Statement-Sets
(Parallelism Enabled by Independence)

Svend Erik Knudsen

Informatikdienste, ETH Zentrum, CH-8092 Zürich, Switzerland
e-mail: knudsen@sd.id.ethz.ch

Abstract

The possibility to specify a collection of statements (without any preferred execution order) is clamed to be important for the programming of parallel computers.

Keywords: programming model, parallel programming, independence, impossible program, Oberon, statement-set.

1 Introduction

In recent years, computer architects responded to the demand for higher performing computer systems designs based on better technology and more parallelism. Multiprocessors (symmetric multiprocessors (SMP), e.g.) have been build, sold, and successfully used for more than 25 years, now. Nowadays, even a desktop workstation might be a SMP. Such systems mainly execute individual programs by using one processor at any given time. The wall clock time needed for the execution of a program depends therefore more on a system's effective processor performance than anything else. The main benefit of SMP-systems has been in the increased throughput. Glass-house managers or system administrators might estimate this, but not necessarily the users.

It is still difficult to apply a substantial amount of parallelism to the execution of a single program. Data parallel problems are the main targets for current automatic parallelizing compilers[1, 2, 3]. Independence among probably irregular program parts is fundamental to enable their simultaneous execution. It is, however, generally impossible to automatically decide the independence among arbitrarily chosen program parts. This paper proposes to signal a collection of independent program parts by a syntactic notion. Such notions might help a system to execute such marked parts simultaneously.

Section 2 classifies the tasks we want to be faster (or cheaper) executed. Section 3 suggests independence as the attribute that allows simultaneous execution of program parts. Section 4 presents the concept of statement-sets as a syntactic notion to specify mutually independence among program parts. Sections 5 and 6 propose two new kinds of statements for Oberon to mark statement-sets and illustrate their usage by examples. Section 7 lists some performance figures of the prototype implementation. Section 8 surveys the implementation. The following section summarises our initial experience, and section 10 concludes the paper.

2 Transformational versus Reactive Programming

To concentrate on the essential point, we consider only transformations (i.e. transformational or functional program parts). Typically, high performance computer systems perform some given (and well understood) transformations most of the time. Such functions are most likely programmed in a sequential and imperative language like FORTRAN or C.

In the following considerations, we ignore programs with explicitly programmed parallelism. Such programs are often reactive and might including message passing or other forms of explicit process communication or synchronisation. We consider transformational and reactive programming as complementary programming models best used to fulfil conceptually orthogonal requirements.

Thus, we ignore the current legacy of thinking of reactive programming as being the natural programming model for programming transformations for a "parallel system". Furthermore, we do intentionally not search for a single programming model that covers the programming of both transformational and reactive problems.

3 On Sequential versus Arbitrary Execution

Current processors' architecture makes use of parallelism at many levels (e.g. gate level, pipelining, multiple functional units, processor level, instruction level, and operating system level). Parallel execution of a single program is only possible, if the program can be split into parts, and if some of these parts are mutually independent. An example might illustrate this theory [4]:

```
    ma := MaxArray(a);
    mb := MaxArray(b);
    m := Max(ma, mb)
```

Function MaxArray yields the value of the greatest element in an array, and function Max gives the greatest value of its two scalar arguments. The two first statements are independent and can, therefore, be executed in any order, even simultaneously. However, the third statement depends on values resulting from the execution of the first two statements. The execution of the last statement may not start until the two previous statements have been executed. It is implicitly assumed that the function MaxArray does not cause side-effects.

ALGOL-like programming languages [5] use the semicolon for the syntactic separation of statements. A semicolon does, however, also imply that the two separated statements have to be executed sequentially (one after another, in the written order). In the example above, this semantic of a semicolon is superfluous for the first semicolon, but not for the second one.

The following two statements summarise the pragmatic learning:

(1) Mutually independent program parts may be executed in arbitrary order.

(2) Mutually dependent program parts must be executed in sequential order.

Program parts are said to be mutually independent, if variables modified by one part are not accessed by any of the other parts.

For the remainder of this article, *independence* stands for *mutual independence.* Similarly, *independent* stands for *mutually independent*.

It seems impossible for a compiler to decide upon independence among arbitrarily selected parts of an arbitrary program. In some (often data-parallel and numeric) cases independence is certainly detectable. In other more complex and probably irregular but still typical cases, a compiler has only little or no chance to detect independence among program parts. (See Appendix.)

4 The Statement-Set Concept

The specification of a strictly sequential execution order is often an over specification, which might disable a compiler to apply simultaneous execution to independent program parts. For transformational problems, the explicit programming of parallelism (e.g. with threads or message passing) is also a kind of over specification, which typically causes an unnecessary slow execution on single-processor systems. Furthermore, we consider explicit "parallel programming" an unnecessary and error-prone programming style for the programming of transformational (i.e. functional) problems, conceptually even worse than the use of GOTO.

We call the notation to express *arbitrary* execution order of a collection of statements permitted a *statement-set*. Statement-sets can, in contrast to explicit "parallel programming", be considered pure directives indicating mutually independence among program parts.

5 Statement-Sets in Oberon

We made a prototype implementation of statement-sets for the programming language Oberon [6]. Oberon was chosen due to its few and clean constructs, and as there is much Oberon-knowledge at the ETH. The proposed extensions are, however, only *intended to illustrate our theses* about the importance of being able to specify independence among program parts. They should *not* be considered an agreed upon part of Oberon. Furthermore, the statement-set concept is *equally applicable and important* for programming languages like FORTRAN or C.

It is important for understanding of the following text to know the syntax of a *statement sequence* :

```
StatementSequence = Statement { ";" Statement } .
Statement = [ Designator ":=" Expr | .. ] .
```

A *statement sequence* is an ordered sequence of *statement*s separated by semicolons.

In the prototype implementation, a statement-set expresses independence among statement sequences. Statement sequences are said to be independent, if a variable modified by one statement sequence is not referenced (used in expressions or assigned to) by any other of the statement sequences of the statement-set. The statement sequences of a statement-set may therefore be executed in any order, even simultaneously. The prototype implementation for Oberon supports two variants of statement-sets: the *set statement* and the *all statement*.

```
SetStatement = "{" StatementSequence {"," StatementSequence}
        "}" .
```

The statement sequences separated by "," must be independent.

```
AllStatement = ALL ident ":=" Expr TO Expr [BY ConstExpr] DO
        StatementSequence END .
```

Every execution (instance, "iteration") of the statement sequence in an all statement must be independent. The control variable *ident* must be of an integer type (SHORTINT, INTEGER, or LONGINT), and its value may not be changed during the execution of the given statement sequence. The value of the control variable is not defined after the execution of the all statement.

The execution of the statement sequences of a statement-set (all statement and set statement) must not be terminated by a *return statement* or an *exit statement*. Otherwise, statement-sets can freely be used in programs; they may be nested, used in recursive procedures, e.g.

The following transformations on statement-sets are possible and reasonable for a target system with only one processor. A set statement is converted to a statement sequence by removing the brackets and replacing the commas by semicolons. An all statement is converted to a *for statement* by substituting ALL by FOR.

6 Some Examples

The examples illustrate the use of statement-sets. These declarations are used in all examples:

```
CONST n = 1000;

VAR matrix: ARRAY n,n OF REAL;
    a, b: ARRAY n OF REAL;
    m, ma, mb: REAL;
    i: LONGINT; j: INTEGER;
```

The example from section 3 reprogrammed with a statement-set.

```
{ ma := MaxArray(a), mb := MaxArray(b) };    (* set statement *)
m := Max(ma, mb)
```

This piece of a program searches the maximum value among all elements in *matrix*.

```
ALL i := 0 TO n-1 DO a[i] := MaxArray(matrix[i]) END;
m := MaxArray(a)
```

Procedure *QuickSort* sorts the elements in array *a* [5].

```
    PROCEDURE QuickSort(VAR a: ARRAY OF REAL);

      PROCEDURE Sort(VAR a: ARRAY OF REAL; l, r: LONGINT);
        CONST stretch = 16;
        VAR i, j, ii, jj: LONGINT; h, key: REAL;

      BEGIN
        ii := l; jj := r; key := a[(ii + jj) DIV 2];
        REPEAT
          WHILE a[ii] < key DO INC(ii) END;
          WHILE key < a[jj] DO DEC(jj) END;
          IF ii <= jj THEN
            h := a[ii]; a[ii] := a[jj]; a[jj] := h; INC(ii); DEC(jj)
          END
        UNTIL ii > jj;
        i := ii; j := jj;

        IF (l<j) & (i<r) & (r-l>=stretch) THEN (* independent *)
          {Sort(a,l,j), Sort(a,i,r)}           (* set statement *)
        ELSE                                   (* sort sequentially
*)
          IF l < j THEN Sort(a,l,j) END;
          IF i < r THEN Sort(a,i,r) END;
        END
      END Sort;

    BEGIN
      IF LEN(a) > 1 THEN Sort(a, 0, LEN(a)-1) END
    END QuickSort
```

7 Performance Figures

7.1 Run-Time System

The execution time figures of the current implementation are quite typical for the level of performance achievable for an implementation of statement-sets. Better performing implementations can however be imagined. The executions of the following pieces of programs illustrate the performance level of the implementation:

```
    PROCEDURE ForN(n: LONGINT);
      VAR i: LONGINT
    BEGIN FOR i := 1 TO n DO END END ForN;

1)  FOR i := 1 TO 1000000 DO END
2)  ForN(1000000)

3)  FOR i := 1 TO 1000000 DO ALL j := 0 TO 0 DO END END
4)  ALL i := 1 TO 1000000 DO END

5)  FOR i := 1 TO 1000000 DO { } END
6)  FOR i := 1 TO 100000 DO { , , , , , , , , , } END

7)  ALL i := 1 TO 100000 DO ForN(10) END
8)  ALL i := 1 TO 10 DO ForN(100000) END
```

Execution times in seconds measured on a Sun SPARC 10 402 (with 2 CPUs running with 40 MHz clock and no secondary cache):

program	sequential	1 CPU	2 CPUs
1	.080	.081	.081
2	.061	.061	.061
3	.203	2.64	5.31
4	.081	.768	1.56
5	.080	2.43	4.74
6	.008	1.87	3.08
7	.090	1.76	1.83
8	.060	.060	.030

The column *sequential* lists the execution times for the eight program pieces when the compiler generates sequential code.

The columns *1 CPU* and *2 CPUs* list the execution times when the compiler generates code for simultaneous execution and the code is executed by 1 or 2 processors respectively.

The performance figures above indicate that statement-sets can be applied for the programming of relatively small program fragments without causing excessive performance losses.

7.2 Use of statement-sets in routines

The following two problems are not necessarily typical for parallelization of applications. They are simply initial test cases for the modified compiler and its run-time system. Nevertheless, they demonstrate the feasibility of statement-sets, even for the programming of recursive routines.

7.2.1 Complex Fast Fourier Transformation

The in place in order complex Fast Fourier Transformation algorithm (cFFT) described [7] has been programmed in the "enhanced" Oberon. The columns *1 CPU* and *2 CPUs* list the measured *speedup* factors of the generated code for simultaneous execution executed by one respectively two CPUs. The column *n* indicates the number of points for which the cFFT was executed.

n	sequential	1 CPU	2 CPUs
256	1.	1.000	1.323
512	1.	1.001	1.608
1024	1.	.949	1.584
2048	1.	.932	1.707
4096	1.	.951	1.633
8192	1.	.969	1.795
16384	1.	.975	1.729
32768	1.	1.006	1.873
65536	1.	1.011	1.808
131072	1.	.992	1.845
262144	1.	.994	1.772

7.3.2 Quicksort

1'000'000 randomly chosen 32-bit integers were sorted by Quicksort [8]. The used routine is an "improved" version of the Quicksort shown in section 6. In particular, the constant *stretch* is used in the same way to control the granularity (and thereby the number) of independent program parts.

stretch	sequential	1 CPU	2 CPUs
100	1.	.971	1.72
1000	1.	.991	1.82
10000	1.	.997	1.85

7.2.3 Remarks

In both examples above, the same source code has been used for the generation of the sequential and the parallel code. The code variant is selected by a compiler option. Similarly, the same binary was used for the *1 CPU* and *2 CPU* cases. Here the run-time system (see next section) was configured by commands. In the case of the cFFT measurements, the same (self-scaling) codes were used for all values of *n*.

The non-linear speedups of the 1 CPU- to the 2 CPU-figures have mainly algorithmic and technical reasons:

Firstly, the initial activation of Sort in the body of Quicksort is executed sequentially. This single activation of Sort for splitting the 1'000'000 elements into two smaller (and independent) sorting-problems causes more than 5% of the whole work to be sequential. This single factor limits the *2 CPU* speedup to a factor less than 1.9. Similarly, the self-scaling Fast Fourier Transformation has also clean sequential program parts.

Secondly, the simultaneous execution of program parts with complex (and heavily changing) access-patterns by several CPUs typically increases the number of cache misses compared to the single CPU-case.

In both cases, the performance figures were obtained by executing the routines on a Sun SPARC 20 502 (with 2 CPUs running with 50 MHz clock and no secondary cache).

8 Prototype Implementation

We have implemented the two proposed statement-sets in the Oberon-2 compiler for Sun SPARC installed with the Solaris 2.3 operating system, a UNIX variant [9, 10]. The implementation was made easier by splitting the task into the following two parts: Firstly, the modifications in the compiler deal mainly with the scanning, parsing, and generation of sequential code. Secondly, a new run-time system module schedules the execution of independent procedures. Accordingly, the survey contains two parts: a description of the extensions to the compiler proper and a description of the supporting run-time system. Both the compiler and the run-time system are programmed in Oberon.

8.1 Modification in the Compiler

The extension of the compiler with the set statement and the all statement has been almost straight forward. This is mainly because no new complex constructs (like new data types) had to be introduced.

The statement sequence of the all statement is translated into a procedure with one value parameter with the same name and of the same type as the control variable itself. For each execution of the all statement, a routine of the supporting run-time system is called with five arguments: *SEK.All(staticLink, routine, low, high, step)*. The meaning and the use of the five arguments should be self-explanatory. *SEK.All* does not return until *routine* has been executed with all desired values of the control variable as argument.

Similarly to the all statement, all statement sequences of a set statement are translated into procedures. The addresses (and entry points) of these routines are entered into a table of procedures. Each set statement causes a routine of the run-time system to be activated: *SEK.Set(staticLink, procTab)*. *SEK.Set* terminates when all procedures referenced by *procTab* have been called.

The statement sequences in an all statement and a set statement might be of any type defined in Oberon with two restrictions: It is nor permitted to leave these statements by an exit or return statement. These restrictions are checked by the compiler.

In the current version of the compiler the compilation of statement-sets to the code mentioned above is enabled by a command line flag. The compiler generates sequential code, if the flag is not set. Section 5 describes simple transformations to do this.

8.2 Run-Time System

The execution of statement-sets is supported by a separately compiled Oberon module, currently called *SEK*. *SEK* schedules submitted routines on the available processors. Furthermore, the run-time system provides a command line interface to configure the system and to display collected statistics.

The code generator of the compiler submits execution-order independent procedures to the run-time system. The rule is that all routines submitted by any single call of *SEK.All* or *SEK.Set* are executed before the submitting routine terminates. The submitted procedures may be executed in arbitrary order (also simultaneously) because they are independent. The submitted routines might even be programmed with statement sets, i.e. they might call *SEK.All* or *SEK.Set*.

The run-time system maintains a queue of submitted procedures including the needed arguments for their execution (actual value of loop control variable in the all statement, e.g.). A submitted procedure is called a *job*. Such a job is described by a *job descriptor*. A job descriptor contains among other fields a reference to a *barrier lock*, an atomicly decrementable counter, which is decremented when the corresponding job has been executed. *SEK.All* and *SEK.Set* first calculate the number of jobs to execute under their control and initialise a locally declared barrier lock accordingly. Hereafter, all jobs are either submitted to the queue of jobs if possible or executed directly. After the submission of the jobs, the submitting routine (*SEK.All* or *SEK.Set*) is waiting in a loop until all submitted jobs has been executed. Jobs are activated in such waiting loops.

In the initialisation phase of the run-time system, a light weight process (Solaris: *LWP*) is by default created as *worker* for each processor minus one in the system. These workers try repeatedly to retrieve a job from the queue of submitted jobs and, if successful, to activate it.

The following program fragment gives an impression of the algorithms used in the run-time system to support the execution of statement-sets. The first running implementation consisted of about 150 lines of Oberon code.

```
...

TYPE
    BarrierLock = RECORD free: BOOLEAN; n: LONGINT END;
    Proc1 = PROCEDURE (LONGINT);

PROCEDURE ExecuteOne;
BEGIN
    (* Retrieve "job" from "queue" if possible.
       If successful: Execute "job";
       Decrement referenced barrier lock *)
END ExecuteOne;

PROCEDURE Set*(staticLink: LONGINT; VAR procTab: ARRAY OF PROC);
    VAR i: INTEGER; lock: BarrierLock;
BEGIN
    lock.free := TRUE; lock.n := LEN(procTab);
    FOR i := 0 TO LEN(procTab)-1 DO
      (* Enter "job" represented by procTab[i], staticLink and
         reference to lock into "queue", if possible.
         Otherwise Exec procTab[i] with staticLink;
         Decrement lock.n atomicly *)
    END;
    WHILE lock.n > 0 DO ExecuteOne END;
END Set;

PROCEDURE All*(staticLink: LONGINT; pi: Proc1;
               low, high, step: LONGINT);
BEGIN (* ... *) END All;


PROCEDURE Worker;
BEGIN
    LOOP ExecuteOne END
END Worker;
...
```

The current run-time system is about three times bigger than the initial version (< 500 lines). This is mainly due to some added features: Gathering and display of statistics, commands to set the number of light weight processes, and some code to suspend and resume light weight processes during otherwise idle periods.

The prototype still lacks come features for general usage: The integration with the Oberon system has not been completely made. The consequences for the feasibility test were however small. A more solid and better integrated implementation is essential for education or professional use.

9 Initial Experiences

9.1 Experiences concerning the modification of the compiler

In Section 8, we mentioned that the modification of the compiler were quite simple. This is mainly because the new statement types (i.e. all statement and set statement) can be transformed easily into constructs that are available in most imperative programming languages: declarations and activations of procedures.

9.2 Experiences concerning the run-time system

The separation of duties between the compiler, that mainly handles syntactic issues, and the supporting run-time system, that implements parallel execution and synchronisation, is recommendable. This has made the testing of the run-time system and its adaptation to other scheduling strategies easy. In particular, no modification of the compiler was necessary for any of our modifications of the run-time system.

9.3 Experiences concerning the use of statement-sets

Our experience so far certainly verify that statement-sets are useful.

The application of statement-sets requires "only" checking of the explicit usage of variables in the statement-set itself. Procedures called from a statement-set must either be free of side-effects or at least be mutually independent of operations in other statement sequences of the statement-set. The latter might be more difficult and sometimes even impossible to check. However, my experience is that these restrictions are felt quite logically and that they typically are easy to verify. Programming pure sequential programs do also require deeper and similar analysis of the usage of variables and of the side-effects of called routines. Freedom of side-effects of called routines is often implicitly assumed when writing sequential programs.

The locality of reasoning makes the application of statement-sets easy: The applicability of a subroutine does not depend whether if it has been implemented by use of statement-sets or not. Mainly the execution time is affected by this. The application of parallelism to the execution of a subroutine is therefore relatively simple.

The low overhead for the execution of a statement-set does help programming. Relative small program pieces, causing few hundreds of instructions to be executed, can easily be part of a statement-set.

The writing of numeric applications indicates a lack of generality of the for statement: In Oberon, the *step* must be a (compile time evaluated) constant. The programming of the Fast Fourier Transformation mentioned in Section 7 would have benefited from it by having been a general expression evaluated whenever the for statement is executed.

It seems almost necessary to be able to declare local variables for the statement sequences in statement-sets, e.g. for loop-control. ALGOL 60 and some of its successors allow such declarations. Some experiments have been made in that area. For clarity reason, these have not been mentioned here.

The command line flag causing the generation of sequential code for a statement-set helps the debugging and the measuring of overheads caused by the generation of non-sequential code.

Programs including the programming language extensions discussed above can interact with an operating system just like other typical sequential programs. The main caveat is that system calls in more than one statement sequence of a statement-set must (also) be mutually independent (i.e. at least MP-save). Even, reactive programs should not cause (conceptual) problems due to the added language constructs.

10 Conclusions

In recent years, it seemed necessary to let single applications be executed by several or many processors to either lower the time or the cost of its execution. Current compilers limit the automatic or simple application of parallelism to essentially data-parallel cases.

This paper suggests mutual independence of program parts to be an important attribute. Only independent program parts may be executed simultaneously. Statement-sets are suggested as programming language construct to express independence among program parts. The semicolon of ALGOL-like programming languages still have the duty to separate mutually dependent program parts: The result of the execution of mutually dependent program parts depends on the order of their execution.

The often perceived statement that parallelism is cumbersome to deal with, is hardly relevant for statement-sets. In this case, *programmers have to reason about dependence and independence among statements, program parts, operations, etc. - which they always have to do- and not about parallelism or asynchrony!*

We have shown that statement-sets are easy to implement in Oberon by a modification of an existing Oberon compiler. Actual programming experience indicate that the use of statement-sets is feasible, both if performance of compiled programs and ease of reasoning is considered.

Aspects like portability of programs, adaptability to different computer architecture's, easiness of debugging and conversion to sequential programs, and quality of generated code might be better than with tools currently used for "parallel" programming.

Statement-sets are worthwhile for more investigation. More theoretical and experimental work has to be done.

Acknowledgements

This work (and this paper) would not have been done without the almost endless discussions with Alfred Schai. My Ph. D. supervisor Niklaus Wirth did both directly and indirectly keep me on the firm ground of clarity (simplicity). Josef Templ supported me by giving me many hints for the initial compiler modification and scheduler realisation. Later on, he even improved the modified compiler and its run-time support. The hint to and the FORTRAN code for the used Fast Fourier Transformation is due to Wesley Petersen. Herman Seiler did also spend many hours with discussions and did show the feasibility of statement-sets by implement statement-sets in his Modula-2 compiler. The two referees provided also important hints for the improvement of this paper.

Appendix: Another Impossible Program

Two program parts are said to be independent if any variable modified by one of the program parts is not referenced (i.e. read or modified) by the other program part.

It can be proven that it is impossible to construct a program that decides, given an arbitrary program P with two parts as input, whether the parts are independent or not. The proof is similar to Strachey's method that shows the impossibility of constructing a program that decides, given an arbitrary program P as input, whether the program P will terminate execution or will run forever.

Thus, suppose it were possible to construct a Boolean function *Independent(p, p1, p2)*, where p identifies an arbitrary program with parts p1 and p2. *Independent* would return the value true, if *p1* and *p2* are independent, and false otherwise. To show that *Independent* cannot be constructed according to the specification, take module *Prog* as the special program *p* and its procedure *Part* both as part *p1* and part *p2*.

```
MODULE Prog;

  PROCEDURE Independent(p, p1, p2: Program): BOOLEAN;
    ...
    ... And side-effect-free.
  END Independent;


  VAR i: INTEGER;

  PROCEDURE Part;
  BEGIN
    IF Independent(Prog, Part, Part) THEN i := -i END
  END Part;

BEGIN
  i := 1
END Prog.
```

Now consider the value of *Independent(Prog, Part, Part).*

Assuming this to be true leads to a contradiction, since according to the assumption this indicates that the procedure *Part* is independent of itself, which in this case it isn't.

On the other hand, assuming *Independent(Prog, Part, Part)* to be false also leads to a contradiction, since this indicates that *Part* is dependent on itself, which in this case it isn't.

Thus, either of the possible values of *Independent* lead to a contradiction. Consequently the function *Independent* cannot be constructed.

This proof is a variant of Strachey's impossible program-proof [11].

References

1 Chapmann, et al., VIENNA FORTRAN Compilation System Version 1.0 User's Guide, Technical Raport, Institut for Software Technology and Parallel Systems, Univ. of Vienna, (Jan. 1993).
2 High Performance Fortran Forum, High Performance Fortran Language Specification, Version 1.0, Scientific Programmingf, 2(1 & 2 1993).
3 W. F. Tichy and C. G. Herter, Modula-2*: An Extension of Modula-2 for Highly Parallel, Portable Programs, Technical Report 4/90, Department of Computer Science, University of Karlsruhe, (Jan. 1990).
4 J.-P. Banâtre, D. Le Métayer. Programming by Multiset Transformation, Commununication of the ACM 36 (1993) 1, pp. 98-111.
5 P. Naur, Revised Report on the Algorithmic Language ALGOL 60, Commununication of the ACM 6 (1963) 1.
6 N. Wirth, The Programming Language Oberon, Department Informatik, ETH Zurich, Report 111 (Sep. 1989), pp. 11-28.
7 C. Temperton, Self-Sorting In-Place Fast Fourier Transformations, SIAM J. Sci. Stat. Comput. 12 (1991) 7, pp. 808-823.
8 C.A.R. Hoare, Quicksort, Computer Journal 5 (1962) 1, pp. 10-15.
9 J. Templ, SPARC-Oberon User's Guide and Implementation, Department Informatik, ETH Zurich, Report 133 (June 1990).
10 SPARC International, Inc. The SPAR Architecture Manual, Version 8, Prentice Hall, Englewood Cliffs, 1992.
11 Strachey, C. An impossible program, Computer Journal 7, (1965) 4, p. 313.