Compiling Parallel Lisp for a Shared Memory Multiprocessor

M.D. Feng   W.F. Wong   C.K. Yuen

Dept. of Information Systems and Computer Science,
National University of Singapore,
Lower Kent Ridge Road,
Singapore 0511.
Republic of Singapore.

Abstract

There is a commonly held idea that although Lisp is a good language for prototyping and software development, it is often too inefficient to be useful in actual implementation. Advances in compiling Lisp have begun to change this notion. However, the situation is less clear in the case of parallel Lisp dialects. In this paper, we report on our experience in implementing a compiler for a parallel Lisp dialect we called *BaLinda Lisp* for a shared memory multiprocessor. For a class of problems, our compiler was able to achieve performances on Lisp programs that are near, if not equal to, their imperative (C) equivalents. However, on other problems, the situation is less than ideal. We investigate this phenomenum and outline the future challenges in realizing a production strength parallel Lisp compiler.

1 Introduction

For symbolic computation, the major 'workhorse' has been the Lisp family of programming languages. However, there is a common view that while Lisp is a good language for prototyping and software development as well as symbolic computation, it is inefficient in actual implementation. Advances in compiling Lisp have begun to disprove this. Attention has turned also to the possibility of performing symbolic computing on multiprocessors by parallelizing Lisp, though research has still to reach the stage of maturity as compared to, say, parallel Fortran.

In this paper, we report on our experiences in constructing a compiler for the parallel Lisp dialect *BaLinda Lisp* which runs on a SPARC-based shared memory multiprocessor. The reported work extends our earlier work on compiling BaLinda Lisp for a distributed environment [3]. The compiler first uses a series of transformations to transform and optimize the code and then generate SPARC assembly codes.

In this paper, we first show that our compiler's performance is superior to previous implementations. To this end, we tested a number of programs. For comparison, the same programs were executed on a distributed network of Transputers. In addition, we also executed the C equivalent of these programs as well as the equivalent code generated by Scheme->C, a public domain translation software that generates C codes from Scheme programs. Results will be presented and discussed. We shall also try to compare our compiler with Mul-T [10], despite some difficulty in comparing the implementations on machines of divergent characteristics. It is shown that BaLinda Lisp is at least competitive to the other parallel Lisp dialects.

Our second aim is to outline some of the problems in making parallel Lisp competitive to other parallel programming languages. While on certain programs, our compiler performed very well, it was disappointing on others. We shall offer some reasons which perhaps will point to the challenges ahead for parallel Lisp research.

2 BaLinda Lisp

BaLinda Lisp is a sequential Lisp dialect augmented with the Linda [2] tuple operations. The system assumes a shared data space containing multi-field records or *tuples*. A task puts a tuple into the *tuplespace* using an `OUT` command:

```
(OUT exp1 exp2 ... expN)
```

where each expression defines a value of any type, whether numerical, logical, text or even array/list. Tuples are not accessed by name or address, but by content, using the `IN` or `RD` commands:

```
(IN exp1 exp2 ... expM ? name1 name2 ... nameN-M)

(RD exp1 exp2 ... expM ? name1 name2 ... nameN-M)
```

These will retrieve from the tuplespace an `N`-field tuple whose first `M` fields match the result of the `M` expressions in the `IN/RD`, and then store the values of the last `N-M` fields into the `N-M` variables specified in the `IN/RD`. `IN` causes the tuple to be removed from the space, while `RD` leaves it for others to access. If no matching tuple can be found, the task executing the `IN/RD` is suspended until another task `OUT`'s the required tuple.

However, we have decided to eliminate the `EVAL` construct from the set of Linda operations included in BaLinda Lisp. Instead parallel tasks are spawned off with the `EXEC` construct. The following

```
( <pre-expression>
  (EXEC <expression>)
  <sibling-expressions> )
<post-expressions>
```

causes `<expression>` to be executed in parallel with `<sibling-expressions>`. When both complete, execution proceeds to `<post-expressions>`. This frees the programmer from the compulsory generation of a tuple upon completing an `EVAL` which we did not find useful and indeed somewhat bothersome.

In addition to Linda operations, BaLinda Lisp introduces *speculative processing* in the `COND` construct. However, this is not within the scope of this paper and we refer the reader to [14] for further discussions on this and other issues of the language.

3 Previous Works

The compilation techniques pioneered in research done on compiling Scheme have been important to our work. In particular, the work on RABBIT [12], ORBIT [9] and HARE [13] compilers has been influential on our work.

On parallel Lisp research, a number of parallel Lisp dialects has been proposed. A good compilation of papers on the subject is found in [7]. With the exception of Multilisp [4], performance results for implementations of most of the parallel Lisp dialects are scarce. One work which is closely related to ours is the modification of the ORBIT compiler to compile Mul-T [10], a variant of Multilisp for the Encore Multimax multiprocessor. The programming model for Multilisp and Mul-T is the *future* and is therefore very different from BaLinda Lisp. Despite the differences in platform, we will show evidence that our implementation is faster than Mul-T.

While the above works are significant and important, none provides us with evidence that parallel Lisp can be a viable competitor to conventional parallel imperative languages. In this paper, we shall explore this possibility by examining the state of the art in compiling Lisp both as a sequential as well as a parallel language. We shall then investigate the problems involved using our compiler for BaLinda Lisp as the subject.

4 Compiling to the Abstract Machine

BaLinda Lisp programs are first compiled to an abstract machine. The idea of using an abstract machine is to allow for portability across a number of platforms. So far we have been able to port the compiler to a distributed memory multiprocessor, namely a network of Transputers, as well as a shared memory multiprocessor, which is what will be reported in this paper. Details of the abstract machine is described in [3].

The actual compilation is done in a number of phases which are by now quite commonly found in Lisp compilers :

1. *Macro expansion phase.* Most of the syntax analysis of the compiler is done in this phase. Derived expressions are transformed into their equivalent primitive expressions. For example, `COND` is transformed into a sequence of `IF` expressions. In addition, parallel and tuplespace operations are transformed into primitive expressions which correspond to the abstract machine instructions.

2. *Conversion to CPS form.* Macro expanded expressions are converted in equivalent *continuation-passing style* [12] expressions. The advantages of the CPS conversion is that code becomes regularized and tail calls are made explicit.

3. *Optimization.* In this phase various optimization transformations are applied. These include β-reduction, boolean expression short-cutting, constant test evaluation and test result propagation, constant folding and redundant subexpression elimination.

4. *Closure analysis.* This determines if a variable is to reside in the heap or the stack and decides what sort of information is to be stored in the environment.

5. *Abstract machine code generation.* Finally, abstract machine code is

generated. The abstract machine is an accumulator-based stack machine. Its instruction set corresponds roughly to the primitive expressions of BaLinda Lisp.

To these phases, one can attach a final code generation phase to generate actual machine code from the abstract machine code.

5 Compiling for a Shared Memory Multiprocessor

The compiler compiles BaLinda Lisp codes for a Sun SPARCserver 1000 with 8 CPUs, running at 50 MHz, 512MB of RAM and 20GB of disk space running under SunOS 5.3 operating system. This is a shared memory multiprocessor which supports both the traditional Unix Inter-Process Communication constructs as well as multithreading with lightweight processes.

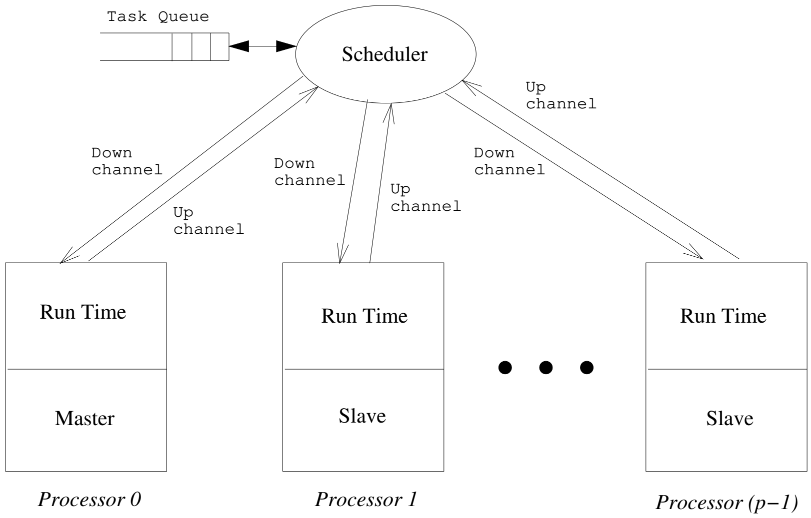

Figure 1: The Runtime Environment.

Fig. 1 shows the runtime environment of BaLinda Lisp. There are p logical processors. All but one execute the **slave** program which is one of the results of the compilation. This is the compiled user program linked with the support runtime routines. All the slave programs have identical codes. There will be one processor specifically designated to run the master program. The user can choose to program in a SPMD (Single Program Multiple Data) style, in which the master will be exactly the same as the slaves, or in a 'master-slave' style, in which case the master program will be different. In the latter, the master typically divides up work which is then given to the slaves for execution.

Each processors is linked up with the scheduler via two logical one-way communication channels. The scheduler has two main functions. First, when a processor (i.e. either the master or one of the slaves) performs an `EXEC`, the function to be `EXEC`'ed and its arguments are sent to the scheduler which then determines which of the processors is idle and will send this `EXEC` request via the down channel to that processor for execution. An idle processor will first send a message to the scheduler informing it of its status and then waits on the down channel for work from the scheduler. Upon receiving work, it will look up a symbol table for the address of the function to invoke and will then invoke the function using the arguments received via the down channel from the scheduler.

The tuplespace, which is shared among all processors, is also implemented in the scheduler. Using the channels, tuplespace operation requests are sent from the processors to the scheduler which will perform the search and match operations, and return the result(s) via the channels to the requesting processor.

On the Sun SPARCserver 1000, each logical processor is bounded to a physical processor with the scheduler, which is not frequently invoked, existing as an unbounded process free to run on any physical processor scheduled under the operating system. It turned out that the strategy used to implement the logical channels is crucial to performance as this is heavily used during parallel and tuplespace operations. We tried several strategies including sockets, pipes and shared memory with semaphores. Each improved on the performance of the former. Finally we settled on spin-locking using assembly level atomic operations, namely the `SWAP` instruction of the SPARC instruction set, on shared memory which gave us the best performance. Table 1 compares various methods of implementing some basic operations. The Ping-Pong program involves four `IN` and `OUT` operations. The empty `EXEC` attempts an `EXEC` on a function which returns immediately. As a comparison, we listed the corresponding timing for the implementation using Transputers. The Transputer system [6] which we shall quote throughout this paper consists of 1 root T805 running at 25 MHz with 17 T800 running at 20 MHz. Each Transputer has 4 bidirectional hardware communication channels operating at 10 Mbits/sec data rate. All of the timings reported in Table 1, however, involves only two processors. An important caveat for all our benchmark numbers is that the SPARCserver 1000 we used typically have over 100 users logged in. While we tried our best to take the timing measurements when the system workload is low, seldom is there less than 10 users logged in. The times reported are the real time returned by the Unix `gettimeofday` system call. The lowest observed time is reported.

	Unix Pipes	**Shared Memory with Semaphores**	**Shared Memory with SWAP**	**Transputer**
Ping Pong	8.5	2.5	0.14	1.9
Single `IN`	4.9	1.3	0.07	0.6
Empty `EXEC`	9.3	3.9	0.121	1.7

Table 1: Implementation Strategy for Logical Channels. (Time in Milliseconds)

While the Transputer has a slower clock than the Sun SPARCserver 1000, it does have hardware support for its channel operations which makes them fast. Despite this we were able to achieve about 10 times better performance on our shared-memory implementation. We note that an empty EXEC takes about 121 μsec. The equivalent operations takes 220 μsec and 470 μsec in Mul-T and Portable Standard Lisp on the BBN Butterfly machine [10]. In our view, 121 μsec for EXEC is still not fast enough. However, the BaLinda Lisp programming model generally gives rise to medium to large granularity, thereby alleviating the impact of slower task creation. We shall elaborate this point further in the next section.

6 Performance Results and Comparisons

In this section, we shall discuss the performance of our compiler. We shall first try to establish the performance of our compiler on sequential codes without floating point arithmetic operations. Integers, together with a two bit tag, can be represented directly by a word as a cons cell pointer, making them very efficient to operate on. We shall then discuss the performance of our compiler on parallel code. Lastly, we analyze the performance problems of our implementation and outline the future work to be done.

6.1 Performance on Sequential Integer Code

Table 2 shows the performance of the two versions of the compiler in compiling sequential code. The aim is to ascertain that the compiler generate good sequential code. The benchmarks used include Tak(24 16 8), Fibonacci ($n = 30$) and Quicksort ($n = 4096$). The comparison is done with :

- the compiler on the Transputer (which must be taken with care as the two machines differ significantly);
- the Scheme->C compiler [1] which translates Scheme programs to C with all optimization options on and compile with "−O2" for GCC C optimization;

- and the sequential C equivalent programs compiled with the native SPARCompiler C 2.0.1 with the "−xO4" optimization option turned on.

	BaLinda Lisp on SPARCserver	**BaLinda Lisp on Transputer**	**Scheme->C**	**C**
Fibonacci	1.155	6.428	1.211	1.205
Tak	0.761	6.889	0.819	0.751
Quicksort	0.017	0.199	0.073	0.015

Table 2: Performance Compared to (Time in Seconds)

As can be seen, our compiler produces code with very competitive performance even against straightforward C equivalents. This is due mainly to the advances in compiling techniques for Lisp.

Program	**1 Processor**	**2 Processors (Speedup)**	**4 Processors (Speedup)**	**8 Processors (Speedup)**
Fibonacci ($n = 34$)	7.92	5.36 (1.47)	2.70 (2.94)	1.34 (5.91)
Fibonacci ($n = 35$)	12.78	8.66 (1.47)	4.36 (2.93)	2.26 (5.65)
Fibonacci ($n = 36$)	20.57	13.75 (1.50)	6.92 (2.97)	3.34 (6.17)
4 by 4 Puzzle	63.67	41.46 (1.53)	27.02 (2.36)	12.42 (5.13)

Table 3: Performance on Two Parallel Lisp Benchmarks (Time in Seconds)

6.2 Performance on Parallel Code

Having established that our compiler produces quality sequential integer code, we now assess its performance on parallel code, in particular its ability to speed up execution of programs by harnessing parallelism.

Table 3 shows the performance of BaLinda Lisp on two parallel Lisp benchmarks. The first is the naive calculation of the Fibonacci number from the definition. For 8 processors, we were able to get a speedup of 6. The next program solves a 4 by 4 tile puzzle given in [8]. The programs were implemented without using tuplespace operations. Rather, parallel tasks are spawned and, when necessary, results are waited for.

No. of Processors	**BaLinda Lisp** $N = 11$ **(Speedup)**	**BaLinda Lisp** $N = 12$ **(Speedup)**	**Mul-T** $N = 11$ **(Speedup)**	**Transputer** $N = 11$ **(Speedup)**
1	1.45	7.90	33.2	19.75
2	0.98 (1.5)	5.40 (1.5)	16.6 (2.0)	10.64 (1.85)
4	0.461 (3.2)	2.82 (2.8)	8.5 (3.9)	5.52 (3.6)
8	0.289 (5.0)	1.33 (5.9)	4.3 (7.9)	2.88 (6.9)

Table 4: Performance on N_queen problem (Time in Seconds)

Table 4 shows the performance of the N_queen problem with $N = 11$. This version of the program finds all solutions. To obtain large granularity, we controlled the number of subtasks generated within the program itself. This is done by controlling the decision to do an `EXEC` or not. As a result, for $N = 11$, the number of tasks generated is 6, 12 and 18 respectively for the 2, 4 and 8 processor configuration. We compared this to the Mul-T implementation of the same problem on an Encore Multimax multiprocessor.

In the Mul-T version, the number of tasks created was 121. However, the Mul-T version did achieve better speedups. By increasing the problem size to $N = 12$, we were able to improve the speedup somewhat.

6.3 Performance Bottlenecks

The ultimate aim of any work in the area of parallel Lisp is to produce efficient implementations that can compete with conventional imperative (parallel) programming languages. We have shown that our implementation can, for integer sequential code, rival that of even C. Our parallel implementation is also competitive when compared to other parallel Lisp implementations - bearing in mind that in BaLinda Lisp, Linda-based synchronization, which is more general than future is supported. But how does it fare against something like parallelized C code? What are the performance bottlenecks in implementing BaLinda Lisp? In an attempt to answer this question, we translated a well-known shared memory parallel processing benchmark mp3d, written in C, into BaLinda Lisp. This is a program found in the SPLASH shared memory parallel benchmark suite [11] that uses the Stanford particle method to examine rarefied flows over objects in a simulated wind tunnel. While the reader may object to the use of a numerical benchmark for testing Lisp, we found that we were able to locate important performance bottlenecks using this benchmark.

The translation was tedious but relatively straightforward. The main problem was the use of record structures and pointers in the C version. We used Lisp arrays to emulate records, though this did make debugging difficult. The sequential BaLinda Lisp and C versions gave exactly the same runtime statistical outputs thereby verifying the correctness of the translation. However, the same cannot be said about the parallel versions. For reasons unknown to us, the runtime statistics differ slightly between runs. The parallelized C version uses Unix's shared memory and the mutual exclusion and conditional variable synchronization mechanisms supported by SunOS 5.3.

	BaLinda Lisp	**Parallel C version**	**Slowdown Factor**
Sequential	4.022	0.393	10.23
8 processors using tuples	101.218	0.905	111.8
8 processors using mutex and conditional variables	8.781	0.905	9.70

Table 5: Performance on `mp3d` (Time in Seconds)

Table 5 shows the performance of the BaLinda Lisp version versus the C version of `mp3d` using the input "test.geom" which comes with the SPLASH benchmark suite. The timing is for 40 time steps of 3000 molecules. We first note that the parallelized C version of the program is some 2.3 times slower than the sequential version. A quick check revealed that some processors can wait up to 5 times the computation time in the function `move` (which takes up 93% of the time in sequential execution) for a barrier synchronization. We attribute this to the small problem size and the multiuser nature of our platform. The parallel BaLinda Lisp version using tuplespace operations is some 25 times slower than the sequential version and more than 100 times slower than the parallel C version. We therefore looked for possible reasons to account for this.

While it is difficult to investigate the actual program to account for the loss in performance, we used a small set of kernel loops to confirm our suspicions about where the possible bottlenecks are.

The first suspect for causing the loss in performance is tag checking. The dynamic typing nature of Lisp makes it necessary to do runtime type checking by inspection of tags attached to data. Since most machines nowadays do not support hardware tag checking, we have to do this time consuming operation in software. The first row of Table 6 shows the time taken to

	BaLinda Lisp	C	Slowdown Factor
Adding 1000 single precision floating point numbers (including loop overheads)	5.77	0.26	22.19
Summation of a 100 x 100 array of integers	3.66	0.46	7.96
Entering and leaving an empty critical region 1000 times using tuplespace operations	61.87	0.55	112.5
Entering and leaving an empty critical region 1000 times using mutex and cond. var.	0.697	0.55	1.27
8 Processor Barrier Synchronization using tuplespace operations	41.53	0.67	61.99
8 Processor Barrier Synchronization using mutex and cond. var.	0.606	0.67	0.90

Table 6: Measuring the Performance Bottlenecks (Time in Milliseconds)

add 1000 single precision floating point numbers. The BaLinda Lisp version which requires tag checking is more than 22 times slower than the C equivalent.

The next bottleneck we identified is in array access. `mp3d`, for example, uses three dimensional arrays. In C, the address can be computed readily and with a single memory access, the desired element can be obtained. Lisp only support one dimensional arrays (also called *vectors*). To get higher dimension arrays, one must use vectors within vectors. In this scheme, to access an element in the d dimension, d memory accesses must be done. The second row of Table 6 reflects the cost of using two dimensional array in BaLinda Lisp by summing up a 100 by 100 array of integers. This is 7.9 times slower than the equivalent C loop. An important contributing factor is probably the cache.

Last but not least, we measured the cost of synchronization. In SunOS 5.3, a critical region can be implemented using the simple `mutex_lock` and `mutex_unlock` functions. In BaLinda Lisp, we use a pair of tuple `IN` and `OUT` to achieve the same effect. The cost of these two mechanisms are measured and reported in the third row of Table 6. It takes 0.5 μsec to do a pair of `mutex_lock` and `mutex_unlock` while it takes 112 times more or 61.8 μsec to enter a critical region using a pair of tuple `IN` and `OUT`. However, the reader should bear in mind that while for simple locking, the tuple operations are not as fast, they are meant to offer a more general synchronization mechanism that includes atomic exchange of arbitrary information. When we also used the same mutual exclusion and conditional variable mechanism in BaLinda Lisp, we achieve nearly the same performance as the C version, as shown in the 4th row of Table 6. This points out that for "pure" task synchronization and mutual exclusion, efficient system functions should be provided and tuple operations are not appropriate.

The cost of tuple operations affecting the efficiency of barrier synchronization using tuplespace operations is shown in the last row of Table 6. Using a $O(\log p)$ barrier algorithm, BaLinda Lisp is still some 61.9 times slower than using simple locks and conditional variables. The timing reported is the lowest observed. In general, the timing of the barrier is extremely sensitive to workload. This is probably the most serious source of performance degradation in mp3d. When we replaced all locks and barriers with the same mutual exclusion and conditional variable mechanisms used in the C version, we were able to achieve significant improvements in speed as shown in the fifth row of Table 5 where the BaLinda Lisp version is less than 10 times slower than the parallel C version. This is in line with the last row of Table 6 which shows that by using mutual exclusion locks and conditional variables, we can achieve the same performance (barring the difficulty in obtaining an accurate timing) in barrier synchronization. In this case, the loss in performance for the parallel BaLinda Lisp version is due solely to the problems associated with the sequential BaLinda Lisp version of mp3d.

7 Future Work

We believe that not only have we identified the major performance bottlenecks, we were also able to quantify their impact on performance. It is therefore a challenge to remove these bottlenecks in order to enhance Lisp's status as a parallel programming language.

On the first problem of tag checking, works in the area of type inference on dynamic typed languages [5], of which Lisp is one, will prove useful. We hope to incorporate such optimizations in our future versions of the compiler.

The second problem concerns the use of regular data structures such as arrays and records in Lisp. In its original form, Lisp's data structure of list and atoms is meant to be general. However, for performance as well as programmability, we see no choice except to extend Lisp to explicitly support data structures such as arrays and records. Works in this area is already underway in the Lisp community.

Another performance issue in sequential Lisp is that of garbage collection. Currently, our compiler uses a very simple mark and sweep algorithm. With increasing memory size, the frequency of garbage collection decreases. Still, we would like to investigate this matter further in the future.

On parallel processing, although the use of tuplespace offers a clean conceptual model for parallel programming, the implementation challenges are many. However, this is to be expected as high level constructs are often difficult to implement efficiently. We shall be looking into revamping the implementation model of our tuplespace and optimization of tuplespace operations, which is still a subject that requires much research. Also neglected in this study was speculative processing and its optimization, which certainly is another major area of work.

8 Conclusion

In this paper we have presented a compiler for the BaLinda Lisp parallel Lisp dialect. BaLinda Lisp supports the generative communication model of Linda which provides the user with a high level parallel programming model that is easy to use.

For sequential codes not involving floating point arithmetic, we were able to achieve performances compatible with that of optimizing C compilers. For parallel code, we demonstrated that relatively good speedups can be achieved. By using a shared memory numerical benchmark, we studied the problems that remains in trying to achieve performance compatible to traditional imperative languages. Although it may be argued that the example used falls outside the problem domain that Lisp is intended for, we were able to identify and quantify performance bottlenecks which, if removed, will no doubt be beneficial across the board. Much work remains to be done but it does seems to us that the idea of Lisp, sequential or parallel, running as competitive alternative to parallel C, say, may not be an impossible dream after all.

References

[1] J. F. Bartlett, 'SCHEME->C : a portable Scheme-to-C Compiler', *DEC Western Research Lab. Research Report 89/1*, Jan 1989.

[2] N. Carriero and D. Gelernter, 'Linda in context', *Comm. ACM*, 32, pp. 444-58, 1989.

[3] M. D. Feng, 'Compilation and Run-time Environment of Parallel Lisp on Distributed Systems', *National University of Singapore PhD. Thesis.* 1994.

[4] R. H. Halstead, 'Multilisp : A Language for Concurrent Symbolic Computation', *ACM Trans. on Prog. Lang. and Sys.*, vol. 7, no. 4, pp. 501-538. Oct 1985.

[5] F. Henglein, 'Global Tagging Optimization by Type Inference', *1992 ACM Symp. on Lisp and Func. Prog.*, pp. 205-215. 1992.

[6] Inmos Limited. *Transputer Instruction Set : A Compiler Writer's Guide.* Prentice-Hall 1988.

[7] T. Ito and R. H. Halstead, Jr. (ed) *Parallel Lisp : Languages and Systems.* Lecture Notes in Computer Science 441. Springer-Verlag 1990.

[8] R. E. Korf, 'Depth-first iterative-deepening : an optimal admissible tree search', *Artificial Intelligence*, vol. 27, pp. 97-109. 1985.

[9] D. A. Kranz, 'ORBIT : An optimizing compiler for Scheme', *Yale University Technical Report YALEU/DCS/RR-632.* Feb 1988.

[10] D. A. Kranz, R. H. Halstead and E. Mohr, 'Mul-T : A High Performance Parallel Lisp', *ACM SIGPLAN '89 Conf. on Prog. Lang. Design and Implementation.* pp. 81-90. Jun 1989.

[11] J. P. Singh, W.-D. Weber and A. Gupta, 'SPLASH: Stanford Parallel Applications for Shared-Memory', *Computer Architecture News*, vol. 20, no. 1, pp. 5-44. 1992.

[12] G. L. Steele, 'RABBIT : A compiler for Scheme', *MIT AI Lab. TR 474.* May 1978.

[13] D. Teodosiu, 'HARE : An optimizing portable compiler for Scheme', *ACM SIGPLAN Notices*, vol. 26, no. 1, pp. 109-120. 1991.

[14] C. K. Yuen, M. D. Feng, W. F. Wong and J. J. Yee, *Parallel Lisp Systems: A Study of Languages and Architectures*, Chapman and Hall, 1993.