From: AAAI-86 Proceedings. Copyright ©1986, AAAI (www.aaai.org). All rights reserved.

The Butterfly™ Lisp System

Seth A. Steinberg
Don Allen
Laura Bagnall
Curtis Scott

Bolt, Beranek and Newman, Inc.
10 Moulton Street
Cambridge, MA 02238

ABSTRACT

This paper describes the Common Lisp system that BBN is developing for its Butterfly™ multiprocessor. The BBN Butterfly™ is a shared memory multiprocessor which may contain 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.

THE BUTTERFLY™ LISP SYSTEM

For several decades, driven by industrial, military and experimental demands, numeric algorithms have required increasing quantities of computational power. Symbolic algorithms were laboratory curiosities; widespread demand for symbolic computing power lagged until recently. The demand for Lisp machines is an indication of the growth of the symbolic constituency. These machines possess architectural innovations that provide some performance increases, but they are still fundamentally sequential systems. Serial computing technology is reaching the point of diminishing returns, and thus both the numeric and symbolic computing communities are turning to parallelism as the most promising means for obtaining significant increases in computational power.

BBN has been working in the field of parallel computers since the early 1970's, having first developed the Pluribus multiprocessor and more recently, the Butterfly, the machine whose programming environment we concern ourselves with in this paper. The Butterfly multiprocessor consists of a set of up to 256 nodes, each containing both a processor and memory, connected by a Butterfly switch (a type of Omega network) (see figure 1). Each node has from 1 to 4 megabytes of memory, a Motorola 68000 series processor and a special purpose Processor Node Controller (PNC). The PNC is microprogrammed to handle inward and outward Butterfly switch transactions, and to provide multiprocessing extensions to the 68000 instruction set, particularly in cases where atomicity is required.[1]

To date, Butterfly programs have been written exclusively in C, with most numeric applications using the Uniform System package. The Uniform System provides and manages a large shared address space and has subroutines which can be used to distribute subtasks to all of the active processors. It has been used to speed up algorithms for matrix multiplication, image processing, determining elements of the Mandelbrot set, and solving differential equations and systems of linear equations.[2 3]

Butterfly™ is a trademark of Bolt, Beranek and Newman.

Under DARPA sponsorship, BBN is developing a parallel symbolic programming environment for the Butterfly, based on an extended version of the Common Lisp language. The implementation of Butterfly Lisp is derived from C Scheme, written at MIT by members of the Scheme Team.[4] The simplicity and power of Scheme make it particularly suitable as a testbed for exploring the issues of parallel execution, as well as a good implementation language for Common Lisp.

The MIT Multilisp work of Professor Robert Halstead and students has had a significant influence on our approach. For example, the `future` construct, Butterfly Lisp's primary mechanism for obtaining concurrency, was devised and first implemented by the Multilisp group. Our experience porting MultiLisp to the Butterfly illuminated many of the problems of developing a Lisp system that runs efficiently on both small and large Butterfly configurations.[5 6]

In the first section, this paper describes `future`-based multitasking in Butterfly Lisp and how it interacts with more familiar Lisp constructs. The second section describes how Butterfly Lisp deals with the problems of task distribution and memory allocation. It contrasts our approach and the Uniform System approach. The third section describes the Butterfly Lisp parallel garbage collector and the fourth section describes the user interface.

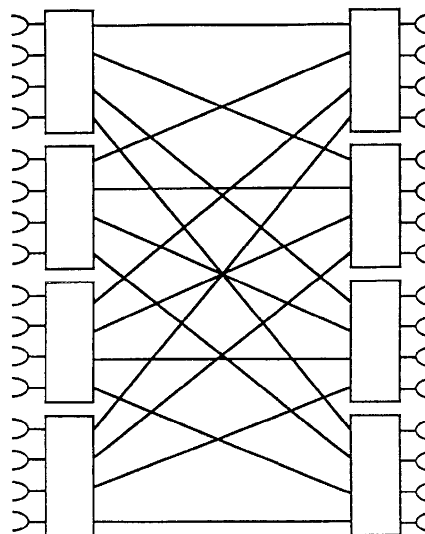

Figure 1: 16x16 Butterfly Switch

BUTTERFLY LISP

Experiments in producing parallel programs that effectively use the available processing power suggest that the fundamental task unit must execute on the order of no more than 100-1000 instructions[7 8 9]. If the task unit is larger, there will be insufficient parallelism in the program. This task size is only slightly larger than the size of the typical subroutine. This similarity in scale implies that the tasking overhead must be within an order of magnitude of the subroutine overhead. To encourage the programmer to use subtasks instead of calls, the tasking syntax should be similar to the calling syntax.

Butterfly Lisp uses the `future` mechanism as its basic task creating construct. The expression:

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

causes the system to note that a request has been made for the evaluation of `<s-expression>`, which can be any Lisp expression. Having noted the request (and perhaps begun its computation, if resources are available), control returns immediately to the caller of `future`, returning a new type of Lisp object called an "undetermined future" or simply, a `future`. The `future` object serves as a placeholder for the ultimate value of `<s-expression>` and 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 `<s-expression>` prior to its arrival, that operation will automatically be suspended until the value becomes available.

`future` provides an elegant abstraction for the synchronization required between the producer and consumer of a value. This permits results of parallel evaluations to be manipulated without explicit synchronization. Thus, Butterfly Lisp programs tend to be quite similar to their sequential counterparts.

This similarity is illustrated by the following example. To convert a simplified serial version of `mapcar`:

```
(defun simple-mapcar (function list)
       (if (null list)
           nil
             (cons (function (car list))
                   (simple-mapcar function
                                  (cdr list)))))
```

into a parallel version requires the addition of a single `future` form:

```
(defun parallel-simple-mapcar (function list)
       (if (null list)
           nil
             (cons (future (function (car list)))
                   (parallel-simple-mapcar
                            function
                             (cdr list)))))
```

In this version of `mapcar`, the primary task steps down the original list, spinning off subtasks which apply `function` to each element. The function, `parallel-simple-mapcar` does not return until all of the subtasks have been spawned. We can create the `futures` more aggressively, as in the following example:

```
(defun aggressive-mapcar (function list)
        (if (null list)
            nil
              (cons (future (function (car list)))
                    (future (aggressive-mapcar
                             function
                              (cdr list))))))
```

A call to `aggressive-mapcar` would quickly start two subtasks and immediately return a cons containing the two `futures`. This makes it possible to start using the result of the subroutine well before the computation has been completed. If a pair of `aggressive-mapcars` is cascaded:

```
(aggressive-mapcar f
      (aggressive-mapcar g x-list))
```

subtasks spawned by the outer `aggressive-mapcar` may have to wait for the results of those spawned by the inner call.

As might be expected, the introduction of parallelism introduces problems in a number of conventional constructs. For example, the exact semantics of Common Lisp `do` are extremely important as in the following loop:

```
(do ((element the-list (cdr element)))
     ((null element))
    (future (progn (process-first (caar element))
                   (process-second
                                 (cadar element)))))
```

In a serial system it doesn't matter if `do` is implemented tail recursively or not. In a parallel system, if the loop is implemented tail recursively the semantics are pretty clear:

```
(defun stepper (element)
        (cond ((null element) nil)
              (t (future
                    (progn
                       (process-first
                                  (caar element))
                       (process-second
                                  (cadar element))))
                 (stepper (cdr element)))))
```

There will be a new environment and so a new binding of `element` for each step through the loop. It is easy to determine which value of `element` will be used in the body of the `future`. This is not the case if the `do` loop is implemented using a `prog`, side-effecting the iteration variable:

```
(prog (element)
 loop (if (null element)
          (return))
      (future (progn
                   (process-first (caar element))
                  (process-second
                            (cadar element))))
       (setq element (cdr element))
      (go loop))
```

Since all iterations share a single binding of `element`, we have a race condition.

IMPLEMENTATION

In some ways, Butterfly Lisp is similar to the Uniform System. Both systems provide a large address space which is shared by all of the processors. Processors are symmetric, each able to perform any task. In both systems programs are written in a familiar language which, for the most part, executes in a familiar fashion. Since the Uniform System has been well tuned for the Butterfly we can use it to study the effectiveness of our implementation.

The Uniform System was designed for numeric computing, taking advantage of the regularity of structure of most numeric problems. At any given time all of the processors are repeatedly executing the same code. This consists of a call to a generator to determine which data to operate on followed by a call to the action routine which does the computation. Loops in typical numeric algorithms can be broken into two parts: the iteration step and the iteration body. On a multiprocessor the iteration step must be serialized, but the iteration bodies may be executed in parallel. To use all processors effectively, the following condition should be met:

$$T_{body} >= T_{step} * N_{processors}$$

The Uniform System produces its best performance when the iteration step is completely independent of the iteration body, as is the case in many numeric programs.

While symbolic tasks are usually expressed recursively rather than iteratively, this shouldn't be a problem. Recursion can be broken into a recursion step and a recursion body. Stepping through a list or tracing the fringe of a tree is not significantly more expensive than incrementing a set of counters. If we are scanning a large static data structure, then our recursion step and recursion body will be independent and we can obtain performance improvements similar to those produced by the Uniform System.

Unfortunately, this condition cannot always be met. Many symbolic programs repeatedly transform a data structure from one form to another until it has reached some desired state. The Boyer-Moore theorem prover applies a series of rewrite rules to the original theorem and converts it into a truth table-based canonical form. Macsyma repeatedly modifies algebraic expressions as it applies various transformations. In these cases, the recursion step often must wait until some other recursion body has finished before the next recursion body can begin; the `future` mechanism is better suited to dealing with this interdependency.

It was because of these differences between the numeric and symbolic worlds that we implemented the `future` mechanism, which can be used to write programs in a number of styles. The basic units of work are either procedures or continuations, both of which are first class Scheme objects. When a task is first created by the execution of the `future` special form, a procedure is created and placed on the work queue. Whenever a processor is idle it takes the next task off this queue and executes it. When a task must wait for another task to finish evaluating a `future`, its continuation is stored in that `future`. Continuations are copies of the control stack created by the `call-with-current-continuation` primitive. When a `future` is determined, any continuations waiting for its value are placed on the work queue so they may resume their computations.

The Butterfly Lisp memory allocation strategy is different from that of the Uniform System. While the Butterfly switch places only a small premium on references to memory located on other nodes, contention for the nodes themselves can be a major problem. If several processors attempt to reference a location on a particular target node, only one will succeed on the first try; the others will have to retry. With a large number of processors, this can be crippling. To minimize contention, the Uniform System assists the programmer in scattering data structures across the nodes. In addition, it encourages the programmer to make local copies of data, providing a number of highly optimized copying routines that depend on the contiguous nature of numeric data structures.

The complicated, non-contiguous data structures used in symbolic computation make the copying operation far more expensive, and thus worthwhile in fewer cases. Copying also introduces a number of consistency and integrity problems. Lisp makes a fundamental distinction between the identity primitive (`eq`) and the equality primitives (`eql`, `equal`, `=`). While other languages draw this distinction, it is frequently fundamental to the working of many symbolic algorithms.

To diffuse data structures throughout the machine, the Lisp heap is a striated composite of memory from all of the processors. The garbage collector, which is described in the next section, is designed to maintain this diffusion. Each processor is allotted a share of the heap in which it may create new objects. This allows memory allocation operations to be inexpensive, as they need not be interlocked.

GARBAGE COLLECTION

Butterfly LISP has a parallel stop and copy garbage collector which is triggered whenever any processor runs out of heap space. (This strategy runs the risk that garbage collections might occur too frequently, since one processor might use memory much more quickly than the others. Experiments indicate that there is rarely more than a 10% difference in heap utilization among the processors).[10]

When a processor realizes that it must garbage collect, it generates a local garbage collection interrupt. The handler for this local interrupt uses a global interrupt to notify every processor that systemwide activity is needed. A global interrupt works by sending a handler procedure to each of the other processors, which they will execute as soon as they are able.

Global interrupts do not guarantee synchrony. Since all processors must start garbage collecting at once and must not return from the interrupt handler until garbage collection is completed, the processors must synchronize before and after garbage collecting. This is accomplished by use of the `await-synchrony` operation, which uses a special

synchronizer object. Each processor awaits synchrony until all processors are waiting for the same synchronizer, at which time they may all continue. Internally, a synchronizer contains a waiting processor count that is atomically decremented as each processor starts waiting for it. When this count goes to zero, all of the processors may proceed.

The garbage collection interrupt handler uses global interrupts and synchronizers something like this:

```
(defun handle-local-gc-interrupt ()
   (let ((start-synch (make-synchronizer))
         (end-synch (make-synchronizer)))
     (global-interrupt
      (lambda ()                ; The Handler
              (await-synchrony start-synch)
              (garbage-collect-as-slave)
              (await-synchrony end-synch)))
     (await-synchrony start-synch)
     (garbage-collect-as-master)
     (await-synchrony end-synch)))
```

On a serial machine, a copying garbage collector starts by scanning the small set of objects called the root, which can be used to find all accessible data. Scanning consists of checking each pointer to see if it points into old space. Old space objects are copied into new space and the original old space pointer is replaced by a pointer to the copied object in new space. A marker is left in old space so that subsequent checks do not copy the object again. Once an object has been copied into new space it must be scanned as well, since it may still contain pointers into old space. The scanning and copying continues until everything in new space has been scanned, at which time there are no more pointers into old space and garbage collection is complete.

The Butterfly Lisp garbage collector works by breaking new space into partitions. These are the basic units of memory which are scanned or copied into. Each processor waits for a partition to appear on the work queue and begins to scan it, copying objects from old space into new space. When it needs memory to copy into, a processor grabs a fresh partition from new space. When it fills a partition, it puts it on the queue to be scanned. The garbage collection starts with one processor scanning the root, but all of the processors are quickly engaged in scanning and copying (see figure 2). Garbage collection continues until all the processors are idle and the queue is empty.[10]

USER INTERFACE

The Butterfly Lisp User Interface is implemented on a Symbolics 3600-series Lisp Machine, communicating 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 are provided, associated with tasks running on the Butterfly. These windows may be 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. A version of the Lisp machine-based data Structure Inspector is also being adapted for examining task and data structures on the Butterfly.

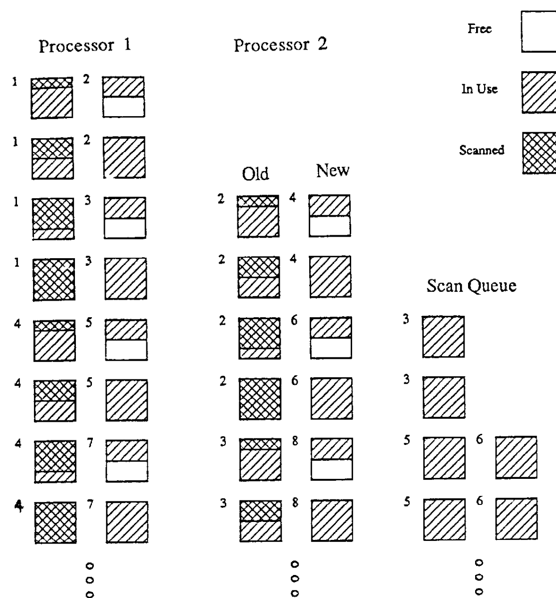

Figure 2: Parallel Garbage Collection - Copy and Scan

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 as a "face panel" to display the system state. This is information collected by a separate process, which spies on the Butterfly Lisp system. 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 (see figure 3).

This status pane also shows the number of tasks awaiting execution, the effective processor utilization, the rate of memory consumption, and an estimate of Butterfly switch contention. The graphical display makes such performance problems as task starvation easy to recognize.

FUTURE DIRECTIONS

Butterfly Lisp is currently interpreted. While this has been adequate during the development of various aspects of the system, such as the user interface and metering facilities, compilation is essential to the realization of the performance potential of the Butterfly. We are currently working on integrating a simple compiler, which will be used to explore the behavior of compiled code in our parallel environment. Later, we will substitute a more sophisticated compiler currently under development at MIT.

The User Interface will continue to be developed, aided by the experiences of a user community that is just beginning to form. We expect more facilities for both logical and performance debugging, with emphasis on graphical representations.

We will continue work already underway to provide compatibility with the Common Lisp standard.

We expect that Butterfly Lisp will become a testbed for exploring data structures and procedural abstractions designed specifically for parallel symbolic computing.

ACKNOWLEDGEMENTS

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

James Miller, for developing C Scheme, and for critical assistance in creating a parallel C Scheme for the Butterfly.

Anthony Courtemanche, for MultiTrash, our parallel garbage collector.

The MIT Scheme Team, especially Bill Rozas, Chris Hanson, Stew Clamen, Jerry Sussman, and Hal Abelson, for Scheme, their help, and advice.

Robert Halstead, for Multilisp, an excellent possibility proof, and for many enlightening discussions.

DARPA, for paying for all of this.
DARPA Contract Number MDA 903-84-C-0033.

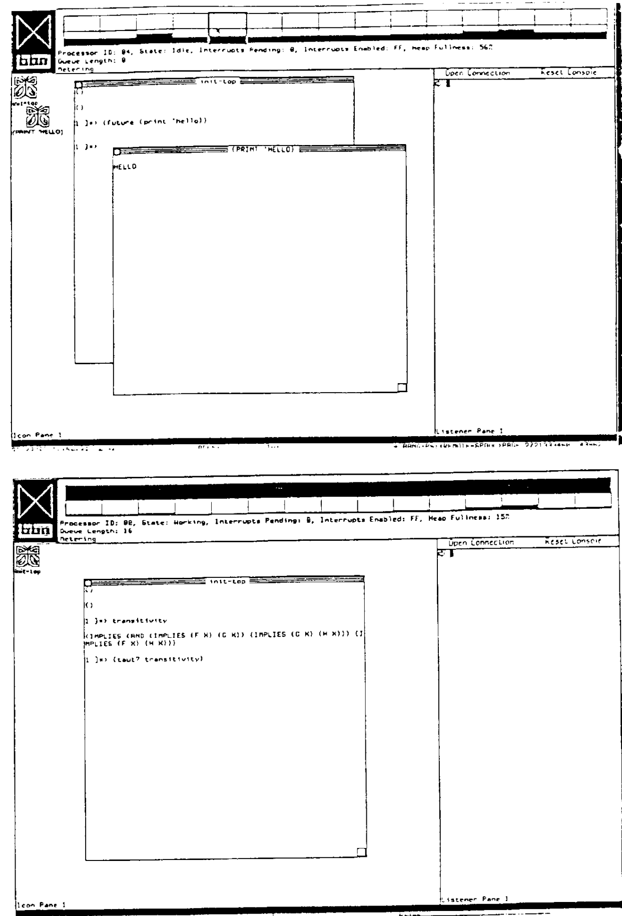

Figure 3: User Interface Display - Idle and Running

[1] Bolt, Beranek and Newman Laboratories
Butterfly Parallel Processor Overview
Bolt, Beranek and Newman
Cambridge Massachusetts
December, 1985

[2] Crowther, W., Goodhue, J., Starr, R., Milliken, W., Blackadar, T.
Performance Measurements on a 128-Node Butterfly Parallel Processor
Internal Bolt, Beranek and Newman Laboratories Paper

[3] Crowther, W.
The Uniform System Approach to Programming the Butterfly Parallel Processor
Internal Bolt, Beranek and Newman Laboratories Paper
December 1985

[4] Abelson, H. et al.
The Revised Revised Report on Scheme, or An UnCommon Lisp
AI Memo 848, M.I.T. Artificial Intelligence Laboratory
Cambridge, Massachusetts
August 1985

[5] Halstead, R.
Implementation of Multilisp: Lisp on a Multiprocessor
ACM Symposium on Lisp and Functional Programming
Austin, Texas
August 1984

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

[7] Gupta, A.
Talk at MIT
March, 1986

[8] Douglass, R.
A Qualitative Assessment of Parallelism in Expert Systems
IEEE Software
May 1985

[9] Bawden, A. and Agre, P.
What a parallel programming language has to let you say.
AI Memo 796, M.I.T. Artificial Intelligence Laboratory
Cambridge, Massachusetts
September 1984

[10] Courtemanche, A.
MultiTrash, a Parallel Garbage Collector for MultiScheme
M.I.T. B.S.E.E. Thesis
Cambridge, Massachusetts
January 1986