Proceedings of the 14th IASTED International Conference on Applied Informatics (Innsbruck, Feb. 1996), IASTED-ACTA Press, pp. 190–192.

Towards a Parallelising COBOL Compiler

Rizos Sakellariou and Michael O'Boyle
Department of Computer Science, University of Manchester
Manchester M13 9PL, United Kingdom

Abstract

This paper briefly describes some of the fundamental issues in developing an automatic parallelising COBOL compiler. The rationale for such a tool and its connection with research in scientific computing is described. This is followed by an account of the useful forms of parallelism to be found in a COBOL program and how they may be detected using dependence analysis. Finally, transformations to uncover parallelism and how this can be mapped onto a parallel architecture are outlined.

Key words: Automatic parallelisation, parallel COBOL.

1 Introduction

Parallel computing has received considerable interest as a means of obtaining increased computer performance. In order to overcome the difficulties associated with programming parallel computers, parallelising compilers have been promoted as a means of emancipating the programmer from this task [1, 2]. In its full concept, a parallelising compiler accepts as an input a sequential program and transforms it to a parallel one tailored to the requirements of a specific parallel architecture. This process, known as automatic parallelisation [3], can be regarded as a mainly two stage process: in the first stage the compiler identifies modules which can be run in parallel and applies program restructuring transformations to improve program's performance. The result of this phase is a semantically equivalent parallel program, which, in the second stage, is mapped onto a parallel architecture.

Most of the current research in automatic parallelisation has been targeted around the parallelism encountered in FORTRAN-like loop structures, i.e. parallelism which results from the repeated execution of the same instructions on different data; this is known as the *data parallel* paradigm [4]. Our motivation for this paper stems from the fact that COBOL programs perform a similar process; in many cases a set of identical program actions are applied to a large number of data records. Given the high number of COBOL program lines currently in use and the slow transition of existing code to more potentially efficient programming paradigms, such as the object-oriented [5], automatic parallelisation of existing programs may provide an alternative source of increasing performance. Considerable research has been directed into parallelising SQL statements [6]; these results may be used for the parallelisation of COBOL programs making access to database files, however, not all programs include this option. Some preliminary work in identifying those characteristics amenable to parallelism has been described in [7].

In this paper we attempt to transplant experience drawn from the development of a parallelising compiler for FORTRAN programs [8] into COBOL and our objective is to highlight the feasibility of a parallelising COBOL compiler. Issues concerning the parallelism encountered in COBOL programs and its detection and mapping onto a parallel architecture are discussed.

2 Parallelism Detection

Two types of parallelism are usually found in computer programs written in conventional sequential languages: *task parallelism* results from the concurrent execution of different parts of a program, while *data parallelism* results from the concurrent execution of the same program statements on different elements of data.

COBOL applications are characterised by the need of handling large amounts of data; thus, although task parallelism may be present, we confine our discussion to the detection of data parallelism. We distinguish between three different types: *file parallelism* exists when running the same program using different files; *record parallelism* refers to the parallelism encountered when processing different records of the same file, for instance as a result of reading a master file to print a name list; and *array parallelism* is associated with the parallelism existing on computations involving arrays. A distinction is also made for the parallelisation of certains constructs like `SORT` or `SEARCH` which can be parallelised by means of appropriately written external library routines; for instance, a parallel sorting algorithm is described in [9]. From a compiler point of view, interest is focused on the first two forms of parallelism.

A necessary requirement for the parallelised program is to preserve the semantics of the original sequential program; thus, two statements can be executed in parallel only if there is no relationship between them which places constraints on their execution order. Such constraints are identified by means of a *data dependence analysis* [3]. In our case, it is important to identify dependences between different loop iterations (loop-carried dependences). Assuming than an iteration i_1 precedes an iteration i_2 then three types of data dependence may exist: a *data flow dependence* denotes that a variable which is computed (written) in iteration i_1 is read in iteration i_2; a *data anti-dependence* denotes that a variable which is read in iteration i_1 is computed in iteration i_2; and a *data output dependence* denotes a variable which is computed in both i_1 and i_2.

To illustrate the above, consider the code fragment shown below, which implements the process of updating an indexed master file from a sequential transaction file:

```
    ...
    MOVE "N" TO END-FILE.
    PERFORM P500-UPDATE UNTIL END-FILE="Y".
    ...
P500-UPDATE.
    READ TRANS AT END MOVE "Y" TO END-FILE.
    IF END-FILE NOT = "Y"
       (...read master according to trans-rec
           update master-rec fields...)
       WRITE MASTER-REC.
```

The repeated executions of the P500 paragraph define a loop; the objective is to examine whether any record parallelism may be exploited by assigning different iterations on different processors. Clearly, the scalars representing the fields of the master record define a data output dependence, since they are computed in any two iterations of the loop, while the scalars representing the fields of the transaction record define a data anti-dependence; applying a technique known as *privatisation* [3], where each processor has its own copy of the variables, both output and anti dependences can be removed. Thus, parallelism may be constrained by data-flow dependences which, in our example, may arise when a record of the master file which has been written in an earlier iteration is read again by a subsequent iteration; this possibility may be eliminated if there are not any two records of the transaction file referring to the same record of the master file. If this is not the case, then updates to the same record of the master file must be executed by the same processor or a synchronisation mechanism must safeguard the order of the execution. Sorting the transaction file before updating the master file may also provide a solution whenever there are multiple records in the transaction file referring to the same record of the master file; then, the transaction file may be split into subfiles, with no two of them referring to the same record of the master file, and exploit file parallelism.

In general, by viewing COBOL files as arrays (where the first record of a file is the first element of an array, and so on), the abstract representation framework and the techniques used in the context of scientific applications can be also applied here. The additional requirement posed is that in certain cases, as in the previous example to ensure that there are no data-flow dependences, some knowledge on the value of the records may also be needed; this may be a complicated issue in the case of loops performing operations on several master files. However, it appears that for programs involving sequential operations on files, their parallelisation is straightforward.

3 Optimising Transformations

It may not be always possible for the compiler to detect directly the parallelism present in programs. Assume, for instance, that the example code illustrated in the previous section was written as:

```
    ...
    READ TRANS AT END MOVE "Y" TO END-FILE.
    PERFORM P500-UPDATE UNTIL END-FILE="Y".
    ...
P500-UPDATE.
    (...read master according to trans-rec
        update master-rec fields...)
    WRITE MASTER-REC.
    READ TRANS AT END MOVE "Y" TO END-FILE.
```

Then, the scalars representing the fields of the transaction record show a data flow dependence since they are getting values (READ statement) at the iteration preceding the one where they are used. To uncover the parallelism, the compiler must be able to restructure the source code to the form shown earlier; similar transformations to eliminate dependences have been studied in the context of array-based scientific codes [3]. Restructuring may also involve a series of transformations to increase the functionality of the source code, such as the removal of GO TO statements [10].

It must be noted though that any restructuring transformations should be applied subject to increasing program's execution time. The latter, i.e. the execution time of the parallel program, t_p, can be modelled as $t_p = t_o + t_s/p$, where t_o is the time spent on overheads due to parallel execution, t_s is the time needed to execute the program sequentially, and p is the number of processors used in the parallel implementation; overheads may be primarily due to communication or synchronisation requirements, and/or load imbalance [8]. Usually, parallelism trades off against overheads; thus an attempt to uncover parallelism may increase some other sources of overhead. Using this model, the parallelising compiler must decide whether the parallelisation of a code may result in a faster execution time (i.e. $t_p < t_s$). For instance, assuming that a synchronisation mechanism is used to safeguard the order of execution in the example of the previous section, the time that the processors spent waiting to acquire a lock should be less than $t_s(1 - 1/p)$ in order to have faster execution time. Further issues on modelling these overheads are discussed in [11].

4 Mapping

In the mapping phase, the parallelism available is mapped onto processors in such a way that overheads are minimised. The approaches traditionally used for `DO..ENDDO` type of loops in scientific applications are either to assign a specific number of iterations to each processor (loop partitioning) or to distribute the data amongst processors (data partitioning) and then having each processor computing the values of the data it owns (owner-computes rule); the former approach is usually preferable on shared memory and the latter on distributed memory parallel computers. In the case of COBOL programs, where data reside on a disk, analogies can be drawn with the *shared-disk* and *shared-nothing* paradigms respectively, which are employed in parallel database technology [6, 9].

Using a shared-nothing model, parallelism may be exploited by splitting the data files into subfiles which are stored in a disk assigned to a particular processor (file parallelism). In order to reduce any communication overheads it is essential that all the records which are accessed at the same loop iteration are located in the same disk. In several cases this may not be possible and an algorithm to reduce any transfers should be devised; such an algorithm is expected to be a simplified version of algorithms used to minimise overheads for FORTRAN programs as the one outlined in [8]. Considering the code fragment shown in section 2, sorting and splitting the transaction file into subfiles of equal size and storing them in the same disk with subfiles of the master file which has been split (assuming it is sorted as the transaction file) at the points where the transaction file is split, may provide a communication-free parallel execution. However, the sorting and splitting phases pose an overhead which should be considered by the compiler before these phases are applied (cf section 3).

In a shared-disk model, loop iterations can be assigned in a round-robin fashion to processors, or they may be assigned to the first idle processor. While both these strategies guarantee a fair distribution of the workload amongst processors, they may suffer from extensive I/O contention. Thus, it is preferable to assign consecutive loop iterations to each processor; considering again the example presented in section 2, this approach ensures that there will be no 'interleaving' in the way that processors access the records of the transaction file. However, in order to partition the iterations as evenly as possible, it is necessary to know their total number; knowing the file size, f, and the record size, r, of the transaction file and assuming that p processors are available, each processor is allocated approximately f/rp iterations. Thus, the `PERFORM..UNTIL` construct may be treated as a `PERFORM..N TIMES`, where `N=`f/r; then, the first processor executes iterations 1 through n/p, in general processor i executes iterations $(i-1)n/p+1$ through in/p (assuming that p divides n).

From an abstract point of view, the process of mapping data located in a disk can be modelled by means of a memory hierarchy model where disk access is associated with the most expensive level in the hierarchy. A possible way of minimising these costs would be to move blocks which are more often accessed (e.g. the records of a master file) to a less expensive level; this strategy, also discussed in [7], needs careful consideration in the selection of the blocks to avoid continuous transfers back and forth.

5 Concluding Remarks

This paper forms the starting point of our research into building an auto-parallelising COBOL compiler. Only a brief outline of the issues involved in parallelising sequential COBOL has been made. It appears that there is a significant connection with well known techniques applied for the automatic parallelisation of scientific programs. Further work is necessary in describing COBOL actions in a representation amenable to analysis, transformation and parallelisation. The impact on data storage and interaction between applications must also be investigated.

References

[1] M. Wolfe, *High Performance Compilers for Parallel Computing* (Addison-Wesley, 1996).

[2] H. Zima, B. Chapman, *Supercompilers for Parallel and Vector Computers* (ACM Press, 1990).

[3] U. Banerjee, R. Eigenmann, A. Nicolau, D. Padua, "Automatic Program Parallelization", *Proceedings of the IEEE, 81* (2), 1993, 211–243.

[4] V. Adve, A. Carle, E. Granston, S. Hiranandani, K. Kennedy, C. Koelbel, U. Kremer, J. Mellor-Crummey, S. Warren, "Requirements for Data-Parallel Programming Environments", *IEEE Parallel & Distributed Technology, 2* (3), 1994, 48–58.

[5] L. J. Pinson, "Moving from COBOL to C and C++: OOP's biggest challenge", *Journal of Object-Oriented Programming, 7* (6), 1994, 54–56.

[6] D. DeWitt, J. Gray, "Parallel Database Systems: The Future of High Performance Database Systems", *Communications of ACM, 35* (6), 1992, 85–98.

[7] L. Richter, "Restructuring of Cobol-Applications for Coarse Grain Parallelization", *Proceedings of the workshop on Commercial Applications of Parallel Processing Systems (CAPPS) 93* (Austin, Texas: MCC Technical Corporation, 1993).

[8] F. Bodin, M. O'Boyle, "A Compiler Strategy for Shared Virtual Memories", in B. K. Szymanski, B. Sinharoy (eds.), *Languages, Compilers and Run-Time Systems for Scalable Computers* (Kluwer Academic Publishers, 1995), 57–69.

[9] A. Kumar, T. Lee, V. Tsotras, "A Load-Balanced Parallel Sorting Algorithm for Shared-Nothing Architectures", *Distributed and Parallel Databases, 3* (1), 1995, 37–68.

[10] J. C. Miller, B. M. Strauss III, "Implications of Automated Restructuring of COBOL", *ACM SIGPLAN Notices, 22* (6), 1987, 76–82.

[11] M. E. Crovella, T. J. LeBlanc, "Parallel Performance Prediction Using Lost Cycles Analysis", in *Proceedings of Supercomputing '94* (IEEE Computer Society Press, 1994), 600–609.