# Mul-T: A High-Performance Parallel Lisp

David A. Kranz* Robert H. Halstead, Jr.† Eric Mohr‡

## Abstract

Mul-T is a parallel Lisp system, based on Multilisp's `future` construct, that has been developed to run on an Encore Multimax multiprocessor. Mul-T is an extended version of the Yale T system and uses the T system's ORBIT compiler to achieve "production quality" performance on stock hardware — about 100 times faster than Multilisp. Mul-T shows that futures can be implemented cheaply enough to be useful in a production-quality system. Mul-T is fully operational, including a user interface that supports managing groups of parallel tasks.

## 1 Introduction

Mul-T is a parallel Lisp system that has been developed to run on an Encore Multimax multiprocessor. Mul-T is an extended version of the T version 3 (or "T3") system[17] that supports parallel processing using Multilisp's `future` construct[10,11,12]. Multilisp's implementation uses a layer of interpretation that limits its speed, but Mul-T uses T3's ORBIT compiler[14,15] (suitably modified) to generate native code for the Multimax's NS32332 processors, leading to a dramatic increase in speed (about a factor of 100 over the Multilisp system). Mul-T names both a parallel Lisp system and the parallel Lisp language it supports; where there is ambiguity, we refer explicitly to "the Mul-T system" or "the Mul-T language."

### 1.1 Parallelism in Mul-T

Mul-T (like Multilisp) is an extended version of Scheme[1,18], a lexically scoped dialect of Lisp. Mul-T's execution environment contains the same sorts of data types and primitive operators as Scheme or any Lisp dialect. In Mul-T, however, many lines of computation, or *tasks*, can be active simultaneously, manipulating objects in a single shared heap.

Mul-T's basic mechanism for generating concurrent tasks is the `future` construct. The expression (`future` $X$), where $X$ is an arbitrary expression, creates a task to evaluate $X$ and also creates an object known as a *future* to eventually hold the value of $X$. When created, the future is in an *unresolved*, or *undetermined*, state. When the value of $X$ becomes known, the future *resolves* to that value, effectively mutating into the value of $X$ and losing its identity as a future. Concurrency arises because the expression (`future` $X$) returns the future as its value without waiting for the future to resolve. Thus, the computation containing (`future` $X$) can proceed concurrently with the evaluation of $X$. When execution of a Mul-T program is not made explicitly parallel using `future`, it is sequential.

The result of supplying a future as an operand of some operation depends on the nature of the operation. *Non-strict* operations, such as passing a parameter to a procedure, returning a result from a procedure, assigning a value to a variable, and storing a value into a field of a data structure, can use a future as easily as any other kind of value, and take no special note of futures. *Strict* operations such as addition and comparison, if applied to an unresolved future, are suspended until the future resolves and then proceed, using the value to which the future resolved as though that had been the original operand.

The act of suspending if an object is an unresolved future and then proceeding when the future resolves is known as *touching* the object. The touches that automatically occur when strict operations are attempted are referred to as *implicit touches*. Mul-T also includes an *explicit* touching or "strict" primitive (`touch` $X$) that touches the value of the expression $X$ and then returns that value.

### 1.2 Overview of the Paper

Mul-T performs several functions whose implementations had to be either changed from or added to the

*M.I.T. Laboratory for Computer Science
†DEC Cambridge Research Lab
‡Yale University

standard T implementation: dynamic binding, storage allocation and garbage collection, task scheduling and management, the `future` construct, and implicit touches. These implementations had to be efficient and they had to fit into a parallel system. Additionally, the T system's user interface had to be redesigned to support users in managing and debugging programs containing many tasks.

This paper describes the problems in the abovementioned areas that were confronted in evolving T into Mul-T, and our solutions to those problems. The resulting system is a complete parallel Lisp system, including user interface. The performance data in Section 4 show Mul-T to be competitive with the best sequential Lisp implementations.

Section 2 describes the major implementation challenges that were confronted in building Mul-T, and Section 3 introduces an "inlining" optimization that can reduce the average cost of the `future` construct by avoiding the cost of task creation and future management when there are already enough tasks to occupy all the processors. Mul-T performance data is given in Section 4. Finally, Section 5 discusses related work and Section 6 offers some conclusions.

# 2 Implementation Challenges

Transforming a high-performance sequential Lisp implementation into a parallel one required many changes to the runtime system, compiler and user interface. We next describe the difficulties encountered in these three areas and how they were addressed.

## 2.1 Runtime System

The runtime system may be thought of as consisting of "kernel" functions (such as garbage collection and stack management) and "user library" functions (such as the procedures `append` and `map`). In this paper, we describe the modifications necessary to produce the "kernel" part of a parallel runtime system. For the most part, the production of parallel "user library" functions is not addressed here.

### 2.1.1 Global State

When converting any sequential program to a parallel one it becomes obvious that mutable global state in the sequential program should have been minimized and localized. The Mul-T runtime system is just a parallel T program derived from T3's runtime system. We were fortunate that this system has very little global state thanks to the diligence of Jonathan Rees.

The notion of a mutable global variable is different in a parallel program because a procedure that reads or writes the variable may be running on more than one processor at the same time. We can identify two kinds of mutable global variables that sequential programs may have:

- A mutable variable may have been made global to avoid passing it as an argument to many or all procedures in the program. In this case the variable should not be shared between instantiations.
- The variable may truly represent some global state of the program. In this case the variable must be shared by different instantiations; access to such a variable may need to be controlled by a semaphore or some other locking mechanism.

In a parallel program the first sort of global variable can cause problems, for example if each of several tasks depends on a private value of the variable. One can either eliminate the global by making it an explicit parameter of any procedure that references it, or the variable can be made "process specific", allowing each task to have its own copy. Mul-T provides support for process specific variables[1] and a syntax for specifying them. All global variables in the T3 runtime system had to be examined to see if they should be process specific (such as `*print-radix*`) or protected with a critical section (such as `symbol-table`).

### 2.1.2 Memory Management and Garbage Collection

One value which is normally represented by a global variable is the *heap pointer*. In Mul-T each processor allocates memory out of a chunk assigned to that processor. These chunks are replenished from a global heap when necessary. A garbage collection is triggered when there is no space left in the global heap. In addition, large objects are allocated directly from the global heap.

The use of chunks reduces contention for the global heap pointer by using a local heap pointer within a processor's current chunk for most allocations. The allocation of large objects directly from the global heap minimizes fragmentation inside of chunks. Since all memory is equidistant from all processors in a bus-based multiprocessor such as the Multimax, there is no penalty in loss of locality for allocating large objects outside of the chunk system. (The situation might be different in a multiprocessor with nonuniform memory access times, where it would generally be beneficial for *all* objects,

[1] by converting T3's dynamic binding mechanism from shallow to deep binding

large or small, allocated by a processor to be in the "chunk" of memory closest to that processor.)

To increase its speed, the garbage collector was parallelized. Increasing the garbage collector's speed was important because Mul-T can have independent "jobs" running concurrently and the garbage collector uses a stop-and-copy algorithm. If garbage collection took a long time, garbage collections triggered by "background" jobs running concurrently with the read-eval-print loop could cause long pauses and impair the quality of a user's interaction with Mul-T.

T3 uses a depth-first garbage collection algorithm based on [4] to help preserve locality of reference. The root set is the stack and a static data area. This algorithm was modified to work in Mul-T by dividing the static data area into segments and having a lock associated with each object to make sure it is only moved once. The steps are:

1. The processor which discovers that no space remains in the global heap interrupts all other processors (via a Unix signal) and waits for them to signal that they are ready. The other processors wait on a semaphore after signalling ready.
2. That original processor then signals the others to start collecting and it starts collecting as well.
3. Each processor roots from the task it was executing when the garbage collection was signalled and then processes segments from the static data area until they are exhausted.
4. The processors synchronize again and, when all have finished collecting, they continue from where they were interrupted.

The results of the parallel garbage collection are quite reasonable but might be improved. In the current algorithm, once an object is moved by a particular processor all of its components will be moved by the same processor. This might lead to an uneven distribution of work. We have not studied the general problem of parallel garbage collection in any detail.

### 2.1.3 Task Queues

As in other parallel Lisp systems, processors obtain Mul-T tasks from queues. Mul-T actually has two queues per processor, one for newly created tasks (the *new task queue*) and one for tasks ready to run again after blocking (the *suspended task queue*). An attempt is made to increase locality by scheduling a task which has already run on the processor it last ran on. This may be important to reduce turbulence in the Multimax's snoopy caches. When a new task is created it is put on the new task queue of the processor creating the task. When a blocked task is made ready to continue it is placed on the suspended task queue of the processor on which it was running when it blocked.

When a processor finishes a task it searches for another according to the following priority order:

1. It runs a task from its own suspended task queue.
2. It runs a task from its own new task queue.
3. It steals a task from the new task queue of another processor.
4. It steals a task from the suspended task queue of another processor.

This policy is a first cut at increasing locality and decreasing the rate of process migration. It does not address the issue of which task should be selected from several on a queue. At present this is simply done in a last-in-first-out manner.

## 2.2 Compiler

Two features of Mul-T require changes to the compiler:

1. The importance of supporting a large number of very lightweight tasks.
2. The implicit touches performed by strict operations.

Supporting these features means introducing overhead even in programs that don't make use of them. To a large extent, the effectiveness of Mul-T as a general purpose environment will depend on how small this overhead can be made.

In most sequential Lisp systems with compilers there is a single stack used by the system and user programs. Stack overflow is either not detected at all or some form of memory protection is used to generate a trap. Under Unix, a large number of stacks can be handled only by explicitly checking for overflow. The check need not be done on every push but must be done by every procedure that needs space on the stack. This check will involve at least a compare and a conditional branch.

A future is implemented as a data object with a particular type tag. Among its components are:

- A stack
- A slot to hold the eventual value being computed
- A representation of the process specific variables

- A queue for other futures waiting for the value being computed

Having strict operations such as + and car implicitly touch their operands means that a type test must be performed on each operand to make sure it is not a future. The cost of both this test and the stack overflow check could be absorbed by well known hardware technology, but they are a real problem on conventional hardware like the Encore Multimax which uses National Semiconducter NS32000 series processors. Implicit touches occur with great frequency and must be reduced to a compare tag and branch. On the NS32000, as well as on the Vax and Motorola MC68000, a cheap tag (bit field) compare can only be performed on a bit or byte. Using the high-order byte of a word as the tag field is not acceptable for a number of reasons, the most important being that 8 bits of the 32 bit address space are lost. Some sequential systems have used this scheme, but parallel programs will use much more space. The other choice is to use one of the low-order bits of a pointer as the future bit. We chose the low bit so code for `(eq? x y)`, where `x` is in `r1` and `y` is in `r2`, looks like:

```
     tbit $0,r1 ; test bit 0, the future bit
     beq  L1    ; if clear, not a future
     jsr  chase-future-in-r1
L1:  tbit $0,r2
     beq  L2
     jsr  chase-future-in-r2
L2:  cmp r1,r2
     ...
```

Many of these touch operations can be eliminated by having the compiler perform a simple first-order type analysis. For example, if a value has been tested once, it doesn't need to be tested the next time it is referenced. In several benchmarks the overhead without these optimizations was about 100%; with the optimizations it ranges from under 20% to nearly 100%; however, 65% seems to be a fairly typical number for programs that do not heavily emphasize iterative loops, such as the Boyer and compiler benchmarks (see Section 4 and the data in Table 4).

### 2.2.1 Futures

The introduction of the `future` construct to the language didn't require any changes to the compiler because (`future` *expr*) was simply transformed into (`*future (lambda ()` *expr*)), where `*future` is a procedure. The creation of the closure argument to `*future` automatically causes the free variables of *expr* to be copied into the heap, as they must be for *expr*'s execution as a separate task. `*future` creates a task containing the closure and puts it on the new task queue.

## 2.3 User Interface

Sequential Lisp systems are known for their strong program development environment. Extending this environment to handle multiple tasks gracefully presents some problems, which other parallel Lisp systems have been only partially successful in solving. Mul-T's solutions are based on the *group*, a collection of tasks resulting from the evaluation of a single expression typed by the user. Groups can be started, stopped, resumed, and killed independently of other groups of tasks. An important departure from conventional sequential Lisp implementation strategy is required to support groups well—the Mul-T runtime system uses distinguished tasks to control exception handling and access to terminal I/O.

Sequential Lisp systems handle an exception (error or keyboard interrupt) by suspending the original computation and invoking a *breakloop*, wherein the user has all the capabilities of the top-level read-eval-print loop. The user may examine the state of the computation, perform an arbitrary amount of other computations, and resume the original computation after correcting the error. If another exception occurs during interaction with the breakloop, a nested breakloop is invoked.

Thus there is a one-to-one correspondence between breakloops and stopped computations. The obvious parallel extension, followed by several other parallel Lisp systems, is for any task receiving an exception to invoke a breakloop. This can cause problems, for example when many tasks get the same error and several breakloops are invoked. In Multilisp, a *censor* is used to control which of multiple breakloops has access to the terminal, in Butterfly Lisp[2], a separate window is created to run each breakloop. In both cases the user must interact with an abundance of breakloops. And if related tasks continue to run, the source of the error may be obscured. MultiScheme[16] stops all tasks when an exception occurs, but the user can inspect only one task from the subsequent breakloop.

Groups in Mul-T allow a more natural parallel extension of the stopped computation idea. All tasks created during evaluation of an expression typed by the user belong to the same group $G$. If an exception occurs the group $G$ is stopped, suspending all of its constituent tasks. At this point the user regains control and may examine and alter the state of any of the stopped tasks. Even though several tasks may be stopped, the computation is represented by a single stopped group. There may be several stopped groups at a given time, analagous to the nested breakloops of sequential Lisps. The most recently stopped group is called the *current group*, and the task in which the exception occurred is called the *current task*. The usual Lisp debugging commands apply by default to the current task of the current group, so using Mul-T feels just

like using T. But the commands also allow referring to other tasks or other stopped groups.

The one-breakloop-per-task approach used by other parallel Lisps is a consequence of the natural implementation of breakloops in sequential systems: exceptions are handled by a procedure call to the breakloop routine, which executes using the stack of the task where the exception occurred. When control of the terminal is "welded on" in this way to tasks performing user computations, exception handling for multiple tasks becomes difficult.

It seems reasonable to unweld the breakloop by using multiple tasks. In Mul-T, control of the interactive terminal stream lies in a distinguished task separate from all tasks performing user computations. Further, there is a distinguished exception-handler task for each processor in the system. These special-purpose "server" tasks run only during exception handling or group termination, and coordinate with the Mul-T scheduler to insure that:

- After an exception is signalled by one task in a group, no other tasks in the group will run.
- Only one processor at a time may run the terminal control task.

A side-benefit of decoupling stopped computations is that the user may resume them in any order, as opposed to sequential Lisp systems where only the most recently suspended computation may be resumed.

# 3 Optimization of `future`

An opportunity to optimize the implementation of `future` arises out of the observation that, in the expression (`future` $X$), execution of the parent task concurrently with the evaluation of $X$ is *permissible* but is not *required*. It is thus permissible for an implementation to evaluate $X$ fully before proceeding with execution of the parent task: in other words, to treat `future` as an identity operator. We refer to this treatment as *inlining* because the expression $X$ is effectively evaluated "in line" as a subroutine, rather than concurrently as a separate task.

In many parallel Lisp programs, creating a task for every use of `future` leads to creation of tasks far in excess of the number of processors available to execute them. It is attractive to consider inlining these excess tasks, assuming (1) that we can accurately identify which tasks are "excess" and hence candidates for inlining and (2) that processing an inlined `future` is cheaper than processing a `future` that creates a concurrent task. The latter assumption is definitely true: inlining (`future` $X$) avoids the costs of setting up a new task (*e.g.*, allocating and initializing space for its stack), the queue management associated with scheduling the new task for execution, and even the cost of allocating and initializing a future object to act as a placeholder for $X$'s value. Further savings are realized when the value of the inlined expression (`future` $X$) is touched: since no future was ever actually created for $X$, touching the value of the inlined (`future` $X$) is no more expensive than touching the value of $X$ itself.

Identifying suitable places to apply inlining is the real challenge in applying inlining effectively. A simple strategy is for a processor to apply inlining to all `futures` encountered when the number of tasks on that processor's queues is greater than or equal to some threshold $T$. If $T = 0$, then *all* `futures` are inlined and no parallel tasks are created; if $T = 1$, then the existence of a single queued task will be enough to suppress task creation; and so on.

The rationale for this strategy is that these queued tasks represent a backlog of work that is available if any processor becomes idle. If this backlog is large enough, there is little point in adding to it. On the other hand, one can speculate that it might be desirable to set $T$ somewhat larger than 1 to provide a modest buffer against variations in the rate at which `futures` are encountered: if several processors became idle within a short period of time, one would like to have "saved up" enough tasks to keep these processors busy, so they will not have to wait for new tasks to be created before resuming useful work.

This inlining strategy can be ineffective due to *task starvation* caused either by *bursty task creation* or *parent-child welding*. *Bursty task creation* refers to the fact that opportunities to create tasks may be distributed unevenly across a program. At the moment when a task is inlined, it may appear that there are plenty of other tasks available to execute, but by the time these tasks finish executing, there may be too few opportunities to create more tasks. Consequently, processors may go idle that could have been kept busy if less inlining had been done earlier in the program's execution. *Parent-child welding* refers to the fact that inlining effectively "welds" together a parent and child task. If an inlined child becomes blocked waiting for a future to resolve (or for some other event), the parent is blocked as well and is not available for execution. If, on the other hand, a non-inlined child blocks, then the parent is still runnable.

In some extreme cases, parent-child welding can even result in deadlock. Consider the following odd program fragment, where `make-semaphore` creates a semaphore and `semaphore-p` and `semaphore-v` perform the usual P and V operations on a semaphore:

```
(let ((s (make-semaphore))
      (x 0))
  (semaphore-p s)    ; lock the semaphore
  (let ((a (future (begin (semaphore-p s)
                          (+ x 1)))))
    (set! x (f 17)) ; compute x
    (semaphore-v s) ; now allow access to x
    (+ a 1)))       ; await a and return
```

The code for the future will block pending the `semaphore-v` operation on `x`; but if the future is inlined, then the `semaphore-v` operation will never take place. Fortunately, such deadlock can never occur in the simple case of Mul-T programs generated by starting from a side-effect-free sequential program and just wrapping `future` around selected subexpressions; nevertheless, whether or not the program given above is viewed as pathological, the fact that inlining can cause it to deadlock where it would otherwise produce an answer is a concern.

The incidence of task starvation due to inlining varies widely from one program to the next: inlining is quite effective on some programs and actually degrades the performance of others (even when the deadlock issue does not arise). Inlining is implemented in Mul-T, and we have been able to measure its benefits in the cases where it works well (see Section 4) and observe the degradation when it does not work well. In the cases where inlining works well, the benefits are large enough that we have been driven to speculate about how to retain inlining's benefits without its liabilities. The result of this speculation is a mechanism called *lazy futures*, which is not implemented in the current Mul-T system. We briefly outline the idea here, however, and hope to report on it in more detail after further work.

Lazy futures is essentially a revocable inlining mechanism: when a `future` is encountered, its task is provisionally inlined, but enough information is retained to enable the inlining decision to be retroactively reversed at a later time. To allow this, tasks must have a stack structure such that the "seam" between the portions of stack pertaining to the parent and child task executions can be found, and that the stack can be split apart at that point to yield two independent stacks—one for the parent and one for the child—even after the child has gone some distance into its execution. This provides a way to "unweld" a blocked child from its parent so that the parent can resume execution, thus solving the associated problems discussed above, including the deadlock problem. Even a running child can be "unwelded" from its parent, furnishing an additional task if it is needed because some other processor has just finished a task.

A system with lazy futures can provisionally inline *every* future and split a task whenever a processor needs a new task to execute. If there were no extra cost to making inlining decisions revocable, this would offer the best of both worlds: the performance advantages of inlining would be obtained in every situation except where task splitting yielded additional needed parallelism. Quantifying the costs and benefits of lazy futures more precisely, however, requires a careful implementation study.

# 4 Performance of Mul-T

Mul-T's performance can be understood both by examining the cost of individual operations and by studying the performance of entire application programs. A useful "benchmark" for the former is an expression such as `(touch (future 0))`, whose execution involves exactly one creation of a task and a future and exactly one synchronization operation on the future. In more detail, the operations performed in evaluating this expression are

1. Make the thunk (in this case, `(lambda () 0)`) to be executed in the child task, and call `*future`.
2. Create a future, create a child task, and enqueue the child task (on the new task queue) for execution.
3. Block the parent task (thanks to the `touch` operation) and enqueue it to await resolution of the future.
4. Dequeue the child task and begin to execute it.
5. Resolve the future and enqueue any tasks waiting on that future (in this case, just the original parent task) on the suspended task queue of the processor where they were executing just before they blocked.
6. Dequeue the original parent task and resume its execution.

Table 1 gives the number of NS32332 instructions needed for each of these steps. (The figures for steps 4 and 6 are approximate since processors that are idle and waiting for an executable task to be enqueued will not always be in the same phase of their waiting loop at the moment when an executable task appears. Moreover, the figures for steps 2–6 assume no lock contention will occur.) The experimentally measured execution time for the approximately 196 instructions required to execute the expression `(touch (future 0))` is about 220 $\mu$sec when executed on one processor. This suggests that the NS32332 is delivering about 1 MIPS on this instruction mix, which emphasizes data structure manipulation and memory-to-memory instructions to a

| | |
|---|---|
| 1. Make thunk and call `*future` | 15 |
| 2. Create future and task; enqueue task | 41 |
| 3. Block touching task | 33 |
| 4. Dequeue and start executing a task | 37 |
| 5. Resolve a future and enqueue waiters | $26 + 14w$ |
| 6. Dequeue interrupted task and resume | 30 |

Table 1: Cost of Mul-T future operations, in NS32332 instructions; $w$ is the number of waiting tasks restarted in step 5.

greater extent than a typical computational instruction mix does.

For comparison, a call to (and return from) the trivial procedure `(lambda () 0)` takes 8 instructions. The approximately 25:1 ratio between the execution times of `(touch (future 0))` and `((lambda () 0))` contrasts with about a 3:1 ratio between the costs of evaluating the same two expressions in Multilisp, where a great deal of interpretive overhead is added to both operations but the impact of the overhead on the procedure call is proportionately greater.

This example actually gives a pessimistic estimate of the overhead associated with `future`. In many cases no tasks will block on a future, reducing the overhead to approximately 119 instructions.

To evaluate Mul-T's performance on a more realistic program, we ran a modified version of the Boyer theorem-proving benchmark from the Gabriel book of Lisp benchmarks[6]. The modifications involved removing some global side effects, yielding a cleaned-up, sequential Boyer benchmark, and then using futures to create a parallel program. As Table 2 shows, the cleaned-up, sequential Boyer benchmark takes 14.5 seconds to run on the unmodified, sequential T3 system. Since the performance of compiled code in the T3 system is about as good as that of any other compiled code on the same hardware[14], we view this as a good estimate of top speed for this application on the Multimax's NS32332 processor. The same sequential code, when run under Mul-T with touch optimizations disabled, takes twice as long to run. Since this code contains no futures, the increase in running time is attributable solely to implicit touches. Although every one of these touches reveals that no action is necessary, they occur frequently enough to double the execution time of the program. When the touch optimizations discussed in Section 2.2 are enabled, the time for the cleaned-up, sequential Boyer benchmark is reduced to 24 seconds, representing a reduction in the overhead due to touch checks from 100% of the execution time on T3 to 65%.

| | |
|---|---|
| T3 | 14.5 sec |
| Mul-T, no touch optimizations | 29 |
| Mul-T plus touch optimizations | 24 |

Table 2: Performance of cleaned-up, sequential Boyer benchmark.

| Number of processors: | 1 | 2 | 4 | 8 |
|---|---|---|---|---|
| Without inlining ($T = \infty$) | 44 | 23 | 12 | 7.5 sec |
| With inlining ($T = 1$) | 25 | 13 | 7 | 4 |

Table 3: Performance of parallel Boyer benchmark.

Table 3 shows the performance of the parallel Boyer benchmark for different numbers of processors and different settings of the inlining threshold. Touch optimizations were enabled during these measurements. Without inlining (*i.e.*, when the inlining threshold $T = \infty$) the execution time on one processor was 44 seconds—20 seconds more than the corresponding time in Table 2. This represents the extra overhead introduced by the use of `future` in the parallel benchmark. The execution times given in Table 3 for 2, 4, and 8 processors, however, clearly indicate that Mul-T has successfully exploited a substantial amount of parallelism in the benchmark, so that the execution time on 4 and 8 processors is less than that of the sequential benchmark on T3.

The second line of Table 3, which was obtained by setting the inlining threshold $T = 1$, offers further encouragement. Inlining is seen to be an extremely effective optimization for the Boyer benchmark, reducing the extra execution time on one processor due to introducing `future` from 20 seconds to one second. Moreover, the execution times given for 2, 4, and 8 processors show that Mul-T has continued to be successful in exploiting parallelism, bettering the T3 performance even when only two processors are used.

Table 4 shows timings of four other Mul-T programs. The first line ("seq") shows the time of a sequential version of each program in T3; subsequent lines show Mul-T times for increasing numbers of processors. Our timings were somewhat variable because Mul-T's "processors" are really Unix processes, subject to interruptions by the UMAX (Unix) scheduler. The figures shown are averaged over 5 or 10 successive trials; typically variation did not exceed 5%.

**permute** finds a set of 10,000 vectors of 20 integers each, such that any two vectors chosen from the set differ in at least 10 positions. The integers range from 0 to 31 and comprise four independent permuations of

| $n$ | permute | queens | compiler | mergesort measured | mergesort theoretical |
|---|---|---|---|---|---|
| seq | 8520 | 27.8 | 98 | .99 | |
| 1 | 11554 | 33.2 | 159 | 1.82 | (1.82) |
| 2 | 5823 | 16.6 | 94 | .99 | .98 |
| 4 | 2995 | 8.5 | 64 | .57 | .60 |
| 8 | 1598 | 4.3 | 53 | .45 | .42 |
| 12 | 1293 | 3.0 | 54 | .43 | |

Table 4: Execution time (in seconds) for Mul-T benchmarks.

eight numbers taken five at a time. The program generates new candidates using a pseudo-random number generator. Each candidate thus generated is compared against every vector already accepted into the set, and is only accepted into the set if it passes the distance criterion in every comparison. The comparison process for each candidate is broken up into parallel tasks, each of which compares the candidate against 40 vectors in the already accepted set. Moreover, up to 16 vectors can simultanously be tested against the accepted set, and as soon as any candidate is either accepted or rejected, another candidate is generated so testing can continue[20]. The result, as seen in Table 4, is that plenty of parallelism was generated with moderate overhead, even though no inlining was used ($T = \infty$).

**queens** finds all solutions to the n-queens problem for $n = 11$. The numerous branches of this search program are independent, although the costs of computing the branches are very uneven. In the version used here, a task was created for each possible pair of positions in the first two rows. This results in $n^2 = 121$ tasks of large granularity, so inlining was not used. The speedup is close to linear; the small difference is probably due to the large task granularity, meaning idle processors toward the end of the computation.

**compiler** is a transformation-based compiler developed by Richard Kelsey [13]; here it is compiling a Pascal program with 21 procedures. The parallel compiler consists of a sequential parsing phase, a compilation phase where each procedure in the file is handled by a separate task, and a sequential output phase. The compiler contains roughly 20,000 lines of T code; the Mul-T version was created in about 3 days. The ease of this translation is as much due to careful, mostly-functional coding in the sequential version as it is to the simplicity of the Mul-T primitives.

The timings show successful parallel execution of the compiler, although several factors limit the speedup—the sequential parse and output phases, uneven loads due to the small number of tasks, and the fact that currently only one task at a time may use the assembler.

**mergesort** performs a destructive merge sort on a list of 8192 integers. The execution pattern of this divide-and-conquer sort is independent of the input data: a list is divided into two sublists, which are recursively sorted in parallel and then merged together. An analysis of this algorithm for $2^l$ processors sorting $n = 2^k$ elements predicts an execution time $t$ of:

$$t(k,l) = \mathcal{O}[(k-l-2)2^{k-l-1} + 2^k]$$

As expected, this reduces to $\mathcal{O}(nk)$ for $l = 0$ (*i.e.* $\mathcal{O}(n \log n)$ on one processor) and $\mathcal{O}(n)$ for $l = k$ (even with $n$ processors, the linear data structure must be traversed). Given the time for **mergesort** on one processor, the above equation was used to predict execution times for 2, 4, 8, and 16 processors, also shown in Table 4. The analysis shows that Mul-T is correctly exploiting the parallelism in **mergesort**. Inlining ($T = 1$) is crucial to good performance in this benchmark, reducing the number of futures created from 8191 to, for example, 350 on 8 processors.

Of the five benchmarks, **mergesort** compares least favorably to its sequential T version, running about twice as slowly in Mul-T on one processor. This is because **mergesort** contains several tight loops into which the compiler must insert implicit touches. Some of the touches are not strictly necessary, and we are considering further compiler optimizations to eliminate them.

# 5 Related Projects

The Mul-T project builds on the results of several other projects. Aside from the obvious debts to the developers of T3 and Multilisp, Mul-T borrows from the work of Carl Hewitt and Henry Baker, who first articulated the concept of futures[3]. Another early parallel Lisp system, which was developed for the BBN Butterfly Machine[5], is MultiScheme[16]. Like Multilisp, MultiScheme used a layer of interpretation in its implementation (although there have been plans to add a compiler) and thus was not fast in absolute terms.

We are aware of three other parallel Lisp systems that have been implemented using a compiler for high performance: Butterfly Lisp[2], implemented for the Butterfly Machine; Qlisp[7,8,9], implemented for the Alliant FX-8; and a parallel version of Portable Standard Lisp[19] (let us call this "PPSL"), also for the Butterfly Machine. Butterfly Lisp and PPSL, like Multilisp and Mul-T, rely primarily on `future` as a concurrency mechanism. Qlisp supports futures but also provides several other concurrency constructs.

Butterfly Lisp, like Multilisp and Mul-T, includes implicit touches for strict operations, but Qlisp and PPSL require programmers to insert explicit `touch` operations whenever a future might appear as an operand to a strict operation (the implementors of Qlisp are in the process of incorporating implicit touches). As Section 4 shows, implicit touches are expensive on stock hardware, but inserting explicit touches will be tedious and error-prone for programs that use futures heavily.

No performance figures have been published for Butterfly Lisp, but figures are available for Qlisp[9] and for PPSL[19]. Since both the Qlisp and PPSL figures are for Lisps without implicit touches, they offer no basis for comparison with the approximately 65% overhead for implicit touches in Mul-T. As far as task management overhead itself is concerned, Mul-T's cost of approximately 200 instructions for `(touch (future 0))` compares with 1400 instructions to create a process in Qlisp[9], which may actually represent only the first part of the processing required to evaluate `(touch (future 0))`. PPSL requires approximately 470 $\mu$sec to create and execute a future containing a trivial procedure call on one processor of a Butterfly Plus multiprocessor, which may be around 50% faster than one NS32332 processor[19].

These performance differences may better reflect the priority attached by implementors of different systems to optimizing different parts of their implementations than any inherent differences in the cleverness of particular parallel Lisp implementations. Nevertheless, the lower cost of futures in Mul-T than in these other systems leads to a more optimistic view of the potential benefits of using futures for "production quality" parallel computing on stock multiprocessors.

# 6 Conclusion

Among the unknowns we faced before undertaking the Mul-T project was the question of what performance to expect, especially for highly parallel programs. Multilisp's layer of interpretation tended to hide the cost of task management and implicit touches under a mountain of interpretive overhead. The lurking question, never definitively answered by experience with Multilisp, was whether a system not burdened by interpretive overhead would find the cost of managing futures to be prohibitive. Mul-T is generally a conservative extension of T: Mul-T's strategies for dealing with futures and tasks are generally fairly simple and straightforward. It is thus encouraging to note that we have been able to implement futures at a moderate cost.

The development of Mul-T has been valuable in several ways. First, Mul-T is a complete, working parallel Lisp system, publicly available to interested users. Second, its single-processor performance is competitive with that of "production quality" sequential Lisp implementations, and therefore a parallel program running under Mul-T can show absolute speedups over the best sequential implementation of the same algorithm. This is attractive to application users whose primary interest is raw speed rather than the abstract gratification of having demonstrated speedup via a time-consuming simulation. Finally, implementing Mul-T has allowed us to experiment with and evaluate implementation strategies such as inlining. The Mul-T experience has also allowed us to probe the limits of implementing futures on stock multiprocessors, and has suggested (for example) that hardware assistance for tag management may be a more significant benefit in a machine for parallel Lisp (where it can eliminate the 65% overhead of implicit touches) than it has ever proven to be in machines for sequential Lisps.

# 7 Acknowledgments

We would like to thank Encore Computer Corporation for sponsoring Rick Mohr's summer visit at M.I.T. to work on Mul-T, and thank Paul Hudak for supporting Rick in taking a summer off from Yale for this project. We also thank M.I.T. for providing a home for Bert Halstead during the period when this work took place. The other members of the Parallel Processing Group at the M.I.T. Laboratory for Computer Science created a pleasant environment for Mul-T's development and graciously contributed their own time and energy to porting applications and developing Mul-T profiling tools. Richard Kelsey was kind enough to port his compiler to Mul-T. We are also grateful to Gautam Thaker of the Harris GSS Advanced Technology Department and to Dan Nussbaum of M.I.T. for their work on the `permute` benchmark. This research was sponsored in part by the Defense Advanced Research Projects Agency of the United States Government and monitored by the Office of Naval Research under contract number N00014-84K-0099.

# References


[1] Abelson, H., and G.J. Sussman with J. Sussman, *Structure and Interpretation of Computer Programs*, MIT Press, Cambridge, 1985.

[2] Allen, D., S. Steinberg, and L. Stabile, "Recent developments in Butterfly Lisp," *AAAI 87*, July 1987, Seattle, pp. 2–6.

[3] Baker, H., and C. Hewitt, "The Incremental Garbage Collection of Processes," M.I.T. Artificial Intelligence Laboratory Memo 454, Cambridge, Mass., Dec. 1977.

[4] Clark, D.W., "An efficient list-moving algorithm using constant workspace," *Communications of the ACM 19(6)*, pages 352-354, June 1976.

[5] Crowther, W., *et al.*, "Performance Measurements on a 128-Node Butterfly Parallel Processor," *1985 Int'l. Conf. on Parallel Processing*, St. Charles, Ill., Aug. 1985, pp. 531–540.

[6] Gabriel, R., *Performance and Evaluation of Lisp Systems*, M.I.T. Press, Cambridge, Mass., 1985.

[7] Gabriel, R.P., and J. McCarthy, "Queue-based Multi-processing Lisp," *1984 ACM Symp. on Lisp and Functional Programming*, Austin, Tex., Aug. 1984, pp. 25–44.

[8] Gabriel, R.P., and J. McCarthy, "Qlisp," in J. Kowalik, ed., *Parallel Computation and Computers for Artificial Intelligence*, Kluwer Academic Publishers, Boston, 1988, pp. 63–89.

[9] Goldman, R., and R.P. Gabriel, "Preliminary Results with the Initial Implementation of Qlisp," *1988 ACM Symp. on Lisp and Functional Programming*, Snowbird, Utah, July 1988, pp. 143–152.

[10] Halstead, R., "Multilisp: A Language for Concurrent Symbolic Computation," *ACM Trans. on Prog. Languages and Systems 7:4*, October 1985, pp. 501–538.

[11] Halstead, R., "Parallel Symbolic Computing," *IEEE Computer 19:8*, August 1986, pp. 35–43.

[12] Halstead, R., "An Assessment of Multilisp: Lessons from Experience," *Int'l. J. of Parallel Programming 15:6*, Dec. 1986, pp. 459–501.

[13] Kelsey, R., and P. Hudak, "Realistic Compilation by Program Transformation," *1989 ACM Symp. on Principles of Programming Languages*, Austin, Texas, January 1989, pp. 281–292.

[14] Kranz, D., *et al.*, "Orbit: An Optimizing Compiler for Scheme," *Proc. SIGPLAN '86 Symp. on Compiler Construction*, June 1986, pp. 219–233.

[15] Kranz, D., "ORBIT: An Optimizing Compiler for Scheme," Yale University Technical Report YALEU/DCS/RR-632, February 1988.

[16] Miller, J., *MultiScheme: A Parallel Processing System Based on MIT Scheme*, Ph.D. thesis, M.I.T. E.E.C.S. Dept., Cambridge, Mass., August 1987.

[17] Rees, J., N. Adams, and J. Meehan, *The T Manual*, fourth edition, Yale University Computer Science Department, January 1984.

[18] Rees, J., and W. Clinger, eds., "Revised$^3$ Report on the Algorithmic Language Scheme," *ACM SIGPLAN Notices 21:12*, Dec. 1986, pp. 37–79.

[19] Swanson, M.R., R.R. Kessler, and G. Lindstrom, "An Implementation of Portable Standard Lisp on the BBN Butterfly," *1988 ACM Symp. on Lisp and Functional Programming*, Snowbird, Utah, July 1988, pp. 132–142.

[20] G.H. Thaker, D.B. Bradley, D. Nussbaum, "Comparative Study of a Parallel Algorithm," Harris Concert Application Note 31, March 89.