**PARALLELIZATION OF AN EXISTING HIGH-ENERGY PHYSICS
EVENT RECONSTRUCTION SOFTWARE PACKAGE**

R. Schiefer

Department of Computing Science, Technical University of Eindhoven, Netherlands
and

CERN, Geneva, Switzerland

E-mail: renes@harmony.cern.ch

D. Francis

Department of Physics, Boston University, Boston, Mass., USA

E-mail: djf@harmony.cern.ch

Abstract

Software parallelization allows an efficient use of available computing power to increase the performance of applications. In a case study we have investigated the parallelization of high-energy physics event reconstruction software in terms of costs (effort, computing resource requirements), benefits (performance increase), and the feasibility of a systematic parallelization approach. Guidelines facilitating a parallel implementation are proposed for future software development.

(Submitted to IEEE Trans. Nucl. Science)

1 INTRODUCTION

The increasing complexity of High-Energy Physics (HEP) experiments leads to a continuous increase in the performance requirements of the supporting software. The next generation of HEP experiments at the Large Hadron Collider [1] will require an increase in computing capacity of several orders of magnitude compared to that of today [2].

At the same time, industry is increasingly offering high-performance computing platforms based on commodity microprocessors with a low latency/high bandwidth interconnection network. These platforms can give cost-effective scalable computing power provided a high degree of parallelism can be introduced into the applications.

Event farming has proven to be an effective technique to boost the performance of event reconstruction applications [3]. It scales well in terms of throughput, as the ratio between communication and computation is low. Two issues are not addressed however: the event analysis time (latency) and computing resource utilization.

A parallel implementation with reduced execution time and resource requirements could allow the introduction of off-line event analysis techniques into the domain of real-time event analysis, e.g. cost effective on-line event reconstruction, interactive event analysis and closed-loop feedback control systems based on reconstruction data. Both average and maximum event latency are of concern here.

To achieve the above, finer grain parallelism compared to event farming has to be introduced into the application. Parallelization can reduce the task size and complexity on each processor, so resource requirements per processor (e.g. memory, an important cost factor) can be less and consequently a more cost-effective solution is obtained. The disadvantages compared to farming lie in the area of scalability, processor utilization (load balancing) and the difficulty of implementation.

Two fundamental forms of parallelism can be identified: *algorithmic* parallelism places the different tasks within the application on different processors; *data* parallelism involves the partitioning of data rather than code. In the latter case, the algorithm that is normally used to process the data is replicated over multiple processors, and each instance works on a local data partition. In the case of no dependencies between instances one speaks of farming, otherwise the term geometric parallelism is used [4].

Given the potential gains in parallelizing HEP software, we need to ask ourselves how easy it would be to take an existing HEP application and parallelize it. Also, how should new, sequential software be written now to allow a smooth transition to a parallel platform at a later date?

In the ESPRIT project GP-MIMD [5] a case study has been performed on the parallelization of the CPREAD off-line event reconstruction program of the CPLEAR experiment [6] at CERN. The following sections discuss first the potential parallelism in CPREAD, followed by an investigation into the implementation effort necessary. The investigation focuses on code decomposition and data organization. These issues are illustrated by the implementation of parallel track fitting within CPREAD.

2 THE CPREAD CASE STUDY

CPREAD is the off-line event reconstruction package of the CPLEAR experiment. It is used as a typical representation of off-line software:

- CPREAD is large—in the order of 200K lines of Fortran.
- It is developed and maintained with the code management tool Patchy [7].

- It uses standard HEP packages such as Zebra [8] for I/O and dynamic memory management, HBOOK [9] for histogramming purposes, and FFREAD [10] for application steering.
- It performs typical tasks such as decoding, pattern recognition, track and vertex fitting, and event display.

Figure 1 shows a schematic of the event reconstruction steps and the data flows within CPREAD. The data structures depicted in Fig. 1 are all implemented as globally accessible Zebra data structures, upon which the physics analysis code operates.

The event reconstruction steps are executed in a sequential order, based on data dependencies and computing optimization (i.e. aiming for minimal average event analysis time). The computing optimization is a function of the *rejection power* of each reconstruction step, defined as the rejection percentage per second of step analysis time. Computing optimization is achieved by executing reconstruction steps in decreasing order of rejection power where data dependencies permit.

Figure 1: Schematic of the event reconstruction steps in CPREAD.

Figure 2: Data dependencies among event data in CPREAD.

Two types of data may be distinguished, *event data* and *long-term data* (e.g. calibration constants). The latter are constant throughout a sequence of events (called a *run*). The event data only exist within a single event analysis.

Each reconstruction step may access both the long-term data and event data to produce new, additional event data, as depicted by the data flow and data dependencies

in Figs. 1 and 2. For instance, the data dependency between D3 and D2 indicates that the track fitting accesses D2 in order to produce D3. All data depicted in Fig. 2 are event data.

3 INVESTIGATION OF POTENTIAL PARALLELISM

The aim is to introduce parallelism at low cost. This means aiming for a parallel version of CPREAD resembling the sequential code as much as possible, minimizing the effort involved in code rewriting. Therefore the program structure (in terms of reconstruction steps and data structures) is to be left intact as much as possible. With this in mind, three (non-exclusive) ways of parallelization are considered:

- Reconstruction steps that have no data dependency between them are run in parallel for the same event. In CPREAD we can execute the analysis on charged tracks in parallel with the calorimeter analysis, as suggested by Fig. 2. This is algorithmic parallelism within an event, and will be referred to as *task parallelism*. Task parallelism reduces the event latency, thereby increasing event throughput. Also, additional rejection paths may be created and exploited if the step running in parallel has significant filtering capabilities, further improving latency and throughput. A negative effect is that running a step in parallel with the step normally succeeding it, implies that the latter now has to deal with more events due to less filtering, making the latency of the step an issue.
- Reconstruction steps that are data interdependent can work in parallel on different events using event pipelining. For example, when pattern recognition has processed an event, the track fitting starts to work on it, but at the same time the pattern recognition can accept a new event. This is *pipelining*, an example of algorithmic parallelism working on *different* events.

Pipeline parallelism does not reduce the latency of the event reconstruction, but does increase the event throughput compared to the sequential analysis.

- Parallelism is introduced within a reconstruction step by replicating an individual step and partitioning the data structure on which the analysis has to be performed, so that the overall structure can be processed in parallel. The simplest example is loop unrolling. In this article, *data parallelism* will solely refer to parallelism within an event, excluding the well-known parallelism applied at the event level, event farming.

There is a high degree of inherent data parallelism in HEP analysis code, given the repetitive nature of the work in each reconstruction step. Data parallelism reduces latency and increases throughput. For instance, for every track in an event a track fit is performed. An additional positive aspect of this type of parallelism applied to a specific reconstruction step is its local impact on the program structure.

Both algorithmic event parallelism and data parallelism can affect the rejection power of a reconstruction step, so a re-ordering of independent steps after parallelization may be beneficial. All three approaches reduce the resource requirements per processor.

In terms of required work, the same gain is easier to obtain with data parallelism than with algorithmic parallelism. The creation of different code modules requires work, proportional to the number of modules, whereas the amount of work involving the

implementation of data parallelism is independent of the number of module copies working together on the same data structure.

Within the processing of an event, data parallelism can only be identified within a reconstruction step. The strong interdependencies between the individual steps leave little scope for reducing the latency with algorithmic parallelism. However, the throughput can be addressed by pipelining the steps. The prime advantage here is the cost effective introduction of additional processors, keeping the overall memory requirement constant.

The choices concerning the amount and type of parallelism to introduce are based on the effort required and the benefits obtainable. The amount of effort is measurable by the amount of code changes, while the benefits are understood in terms of the improvements to latency (event analysis time) and throughput.

To quantify the cost and benefit parameters, a profiling tool was used to measure the amount of computing time spent in each step. Table 1 shows a breakdown of the CPU time for a representative CPLEAR event sample, as well as the *cumulative* rejection percentage at each step (columns 2 and 3). For example, it can be seen that 19% of the execution time was spent on pattern recognition, and in this step 34% of the events it processed were rejected. In total, the basic five reconstruction steps account for 77.4% of the execution time. The remaining time (22.6%) is mainly spent on i/o and not relevant for this study.

Column 4 shows the CPU breakdown for a sample of events that go through all steps; the percentages therefore indicate the relative contribution of each step to the maximum latency of the event processing.

Table 1
CPREAD Profiling on IBM 3270, 1200 events

Step	CPU (%)	Reject (%)	Latency (%)	Speed $O()$	Change (%)
Decoding	11	0	8.0	6	< 5
Pattern	19	34	13.9	< 10	> 25
Track fit	15	37	15.9	4	< 1
Vertex fit	27	70	30.3	4	< 10
Calorimeter	5.4	79	17.7	< 10	< 15
Total	77.4	—	85.8	—	—

Comparing the decoding and calorimeter percentages in columns 2 and 4, the following can be observed: an increase in throughput can be established by parallelizing the decoding as its contribution is twice the contribution of the calorimeter. However, a parallel calorimeter analysis is more effective for decreasing the maximum latency.

The numbers in column 5 are estimates of the speed-up that may be obtained by applying data parallelism and are based on the task being performed in terms of physics. For example, on average four track fits need to be performed per event; fitting the tracks in parallel should naively reduce the latency of this step by a factor of four. The amount of code change that is associated with the obtainable speed-up is a first-order estimate. Further precision requires a detailed study of the application structure.

Table 2 shows the calculated benefits for event processing latency and throughput for the different types of parallelization strategies (applied separately and a combination of data parallelism and pipelining). The projections are based on the figures from Table 1. The calculations do not take into account communication overheads and imperfect load

balancing with data parallelism, as these effects are difficult to estimate.

The figures for the sequential version show a maximum latency of 85.8% (the total of column 4 in Table 1) whereas the average latency is less because an average event does not complete all the analysis steps. The throughput of the sequential version is a direct function of this average latency.

Table 2

Calculated performance improvement. The numbers in parentheses are normalized to the sequential version

Version	Throughput (#evts per %)		Average Latency (%)	Max. Latency (%)
Sequential	1/56.8	(1.0)	56.8 (1.0)	85.8 (1.0)
Task par.	1/51.5	(1.1)	51.5 (0.91)	68.1 (0.79)
Pipelining	1/21.4	(2.7)	84.1 (1.48)	116.0 (1.35)
Data par.	1/10.7	(5.3)	10.7 (0.19)	16.0 (0.19)
Combined	1/4.9	(11.6)	19.6 (0.35)	26.0 (0.3)

The task parallelism strategy has a moderate improvement. The maximum latency drops by 17.7%, the time required for the calorimeter analysis (see column 4, Table 1), because this analysis is now done in parallel with the other steps. This has a minimal positive effect on both the average latency and throughput, especially when compared to the amount of extra processing power required. This example shows that task parallelism can improve both the latency and throughput, but the benefit is highly dependent on the application.

A pure pipelining strategy of the five steps improves throughput, but the differences in step latencies (pipeline imbalance) have a negative effect on the event latency, because consecutive events in the pipeline can block each other. This effect is partly compensated for by the dropping out of events (rejection). The figures in Table 2 are obtained via simulation.

The data parallelism strategy addresses the latency of the individual reconstruction steps and thereby the throughput. The figures in Table 2 are based on the speed-up figures from Table 1 applied to each step, and show a relatively high performance improvement.

The combined strategy of data parallelism and pipelining gives maximum throughput. In the case of an unbalanced pipeline, data parallelism can be applied to balance the pipeline i.e. by improving the latency of the pipeline bottleneck, the work load is spread more efficiently over the processors.

4 INVESTIGATION OF IMPLEMENTATION

In the previous section the scope for parallelism in CPREAD was investigated. In this section, we discuss the approach that should be taken to implement parallelism, and what obstacles may be encountered in doing so.

4.1 Development approach

Current HEP software development and maintenance makes use of code management tools such as Patchy [7] and its successor CMZ [11]. The code modifications necessary for a parallel version need to be implemented using the same tools and allowing the sequential and the parallel versions to co-exist. This corresponds to the situation in which different members of an HEP collaboration may require a sequential or parallel version,

depending on the hardware available to them. It also facilitates the necessary integration of ongoing code developments, thus ensuring that any parallel version stays as up to date as the sequential version. HEP code is continuously evolving!

Patchy and CMZ provide version management for a large library of Fortran routines. A program is constructed from this library by executing a user-written extraction macro which selects the appropriate routines and combines them into a program. Code management is facilitated by the decomposition of the code pool into a hierarchical structure. For this study, CMZ was adopted.

The Mona Lisa parallel programming paradigm was used as an additional structuring tool [12], as the development of a parallel application requires additional structuring to that provided by a sequential programming language. It is shaped as a language extension, offering a set of primitives which allow the exchange of data among modules that run in parallel and together make up the parallel program. Mona Lisa provides development support in several ways:

- In a single file, declaration statements indicate the start and end of the individual code modules that together make up the parallel program. A front-end translator automatically produces a Fortran file for each module, reducing user file management.
- Data exchange between modules occurs through user designated variables. The Mona Lisa front-end translator performs type checking on the communication of the values of these variables, trapping potential programming errors before ‘compile time’.
- The VIPER visualization tool [13], part of the Mona Lisa development environment, is used in the process of analysing, improving, and debugging a parallel program at the level of the interacting modules. It provides insight into the parallel computation through graphical visualization of tracing information produced by the parallel program during execution.
- Every Mona Lisa program has a built-in component, the program manager. The parallel program is executed by starting up the program manager. It then takes care of loading the modules on processors, starting the application, and unloading the modules once the application has finished. It also has the capability of run-time deadlock detection.

To summarize, CMZ is used for static and syntactic code management, whilst Mona Lisa provides dynamic and semantic development support. Automatic code generation (via extraction macros and the Mona Lisa front-end translator) and the existence of a ported sequential version ensure systematic code and result comparison.

4.2 Porting CPREAD

In general, the parallel platform will involve different hardware with different precision. The sequential version of the application should first be ported to this platform, thus decoupling the porting of the application from the parallelization stage. The existence of a sequential version on the target platform allows a better comparison of performance improvement and is needed to verify that the parallel version produces the same results as the sequential version.

The porting of CPREAD to the target platform was combined with a study of the existing code modularity. To this end the code was ported incrementally to the new platform. The first ported version only performed program initialization and termination.

Each successive version included an additional reconstruction step. This incremental porting avoided putting effort into porting code that would not be used in this case study, for example the code needed for event displaying.

The porting of the code involved not only the foreseen problems (different i/o handling, arithmetic precision, etc.) due to a new hardware platform, which in this case study was a Transputer network, but also some additional code changes due to the use of a different compiler.

4.3 Code decomposition

For any of the proposed parallelization approaches to be effective and efficient in terms of code size per processor, the code needs to be decomposable into the modules that correspond to the reconstruction steps. A coarse decomposition of CPREAD already existed, based on the existing hierarchy and extraction macros. However, a finer grain extraction procedure was required. The following two points prohibited a systematic change of the existing extraction macros to obtain the required level of extraction and necessitated new macros:

- The existing hierarchical code structure only roughly coincides with the decomposition into reconstruction steps. The routines that make up one reconstruction step are not grouped together, but exist at different places within the hierarchy.
- The existing macros, for reasons of simplicity, extract code regardless of whether it is used at run-time or not, in a coarse grain extraction process.

Based on the structure of the code at the highest level in the program, an attempt was made to extract, in single stages, the program initialization, termination, and reconstruction steps. These modules were all identifiable by an appropriate steering routine. By recursively selecting the routines that are called (a procedure which was largely automated), the set of routines constituting a reconstruction step could be constructed. However, this strategy of creating code modules failed due to the fact that at a lower level the identified modules had too many interdependencies, although there appeared to be a clean separation between individual reconstruction steps at a higher level.

The adopted procedure to create code modules was a combination of the procedure mentioned above and the elimination of calls to routines that were deemed not to be relevant to the code module being extracted. This required an understanding of the purpose of the individual routines.

4.4 Data organization

The data flows in the parallel version will principally concern event data, stored in Zebra structures: dynamic data structures based on banks (variable sized arrays), organized in tree configurations. Although an individual bank is contiguously stored in memory, the overall tree structure may be dispersed.

The standard mechanism of data communication on MIMD platforms is via the use of message passing. The message passing library calls communicate blocks of contiguous data, thus enabling the communication of individual banks. For the communication of large data structures, consisting of several banks, the communication overhead becomes excessive as multiple communications are required. Zebra provides a set of calls that implement the selective copying of (partial) Zebra structures to an array (and vice versa), thereby establishing a contiguous data block that can be communicated in one message

passing call. However, the additional overheads introduced by reformatting the data must be weighed against that of multiple communications.

5 PARALLEL TRACK FITTING

In the previous sections scope for parallelism in CPREAD has been discussed, followed by a discussion of implementation issues. In this section we apply this to an example of data parallelism: parallelization of the track fitting. Owing to its minimal impact on the algorithmic structure, this example allows us to focus on the development approach, code partitioning, and data flow implementation with minimal application-specific code rewriting.

The data sets representing the individual tracks are self-contained (mutually independent), and a fit is performed to each data set invariantly within a loop construct. Data parallelism is therefore introduced by simply unrolling this loop.

The first step towards a parallel version was to decide how many modules to implement and what tasks they would perform. Subsequently, the data flows between the modules were identified, and finally, using the task assignment and the data flows, the individual modules were constructed.

Two different modules were developed: one which performed the full CPLEAR event reconstruction minus the track fitting (called the master) and a second module which performed a single track fit (called a worker). A representative CPLEAR event sample contains events with two to six tracks, approximately 80% being 4-track events. Therefore a 4-workers configuration was implemented.

The principal data flow consists of event data: track banks, communicated from master to worker, and the resulting fit banks, communicated from the worker to the master. Both the track banks and fit banks are around 1 Kbytes in size and stored in a linked list structure in the master. In addition, track fitting uses long-term data. These data are independent of the master or worker and as the name suggests do not have to be communicated per event. For these reasons they were replicated over all modules. Updates of long-term data were triggered by a change in the run number in the master. This run number was therefore also communicated from the master to the workers.

The construction of the master and worker modules was relatively straightforward: the modules were composed of the individual components obtained during code decomposition. The master code is identical to the sequential code without the track fitting step. The worker code consists of program initialization/termination, the track fitting step, and the routines necessary for updating the long-term data. Additional code in both the master and worker were added to implement the data flows.

The first implementation of the parallel track fitting version was on a T805 Transputer network. The program execution was analysed with VIPER, which showed fairly good load balancing and correct program behaviour (module interaction). The measured speed-up factor was 1.5, compared with the theoretical value of 4. The visualization of the program execution over time with VIPER showed that the poor performance was due to the, expected, large Mona Lisa communication overheads.

The parallel version was subsequently ported to the next generation of Transputers, the T9000, replacing the Mona Lisa communications by native message passing library calls. The improved communications in terms of latency and bandwidth, supported by the native message passing library, resulted in a speed-up factor of 3.

To investigate the discrepancy between the measured speed-up factor of 3 and the expected speed-up factor of 4 a timing analysis was performed. This analysis shows that

the maximum track fitting time within an event (39.97 ms) is 15% above the average track fitting time. This imperfect load balancing causes the obtainable speed-up factor to drop to 3.5.

This limit of 3.5 still does not take into account the overheads introduced in the parallel version by communication and Zebra operations. Table 3 shows that these extra tasks are non-negligible in terms of CPU consumption. Taking into account the measurements in Table 3, the predicted speed-up is 3.0 as measured. An important observation is that the Zebra operations required to move the data structures between the Zebra dynamic storage area and the static communication arrays were relatively expensive: 8% of the total execution time.

Table 3
CPU breakdown on T9000 of the parallel track fitting step

Aspect	CPU in ms	CPU in %
Algorithm (fitting)	39.97	87.6
Communication	2.01	4.4
Zebra	3.67	8.0
Total	45.65	100

6 DISCUSSION

The tasks performed by typical HEP applications have substantial potential for parallel execution. This case study looked at one example of these tasks in particular, track fitting. We have shown that with minimal code changes a speed-up factor close to the theoretical maximum was achieved. The maximum theoretical speed-up factor was not obtainable due to load imbalance and communication overheads introduced by data parallelism. Indeed, track parallelism is more sensitive to load imbalance than event farming, where it is not an issue. This parallelism can be equally applied to the other tasks performed in HEP reconstruction. Scope for large-scale parallelism of this type exists in future experiments at the LHC, where for instance the number of tracks per event increases by orders of magnitude.

Communication overheads, introduced via data parallelism, could be minimized using the functionality offered by Zebra. For current and future parallel Fortran applications this functionality, the grouping and structuring of data, allows efficient data communication to be implemented. However, the Zebra data structures should be defined very carefully. As banks can be used as the unit of data distribution, independent data sets that can be processed in parallel should not be stored in a single bank, whereas interdependent data should. These two points could result in minimal communication and Zebra operation overheads. The track fitting example showed that, although no data reformatting was performed, the copying of individual track banks to communication arrays lead to significant overheads.

This case study also looked at the code modularity of CPREAD. It proved very difficult to decompose the application into its component reconstruction steps although it was developed and maintained by a code management tool. Code management tools such as Patchy and CMZ can and should be used more efficiently to achieve code modularity, a basic software engineering requirement.

Modularity should also be addressed at the level of the algorithms. Logically independent algorithms should be self-contained at the physical code level. The parallel execution of these algorithms is easier if code optimization issues are decoupled from the

algorithms. This could lead to a substantial reduction in the amount of code rewriting.

In this case study various tools were found to be necessary for a structured parallelization strategy. A profiler was needed to assess the potential for different types of parallelism. During development, a code management tool (CMZ) was needed for classical code management. A parallel programming paradigm (Mona Lisa) provided the tools for additional structuring, tuning and debugging.

In the future, good programming techniques will be used to allow efficient/effective parallelism. Indeed, one could say that developing parallel software requires careful software development and similar techniques applied to sequential software will improve its quality as well. Parallel programming techniques could be regarded as a tool for quality software development in general.

The use of fine grain parallelism allows a reduction of the resource requirements per processor. In particular, the memory requirements per processor for code, long-term data and event data can be significantly reduced. This was demonstrated in the track fitting example, where the code of one worker amounted to less than 20% of the sequential program code size. The savings of resources should be weighed against the disadvantage of suboptimal use of processors that is introduced by fine grain parallelism.

Table 4 shows the strong and weak points of the various parallelization strategies, with ratings varying between ++ (very positive), 0 (neutral) and – (negative).

Table 4
Parallelization characteristics

	Data parallelism		Algorithmic	
Aspect	Event farm	Track farm	Pipe line	Task par.
Throughput	++	+	+	–
Latency	0	++	0/–	0/+
Scalability	++	+	–	–
Resources	–	+	+	++
Effort	++	–	+	+

No single parallelization strategy appears to address the issues of latency, throughput, implementation effort, computing resources and scalability. The best approach would appear to be a combination of the various parallelization strategies. Scalability of throughput is best assured by farming out events to workers. Latency, not addressed by event farming, can be addressed by the addition of finer grain strategies. For example, the replacement of a worker by a team of subworkers, implementing a pipeline with data parallelism at each stage.

Acknowledgements

We gratefully acknowledge the support provided by the European Union via the GPMIMD project P5404, and one of us would like to thank the US National Science Foundation for funding. We would like to thank R.W. Dobinson and P. van der Stok under whose supervision the work has taken place.

References

- [1] The ATLAS Technical Proposal, CERN/LHCC/94-43, LHCC/P2, ISBN 92-083-067-0.
- [2] C. Rubbia, Commission of the European Communities, Report of the High Performance Computing and Networking Advisory Committee, 1992.
- [3] M. Michelotto, 'Delfarm, the DELPHI offline production farm', Proc. 10th Computing in High-Energy Physics Conference (CHEP'92), Annecy, France, 1992, eds. C. Verkerk and Wojcik (CERN Report 92-07).
- [4] A.J.G. Hey, 'Experiments in MIMD parallelism', in Lecture Notes in Computer Science, **366**, (Springer-Verlag 1989) pp. 28.
- [5] GP-MIMD Consortium, GP-MIMD P5404 Technical Annex for Esprit, revised 1994, Commission of the European Communities.
- [6] L. Adiels et al., Proposal for the experiment PS195, CERN/PSCC/85-6/P82.
- [7] H. Klein, J. Zoll, 'Patchy Reference Manual', CERN program library, 1994.
- [8] Zebra, CERN program library entries Q100 and Q101.
- [9] HBOOK, CERN program library, entry Y250.
- [10] FFREAD, CERN program library, entry I302.
- [11] CodeME, France, 'CMZ User's guide', 1994.
- [12] A. Schneider, Programming Parallel Computers, Ph.D. thesis, University of Geneva, 1993.
- [13] R. Schiefer, 'Viper: a tool for the Visualisation of Parallel Programs', Proc. 3rd EuroMicro Workshop on Parallel and Distributed Processing, Italy, 1995 (IEEE Comp. Soc. Press, 1995, ISBN 0-8186-7031-2) pp. 540-546.