Case Studies in Parallel Programming

D. ABRAMSON
Division of Information Technology
CSIRO
c/o Department of Communication and Electrical Engineering
Royal Melbourne Institute of Technology
P.O. Box 2476V
Melbourne 3001, Australia

INTRODUCTION

This paper contains case studies of two computationally expensive programs, and discusses the techniques used for producing parallel versions of the code. The first program uses simulated annealing for constructing school timetables. Simulated annealing is a combinatorial optimisation technique based on statistical mechanics. It is used for finding minimal cost solutions to problems which are too large for exhaustive search techniques. It models a set of cooling atoms which are governed by inter-particle forces. The atoms are cooled slowly in order to produce a configuration with a very low system energy. The problem with slow cooling is that it requires a large amount of processor time. An extremely elegant technique for improving the execution speed is to use parallel processors. It is possible to divide the atoms into partitions and then cool them separately. Synchronisation is required only when atoms interact with others in different partitions. This type of parallelisation can show good speedup, and can reduce many hours of processor time to minutes of elapsed time. The paper will present some results gathered on an Encore MULTIMAX shared memory machine.

The second application is a printed circuit router program. Computing the paths of the tracks in printed circuit boards is an extremely expensive process. It involves finding a path for each wire which does not cross any previously created tracks. The paths are found using a maze router algorithm, which advances a wavefront into unfilled cells. A parallel decomposition of the problem consists of routing more than one wire at a time. Because of the relationships between tracks, it is initially difficult to see a good parallel algorithm. However, wires tend to be clumped together thus exhibiting a large degree of locality. Our parallel algorithm involves using a divide and conquer technique for partitioning the circuit board. A recursive program divides the board into two partitions, and passes the relevant wires to each partition. Wires which are not totally contained in the two sub partitions are held until the sub partitions have been routed. The recursion continues until either a fixed maximum depth is reached, or there are no wires left. Routing is then performed on the way back up the recursive tree. The scheme cannot show the same linear speedup found in the simulated annealing code, but does benefit from a small number of processors. Again, a number of test runs on an Encore MULTIMAX are presented.

The paper contrasts two different partitioning strategies, and illustrates their strengths and weaknesses. The programming paradyme that has been used is the *shared memory* model, in which many processes share access to structures in one global memory pool. The paper comments on the use of message passing and dataflow machines as alternatives for the shared memory model.More details on both of these applications can be found in other papers, namely Abramson (1988) and Abramson and Freidin (1989) .

TIMETABLE PROBLEM and SIMULATED ANNEALING

The problem of creating a valid timetable involves scheduling classes, teachers and rooms in such a way that no *teacher, class or room* is used more than once per period. For example, if a class must meet twice a week, then it must be placed in two different periods to avoid a clash. The timetable is to be distributed across a fixed number of periods per week. A class consists of a number of students. We will assume that students have already been grouped into classes. In each period a class is taught a *subject*. It is possible for a subject to appear more than once in a period. A particular combination of a teacher, a subject, a room and a class is called an *element*. An element may be required more than once per week. The combination of an *element* and a frequency is called a *requirement*. Thus, the timetabling problem can be phrased as scheduling a number of requirements such that a requirement, teacher, class or room does not appear more than once per period.

It is possible to define an *objective* or *cost function* for evaluating a given timetable. This function calculates the number of clashes in any given timetable. An acceptable timetable has a cost of 0. The cost of any period is of the sum of three components: a class cost, a teacher cost and a room cost.

The class cost is the number of times each of the classes in the period appears in that period, less one if it is greater than zero. Thus, if a class appears no times or once in a period then the cost of that class is zero. If it appears many times the the class cost for that class is the number of times less one. The class cost for a period is the sum of all class costs. The same computation applies for teachers and rooms. The cost of the total timetable is the sum of the period costs.

SIMULATED ANNEALING

Simulated annealing (SA) is a Monte-Carlo technique which can be used to find solutions to optimisation problems. A good review of the theory and practice can be found in Van Laarhoven and Aarts (1987) . The technique simulates the cooling of a collection of hot vibrating atoms. When the atoms are at a high temperature they are free to move around, and tend to move with random displacements. However, as the mass cools the inter-particle bonds force the atoms together. When the mass is cool, no movement is possible, and the configuration is frozen. If the mass is cooled quickly then the final system energy may not be minimal. However, if it is cooled slowly, then the final energy may be the lowest possible. At any given temperature a new configuration of atoms is accepted if the system energy is lowered. However, if the energy is higher, then the configuration is accepted only if the probability of such an increase is lower than that expected at the given temperature. This probability is given by $P(\Delta E) = e^{-\Delta E/KT}$, where K is Boltzmann's constant.

By modelling optimisation problems as a set of randomly vibrating atoms, it is possible to find optimal, or sub-optimal, solutions. Many optimisation problems can be considered as a number of *objects* which need to be scheduled such that an objective function is minimised. The vibrating atoms are replaced by the objects, and the value of the objective function replaces the system energy. An initial schedule is created by randomly scheduling the objects, and an initial cost (c_0) and temperature (T_0) are computed. Subsequent permutations are created by randomly choosing two objects, interchanging them, and computing a change in cost (Δc). If $\Delta c \leq 0$ then the change is accepted. However, if $\Delta c > 0$ then the probability of that change is calculated,

$$P(\Delta c) = e^{-\Delta c/T}.$$

If the probability is greater than a randomly selected value in the range (0,1) then the change is accepted. After a number of successful permutations the temperature is decreased by a cooling rate, R, such that $T_n = T_{n-1} * R$.

One of the advantages of simulated annealing over algorithms which always seek a better solution (hill climbing algorithms) is that simulated annealing is less likely to get caught in local minima, because the cost can increase as well as decrease.

APPLYING SA TO THE TIMETABLING PROBLEM

The application of simulated annealing to the timetabling problem is relatively straight forward. The atoms are replaced by elements. The system energy is replaced by the timetable cost. An initial allocation is made in which elements are placed in a randomly chosen period. The initial cost and an initial temperature are computed. The cost is used to reflect the quality of the timetable, just as the system energy reflects the quality of a substance being annealed. The temperature is used to control the probability of an increase in cost and relates to the temperature of a physical substance. At each iteration a period is chosen at random, called the **from** period, and an element randomly selected from that period. Another period is chosen at random, called the **to** period. The change in cost is calculated from two components:

1) The cost of removing the element from the **from** period
2) The cost of inserting the element in the **to** period.

The change in cost is the difference of these two components. The element is moved if the change in cost is accepted, either because it lowers the system cost, or the increase is allowed at the current temperature. Unlike the classic simulated annealing technique which would actually swap two elements, an element is removed from one period and placed into another. This allows the number of elements in one period to increase or decrease, and for all periods to a contain different numbers of elements. If two elements were swapped then it would not be possible to change the number of elements per period.

The cost of removing an element consists of a class cost, a teacher cost and a room cost. Likewise, the cost of inserting an element consists of a class cost, a teacher cost and a room cost. If after removing an element from a period the number of occurrences of that class is > 0, then the class cost saving is 1. Similarly, if there are one or more occurrences of the teacher after that teacher has been removed then the teacher saving is 1. This technique also applies for rooms. The cost of inserting an element can be calculated using the same basic technique. In this way it is possible to determine the change in cost incrementally without recalculating the cost of the entire timetable. This attribute is particularly useful when the parallel version of the algorithm is implemented.

A PARALLEL ALGORITHM

Simulated annealing, while very effective at solving the timetabling problem, can be extremely slow (for example, a data set for a real school took 14 hours of processor time on a SUN 3/60). The elapsed time taken to run the simulated annealing algorithm described can be improved by using a parallel algorithm rather than a serial one. In the serial algorithm, each permutation of the elements is performed sequentially, and the new configuration either accepted or rejected. A new configuration is not generated until the previous one is performed. However, it is possible to perform multiple permutations concurrently, providing each permutation is independent of the other permutations.

A parallel algorithm can be implemented by assigning multiple *processes* to the task of permuting the timetable. The timetable must be held in a *shared* memory area accessible to all processes. Each process independently chooses an element to move (from a **from** period), and a **to** period. In order to prevent other processes from choosing the same element and **to** period, they must *lock* the element. It is not actually desirable to lock the entire **to** period, as this would severely limit the number of concurrent swaps which were possible. Instead, they only need lock the *teacher, class* and *room* in the **to** period. Similarly, the *teacher, class* and *room* must be locked in the **from** period. These items must be locked so that no other process can effect the cost computation of a given process. The incremental cost computation technique allows a process to calculate the change in cost without recomputing the cost of the entire timetable. Once these items have been locked a process can determine the change in cost

independently from all other potential swaps. If a process chooses an element, teacher, class or room which is already locked, then it must abandon the choice and try another.

The maximum number of concurrent processes depends on the size of the timetable. If there are too many processes for a given number of elements, then the number of processes abandoned swaps will be too high. Every time a choice is abandoned the effective speedup is decreased.

The *locks* described above can be implemented by simple read-modify-write variables in shared memory. A process can read a lock, and write a special marker value into the lock with an indivisible cycle. If the process reads the special lock value then it knows that the lock is already current and can abandon the choice. Such read-modify-write variables are not uncommon for multiprocessor machines. True semaphores are not required because the process does not wish to suspend when a lock is already claimed. Deadlock is not possible because a process *backs-off* any transaction which it cannot complete.

Implementing the parallel algorithm on a shared memory multiprocessor is relatively simple. The timetable must be held in shared memory, together with the lock variables. Once the timetable has been initialised the master process can *fork* and spawn as many child processes as necessary. Each child process permutes the timetable until the system is frozen, or the timetable has been solved. Each process can maintain its own temperature, or access a shared temperature variable. Similarly, each child process may share a common random number generator or maintain its own. If they use separate random number generators then each must use a different initial seed to avoid the same pseudo random sequences.

EXPERIMENTAL RESULTS FOR PARALLEL ALGORITHM

The parallel algorithm described in the previous section has been implemented on a conventional shared memory multiprocessor, a 10 processor Encore MultiMax. Some test data was presented to the parallel program, and the effective speedup was measured. The results are shown in Table 1. The execution time is shown for the purely serial code on the Encore, and then the parallel code using 1, 2, 4, 6 and 8 processors. It was not possible to use all 10 processors because of external demands on the machine. The Peak speedup is defined as the time for the single process run divided by the smallest multiprocess time. The peak speedup for the small problems is low because there are not sufficient resources to keep the processors busy. In general, the larger the problem, the greater the speedup. It can be seen from these examples that whilst not ideal, the speedup in many cases is significant.

When evaluating a parallel algorithm, it is important not just to consider speedup, but also absolute speed. The efficiency of the parallel algorithm can be expressed as the time taken to solve the problem using the parallel code with one processor, divided by the time taken using a serial version of the program with one processor. A number of experiments have indicated that the parallel code varies from equal, to at worst two times slower than the serial version. Thus, in this worst case two processors are required before the parallel code overcomes the cost of the locking and synchronisation code. The serial time is shown in Table 1. In most of these

TABLE 1 - Results of Parallel Execution

Test Data	Number Elements	Serial Time	Time in Secs for Number of Processors 1	2	4	6	8	Speedup
1	100	43	41	20	16	13	13	3.2
2	150	79	97	52	29	27	22	4.3
3	200	139	180	87	54	38	33	5.4
4	250	211	255	142	85	71	72	4.4
5	300	390	729	409	218	157	137	5.3
6	400	402	529	288	159	103	78	6.7
7	600	807	842	528	256	165	150	5.6
8	600	774	906	473	265	194	135	6.7
9	2252	5700	6900	3840	2100	1440	1020	6.8

examples the parallel version was only slightly slower than the serial code. Large timetable data sets could easily be expected to use up to 32 MIMD processors, providing a significant speedup. Further, it is possible to omit much of the locking code, which would reduce the cost of the parallel solution substantially by lowering the overheads. This technique is currently being investigated.

ROUTING USING LEE'S ALGORITHM

Routing a printed circuit board involves finding paths for the wires that connect integrated circuits. The wires are not allowed to cross on any given layer, however, more than one layer may be used to connect all wires. There are many different routing algorithms, and a good summary can be found in IEEE (1983). The oldest and most general technique uses a wavefront, which is advanced from the source to the destination. First described by Lee in Lee (1961), this scheme finds the shortest path between any two points, and is based on a fixed grid of cells. Cells either contains a value indicating that the board position is either occupied by a hole, circuit track, or part of an advancing wave. The cell corresponding to the source location is initialised with a low score value, e.g. 1, and is then added to a current wave front list. Then all of the cells surrounding the current wave front are set to the score 2, and are added to the wave front list. The initial cell is subsequently removed from the wave front list. This process continues until the wave either meets the target destination, or entirely fills the board. If the destination is not touched, then there is no path from the source to the destination. If the wave meets the target, then a path is traced by tracing a path of decending cell value from the destination until the source is reached. This basic technique is enhanced to cater for practical considerations in the routing of printed circuit boards and integrated circuits.

A PARALLEL ROUTER

The basic parallel algorithm attempts to detect wires that are *unrelated* and route them concurrently. For example, wires that are at different ends of a board are likely to be completely independent, and thus could be routed at the same time. These wires are grouped by recursively dividing the board into regions, and passing the wires completely contained in a region to the router separately. Wires which are contained in more than one region are held until all those at lower levels have either been successfull routed, or could not be routed because of congestion.

At each stage, the router calls itself, and splits the region it has been passed into two sections. It created three wire lists; two for those wires completely contained in each region, and one for those wires which cannot be partitioned. This process continues until either there are no wires left to route, or some preset maximum depth has been reached. After the routine has called itself recursively, it proceeds to route the wires it had held onto, and also attempts to route those which could not be connected by the recursive call. Thus, each call to the router accepts a wire list, and returns a wire list containing the wires it could not connect.

This process continues until it returns to the root of the call tree. If all of the wires have been connected then the router terminated, however, if there are wires left then it starts the process over again with two new layers. It is worth noting that this recursive procedure effectively sorts and routes the wires by length, because the shortest wires are passed down to the lower call levels, and are thus routed first.

In itself, this recursive algorithm is sequential. However, by replacing the recursive call with a recursive *parallel* procedure call, each section can be routed concurrently. There is no need for any communication between the processes after the procedure call because they are all operating in separate areas of the board. If the algorithm is being executed on a shared memory multiprocessor, the board should be placed in shared memory to avoid copying it between calls. The reference parameters used in the call must also be placed in shared memory so that the results can be returned to the calling processs. If the machine is a distributed message passing machine, then the wire lists and board contents must be transmitted to the processor executing the called code.

EXPERIMENTAL RESULTS

The parallel router was exercised on a circuit board which measured 5 inches by 8 inches and contained 559 wires between 52 integrated circuits. Whilst this is quite a small circuit, it was large enough to demonstrate the effectiveness of the router without using enormous amounts of processor time. The program was run using 1, 2, 4 and 8 processors. During each run two traces were maintained. The first logged the number of active processes against time. The second logged the number of wires routed against time, and shows the progress of the program. These traces are plotted for each of the test runs and are shown in Figures 1 and 2, although they are not all included in this paper due to lack of space. The activity traces show how much concurrency is being extracted at any point in time. They all exhibit the same basic form; they start by using the maximum number of processors, but diminsh as the run proceeds. The reason for this is that the divide and conquer algorithm proceeds to the bottom of the recursion tree where it routes as many wires conncurrently as possible. Consequently, it consumes as many processes as possible. However, the long wires are held until the shorter ones have been routed. The progress traces also all have the same form. The shorter wires are routed first and are connected fairly quickly. The more processors available, the quicker these wires are connected. The longer wires take more time to connect for a few reasons. First, the basic maze routing algorithm execution time is proportional to the square of the wire length. Second, the board is already congested by the shorter wires, and thus the longer wires have to route around them. Third, there are fewer processors which can be applied to the longer wires because they span more than one board region. Because of these reasons the curves all flatten out, and the time to route the entire board is dictated by the time to route the long wires.

The results show that increasing the number of processors from 1 to 8 has only decreased the execution time from 180 minutes to 100 minutes. Because of these problems, an additional level of concurrency was added to the program. In the modified scheme, the wave front for each wire is advanced by more than one process, thus speeding the routing of individual wires. This technique is not nearly as efficient as wiring separate wires concurrently because using more than one process to advance the wave front requires substantial interprocess communication and synchonisation, whereas the divide and conquer technique does not require any interprocess communication until all wires at a level are routed. However, the parallel wave front approach has the ability to speed the routing of the long wires, which are responsible for controlling the time to connect the entire board. Using this scheme, the execution time can be further reduced to 50 minutes. Whilst not optimal this can constitute a significant speedup. Work is continuing on increasing the effciency of the algorithms used in the router.

CONCLUSION

Whilst it is too early to report, this scheme seems to require no more cycles to achieve the same cost solution, and is much faster because the locking overhead is not present. With this approach, the parallel code could easily be as fast as the serial version.
This paper has demonstrated how concurrency can be applied to two different computationally expensive programs. Both use different programming techniques to divide the problem. The simulated annealing code is best solved using a number of different worker processes, each of which manipulate and relax a central shared structure. Workers are created once when the program starts. They resolve their resource contention using simple memory locks, without the need for complex semaphore or monitor software. Deadlock is prevented by using a backoff technique. The simulated annealing code exhibits quite good speedup, especially for large problems. One inefficiency in the algorithm is that it requires shared resources to be locked so that two processes do not try and move the same tuple. This locking can make a single process program as much as two times slower than the serial code, although most of the examples shown in this paper are within 30% of the serial speed. Some work has been done on the effect of removing the locks altogether, and allowing the algorithm to proceed with inaccurate values for the costs.

nches by 8 inches and
a small circuit, it was
g enormous amounts of
uring each run two
against time. The
progress of the
n in Figures 1 and 2,
activity traces show
exhibit the same basic
sh as the run
oceeds to the bottom
sible. Consequently, it
eld until the shorter
The shorter wires are
ble, the quicker these
w reasons. First, the
of the wire length.
longer wires have to
ed to the longer wires
the curves all flatten
he long wires.

s only decreased the
ms, an additional
the wave front for
g of individual wires.
ntly because using
process
nique does not require
ver, the parallel
which are
scheme, the
is can constitute a
algorithms used in

to achieve the same
nt. With this

ent computationally
the problem. The
processes, each of
once when the
y locks, without the
y using a backoff
cially for large
es to be locked so
ke a single process
the examples
n done on the effect
n inaccurate values

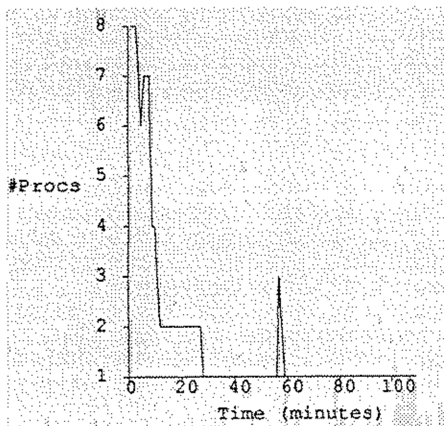
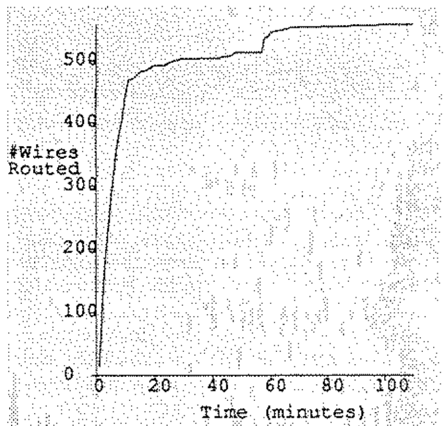


FIGURE 1 Activity Trace 8 CPU



Progress Trace 8 CPUs

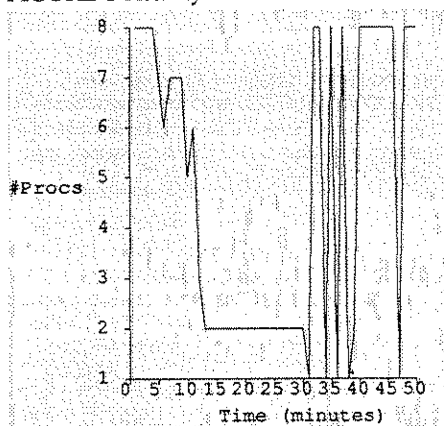
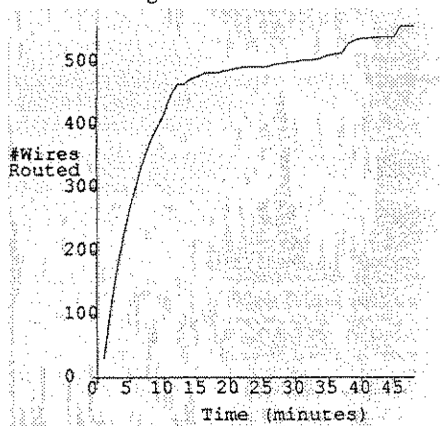


FIGURE 2 Activity Trace 8 CPUs Parallel Wave



Progress Trace 8 CPUs Parallel Wave

The router code uses a divide and conquer algorithm rather than a fixed number of workers. Each time the board is divided a new process is created, and each time a procedure call is made a process is destroyed. This fork-join approach has been largely discouraged because the process creation and destructions costs are usually quite large. However, in this problem, the cost of routing a number of wires in an area is so high, that the process creation and destruction costs are not relevant. The algorithm performs quite well on problems with a very high degree of wire locality, because there are very few wires kept at higher levels of the call tree. However, as the number of non-local wires becomes significant, the algorithm performs very badly. This performance degradation is not because of the implementation, but because the wires held are longer than those passed on, and thus take much longer to route (The complexity increases roughly as the square of the distance between the wires). To make matters worse, they are being routed by fewer processes; thus the algorithm slows down from two factors. To improve the performance of this program we have added a second level of concurrency, which takes over when there are not enough recursive workers to occupy the multiprocessor. In the second scheme the wave front is advanced by a number of worker processes, in much the same manner as they are used in the simulated annealing code.

Both of these programs have been implemented on shared memory machine. It is almost impossible to map the simulated annealing efficiently onto a message passing machine because, whilst the timetable itelf can be split across many machines, information about the numbers of teachers, classes and rooms in each period must be available to all processors. Further, this

information is quite volatile, and could not be copied around efficiently. The printed circuit router could be mapped onto a message passing machine, but the board structure would need to be transmitted between processors each time a split and join occured. For large circuits, this could involve quite a large amount of data. Another disadvantage is that the processor which runs the process at the root of the recursive tree requires sufficient memory to hold the entire board.

Key sections of these programs have also been written in a functional language called SISAL (McGraw (1985)) for execution on a prototype dataflow machine as described in Abramson and Egan (1988). Dataflow machines allow easy extraction of concurrency, and SISAL allows easy expression of concurrency without program *side-effects*. The simulated annealing code has also been written in a low level dataflow language (DL1) and run on the prototype machine. The results of the simulated annealing code written in DL1 were similar to the shared memory results. Many problems have been encountered in expressing the problem in the SISAL, mostly because the *single assignment* property of the language makes it very difficult to perform parallel updates on shared structures without excessive data movement.The low level dataflow language did not present any major obstacles other than its lack of expressive power as a general purpose programming language. The concurrency was extracted easily, indicating that the dataflow multiprocessor was an appropriate execution environment for this class of problem.

The router was coded in SISAL without difficulty, and allowed easy expression of the concurrency. However, because of the nature of the problem, the SISAL implementation required large amount of data to be copied and rebuilt. Research is continuing on the best way to specify such problems with minimal data movement, without restricting the concurrency.

ACKNOWLEDGEMENTS

The Parallel Systems Architecture Project is a joint project between the Commonwealth Scientific and Industrial Scientific Organisation (CSIRO) and the Royal Melbourne Institute of Technology (RMIT).The programs used for producing the experimental results were written by Tony Starr, Junko Freidin, Ha Nguyen and Mark Gazzola. The members of the Parallel Systems Architecture Project, in particular Dr. G. Egan, assisted with various parts of this work. Thanks must also go the the RMIT Department of Computer Science for providing access to their Encore multiprocessor.

REFERENCES

1. Abramson, Using Simulated Annealing to Solve School Timetables: Serial and Parallel Algorithms, RMIT Technical Report TR-112--069R, 1988, submitted for publication.

2. Abramson and Freidin, A Parallel Router for Printed Circuit Boards, RMIT Technical Report, 1989, in preparation.

3. Abramson and Egan, An Overview of the RMIT/CSIRO Parallel Systems Architecture Project, *The Australian Computer Journal*, Vol 20, No 3, pp 113 - 121, 1989.

4. IEEE,*Transactions on Computer-Aided Design of Integrated Circuits and Systems*, October 1983, Volume CAD-2, Number 4, 1983, ISSN 0278-0070.

5. Lee, An Algorithm for Path Connection and its Applications, *IRE Transactions on Electronic Computers*, Vol EC-10, No 3, pp 346 - 352, September 1961.

6. McGraw, SISAL: Streams and Iteration in a Single Assignment Language, Language Reference Manual, Lawrence Livermore National Laboratories Technical Report, M-146, March 1985.

7. Van Laarhoven and Aarts, *Simulated Annealing: Theory and Applications*, D. Reidel Publishing Company, Kluwer Academic Publishers Group, 1987.