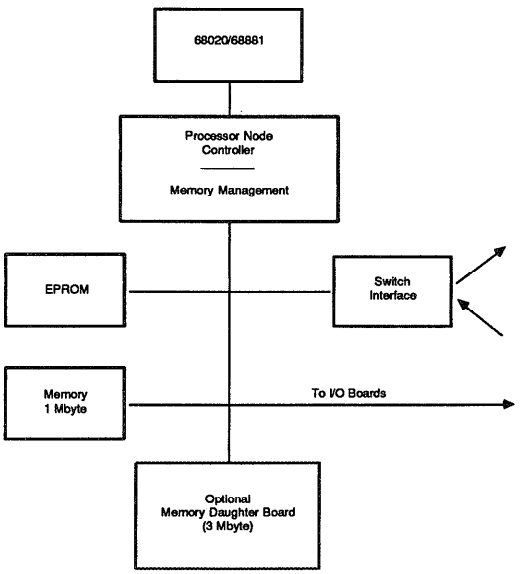

Figure 2: The Butterfly Processor Node

IV. Imaging Execution of Butterfly Lisp Programs

A. Introduction

Computer software execution is both invisible and complex and thus it has always been a challenge to present an image of how programs are executing. Many tools have been developed to perform this function in uniprocessor programming environments. For example, debuggers present a picture of processor state frozen in time, while profilers provide a summary distribution of program execution time. Useful as these may be, the added dimension of concurrency renders the typical suite of such tools insufficient. This realization led us to the development of the capability described in this section.

B. Metering and Presentation Facilities

The Butterfly Lisp User Interface uses a transaction-oriented protocol that carries the various logical I/O streams between the Butterfly and the 3600. To support our new imaging facility, the protocol was extended so that the tasking system could transmit a packet for each major scheduling operation. When metering is enabled, a packet is sent when a task

- is created,
- begins executing on a processor,
- requires the final value of another task which has not yet been computed,
- finishes executing,

- terminates, enabling another task to resume execution.

Each packet contains the time, the unique identifiers of the relevant tasks and the processor number.

These packets allow the User Interface to construct a complete lifetime history of each task, reconstruct the task creation tree, and partially reconstruct the data dependency graph. For efficiency reasons, if the value of a future has been determined before it is needed, the reference is not recorded. The User Interface presents this information in a special window. The top pane of the window contains the task histories in depth-first order of their creation. The bottom pane contains a derived chart of aggregate information about the tasks in the system. In both panes, the horizontal axis is the time axis and the two charts are presented in the same time scale.

The history of each task is displayed as a horizontal rectangle ranging along the time axis from task creation time to task termination time. When vertical spacing permits, the gray level indicates the state of the task that may be running (black), on the scheduler queue (gray) or idle (white). The uppermost task history rectangle in figure 3 shows a task that is created and placed on the scheduler queue (gray). It then runs (black), creating subtasks (see arrows), until it needs the value computed by another task (see arrow). At this point it goes idle (white). When the subtask has determined that value, this task is requeued (gray) and then runs (black) to completion.

An arrow drawn from one task history to another indicates that either:

- a task has created another task, or
- a task needs the result of another task, or
- a terminating task has restarted another task.

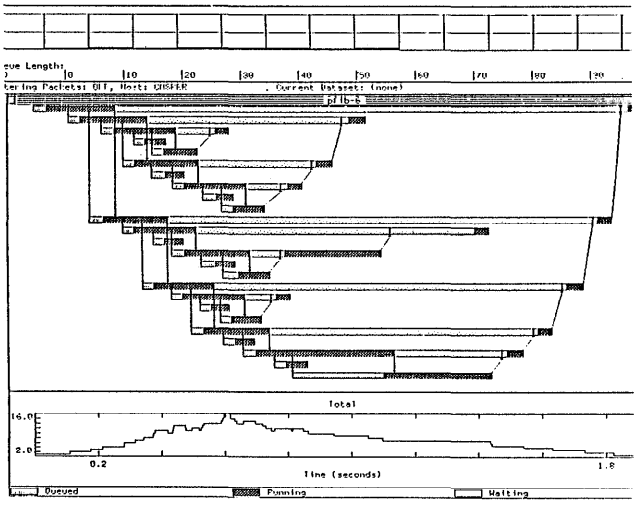

Figure 3: Execution image of BBNACI function

Since the arrows are always associated with task state changes, the actual meaning of a particular arrow can easily be derived from the context. When a task is created, the creating task continues running so the history rectangle is black to the right of the creation arrow. When a task needs the result of another task it will go idle so the history rectangle will go from black to white. When a task terminates, the history rectangle ends, and any waiting tasks are restarted.

As the number of task histories to be shown increases, the history rectangles will be made smaller. If necessary, the black-gray-white shadings will not be shown. If still more data is available it will not be drawn, but can be displayed by using the mouse to scroll the window. The mouse can also be used to zoom in on the task histories by selecting a rectangular region, which will be expanded to fill the entire window. This zooming can be done repeatedly, and the image can be returned to its original scale by pressing the middle mouse button. As an additional aid to deciphering the task structure, pointing the mouse at a particular task history will "pop up" the text description of the task being computed.

The more conventional graph in the lower pane displays the number of tasks in a particular state plotted against time. Ordinarily, this graph shows the total number of tasks in the system. There is a menu of graphing options that allows the user to select a plot of any combination of running, queued, or idling tasks.

The images in this paper were produced using an interpreted version of Butterfly Lisp running on a 16 processor Butterfly (with one node used for network communications). Figure 3 was produced by the execution of the following recursive Fibonacci algorithm:

```
(define (bbnaci n); Pronounced bi-bin-
                  ; acci
  (if (< n 2)
      n
      (+ (future (bbnaci (-1+ n)))
         (future (bbnaci (- n 2))))))
```

C. Example: Boyer-Moore Theorem Prover

The Boyer-Moore theorem prover (a classic early AI program and part of the Gabriel Lisp benchmark suite [Gabriel, 1985]) works by transforming the list structure representing the theorem into a canonical form using a series of rewrite rules called lemmas. The final form, which resembles a tree structured truth table, is scanned by a simple tautology checker. The algorithm starts rewriting at the top level of the theorem and recursively applies the rewriter to its results. Parallelism is introduced by creating a subtask for each subtree that might need to be transformed.

Figure 4 shows the proof of modus-ponens:

```
(implies (and (implies a b) a) b)
```

The proof performs 13 transformations that appear as the 13 clusters in the execution image. Two levels of parallelism are visible. At the finer level, various tasks attempt to apply rewrite rules, usually unsuccessfully. Many short lived tasks are created, but parallelism is limited by the small size of the associated list structure. This limited parallelism appears within each cluster of tasks. At the coarser level, the various lemmas are applied and their dependencies are preserved. The first set of transformations must be performed serially because each depends on the results of the previous one. Later transformations can be performed in parallel on isolated subbranches of the expanded list structure. This parallelism appears in the arrangement of the clusters.

The rectangle drawn around cluster 6 is a region that has been selected for enlargement. Clicking the left mouse button changes the cursor from an arrow to the upper left hand corner of a rectangle. When this is clicked into place, the mouse allows the user to "rubber band" a rectangle by moving the lower right hand corner. The next click enlarges the region within the rectangle, filling the entire screen as shown in figure 5.

D. Summary

The illustrations in this paper demonstrate the utility of this new parallel program execution imaging technique. In the future, we will add a variety of viewing modes to provide alternate images of execution and we plan to allow the user to explicitly annotate the diagram by sending appropriate data packets. It should also be possible to use the metering data to detect circular dependencies, unused task subtrees, task starvation, and other common pitfalls on the path to parallelism.

V. Butterfly Common Lisp

A key aspect of the Butterfly Lisp effort is the support of a concurrent variant of Common Lisp. In this section we discuss some of the issues that arise in building a Common Lisp on a Scheme base.

A. Scheme

Scheme is a dialect of Lisp developed at MIT by Gerald J. Sussman and Guy Steele in 1975. While the language has evolved during the past 12 years [Rees *et al..*, 1986], it has, from the beginning, featured lexical-scoping, tail-recursion, and first-class procedural objects, which unify the operator and operand name spaces (the first element of a form is evaluated in exactly the same manner as the rest). Scheme is in an excellent base on which to build a Common Lisp, having simple, powerful and well-defined semantics. For example, Scheme provides a very general facility for creating advanced control structures, i.e., the ability to create and manipulate 'continuations' (the future course of a computation) as first-class objects. This capability is critical to, and greatly simplifies, our implementations of Common Lisp tagbodies, block/return-from, and catch/throw. In addition to the language features described in the Scheme report [Rees *et al..*, 1986], Butterfly Common Lisp makes significant use of extensions to the language provided by MIT CScheme:

- Environments are first-class objects, i.e., environment objects, and operations on them, are directly accessible to the user.
- The Eval function exists, accepting an environment as a required second argument.
- CScheme provides many of the arithmetic types required by Common Lisp, including an efficient bignum implementation.
- Lambda lists that support rest and optional arguments: sufficient power on which to construct Common Lisp's more complex lambda lists.
- A macro facility of sufficient power to construct Defmacro.
- A simple protocol for adding new primitives, written in C, to support some of the more esoteric Common Lisp operations.
- Fluid-let, a dynamic-binding construct that is used in the implementation of Common Lisp special variables
- Dynamic-wind, a generalization of Common Lisp unwind-protect

B. Mapping from Common Lisp to Scheme

It was decided early in the evolution of our Common Lisp that we would avoid modifications to the Scheme interpreter and compiler, unless semantic or performance considerations dictated otherwise. In addition, we decided that we would capitalize as much as possible on CMU's excellent Common Lisp, Spice.

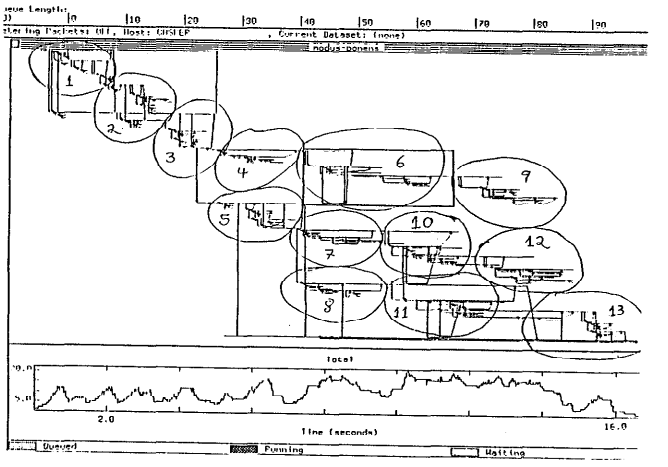

Figure 4: Execution image of Boyer-Moore Theorem Prover

Our task then became one of identifying and implementing enough of the basic Common Lisp semantics to be able employ the CMU Common Lisp-in-Common Lisp code. Happily, in many cases, Scheme and Common Lisp were identical or nearly so. In others, language differences required resolution; Common Lisp's separate operator/operand name spaces are an example. Here, an obvious solution would have been to add a function definition cell to the symbol data structures, necessitating a non-trivial interpreter and compiler modification in the treatment of the evaluation of the first element of function-application forms. We chose a less invasive solution: when the Scheme syntaxer (which converts reader-produced list structure into S-code, the input to both the interpreter and the compiler) senses any form with a symbol in the function position, it substitutes a new, uninterned symbol having a pname that is derived from the pname of the original. Function-defining constructs, e.g., defun, (setf (symbol-function)), place the functional objects in the value cell of the corresponding mutated symbol. The symbol block has been expanded so that the pairs can point to each other.

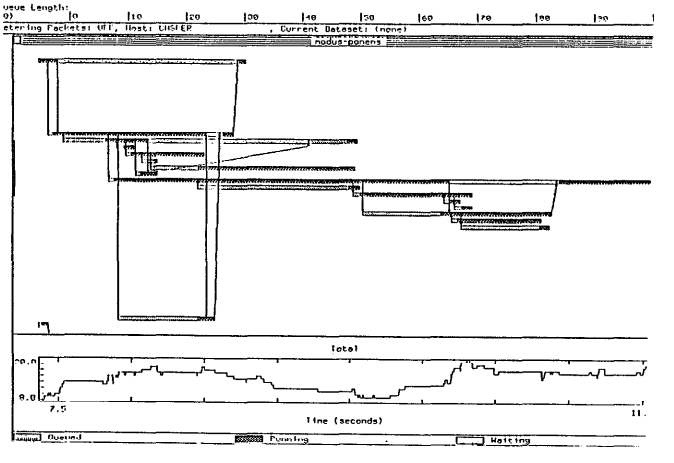

Figure 5: Execution image of Boyer-Moore Theorem Prover -- close-up view

Common Lisp packages are another example of the gulf between the two languages; Scheme relies exclusively on lexical scoping to solve name-conflict problems. The solution was simply a matter of extending the symbol table system of Scheme to accommodate the needs of the Spice Lisp package system and reader. This has amounted to little more that porting the Spice Lisp package system, and viewing Scheme symbols as belonging to the 'Lisp' package. To do this, the package operations are multiplexed between the Scheme and Spice data structures via a simple object-oriented tool.

Common Lisp provides a large set of control mechanisms, including looping and branching. Scheme has no inherent looping or branching, and instead relies on tail-recursion optimization for efficient iteration and on continuations for other types of control structures.

The simpler looping forms of Common Lisp are easy to implement in Scheme. For example, examine the following simple use of dotimes, in which no go's appear:

```
(dotimes (i n) (print (factorial i)))
```

transforms to the Scheme:

```
(let ()
 (define (dotimes-loop-1234 i)
  (if (= i n)
      nil
      (begin
        (print (factorial i))
        (dotimes-loop-1234 (1+ i)))))
 (dotimes-loop-1234 0))
```

The more complicated branching implied by tagbody is implemented using continuations. In essence, the sections of a tagbody between labels become zero-argument procedures that are applied in sequence, calling each other tail-recursively to perform a go.

Mechanisms for parallelism extant in Butterfly Lisp are inherited by Common Lisp. These include future, the locking primitives, and all metering and user-interface facilities, such as the imaging system described in section 4.

c. Implementation Status

Currently, almost all of Common Lisp is implemented and runs in interpreted mode. Much of Portable Common Loops, a severe test of any Common Lisp implementation, has been ported.

VI. The Butterfly Lisp Compiler

The Butterfly Lisp compiler will be based on a new Scheme compiler, LIAR, recently developed at MIT. LIAR performs extensive analysis of the source program in order to produce highly optimized code. In particular, it does a careful examination of program structure in order to distinguish functions whose environment frames have dynamic extent, and are thus stack-allocatable, from those requiring frames of indefinite extent, whose storage must be allocated from the Lisp heap. This is an extremely important efficiency measure, dramatically reducing garbage collection time.

LIAR also deals efficiently with an interaction between lexical scoping and tail-recursion: a procedure called tail recursively may make a free reference to a binding in the caller's frame, thus it is not always possible for the caller to pop his own frame prior to a tail-recursive call; the callee must do it. Furthermore, the compiler cannot always know how the called procedure will be called (tail-recursively or otherwise), and thus cannot statically emit code to clean up the stack, since the requirements are different in the two cases. This is a problem that requires a runtime decision about how to deal with the stack. LIAR handles this with an extremely efficient technique that preserves the semantics of lexical scoping and the constant-space attribute of tail-recursion.

At the time of this writing, LIAR is undergoing extensive testing both at MIT (on the HP Bobcat) and at BBN on the Sun 3. The port to the Butterfly computer will begin shortly and is expected to take a matter of weeks. In addition, the compiler will continue to be refined (certain basic optimizations, such as user control over in-lining of primitives -- the compiler presently in-lines only car, cdr, and cons -- have been omitted for the present, the strategy being to get a relatively unadorned version working first). We will also be modifying the compiler in behalf of Common Lisp. In particular, processing of Common Lisp's optional type, 'inline', and 'optimize' declarations can be critical in achieving acceptable Common Lisp performance on a processor such as the MC68020.

It is hoped that we will be able to report results obtained from running compiled Lisp on the Butterfly at the time of the conference.

Acknowledgements

The authors would like to thank the following people for their contributions to our efforts:

- The MIT Scheme Team: Jim Miller, Chris Hanson, Bill Rozas, and Professor Gerald J. Sussman, for their tremendous skill, cooperation, and support.
- Laura Bagnall, the person responsible for all the magic that happens on the 3600 side of our system. Laura was a member of our team almost since the project's inception and recently left us to return to MIT in pursuit of a Ph. D.
- Jonathan Payne, a University of Rochester undergraduate who has worked with us during school vacations, and has made a very significant contribution to the project, particularly to the Common Lisp implementation.
- DARPA, for providing support for this work.

References

[Steinberg *et al.*, 1986] Steinberg, S., Allen, D., Bagnall, L., Scott, C. The Butterfly Lisp System Proceedings of the August 1986 AAAI, Volume 2, Philadelphia, PA, pp. 730

[Crowther *et al.*., 1985] Crowther, W., Goodhue, J., Starr, E., Milliken, W., Blackadar, T. Performance Measurements on a 128-Node Butterfly Parallel Processor Internal Bolt, Beranek and Newman Paper, Cambridge, Massachusetts

[Gabriel, 1985] Gabriel, R. Performance and Evaluation of Lisp Systems MIT Press, 1985

[Halstead, 1985] Halstead, R. Multilisp: A Language for Concurrent Symbolic Computation ACM Transactions on Programming Languages and Systems

[Rees *et al.*., 1986] Rees, J., Clinger, W., et al Revised^3 Report on the Algorithmic Language Scheme MIT report, Cambridge, Mass.