Sparse Matrix Computations on an FFP Machine

TR88-049

October 1988

Bruce T. Smith
Raj K. Singh
Gyula A. Magó

The University of North Carolina at Chapel Hill
Department of Computer Science
CB#3175, Sitterson Hall
Chapel Hill, NC 27599-3175

This work was supported in part by NSF grant MIP-8702277 and by the Office of Naval Research, Contract N00014-86-K-0680.
Presented at Frontiers '88: The Second Symposium on the Frontiers of Massively Parallel Computation, October 10-12, 1988. Fairfax, Virginia

Sparse Matrix Computations on an FFP Machine[*] (Preliminary Version)

B.T. Smith, R.K. Singh and G.A. Magó
Department of Computer Science
University of North Carolina
Chapel Hill, NC 27599-3175

Abstract

We describe and analyze an algorithm for performing Gaussian elimination on sparse linear systems with an FFP Machine, a small-grain parallel computer. Given an equation $Ax = b$, where A is an $n \times n$ matrix, our algorithm yields a permuted upper-triangular system, from which we obtain x by back-substitution. If A has e non-zero entries and if f fill-ins are created during elimination, then our algorithm solves the system in $O(h \times (e + f))$ time, using $O(e + f)$ processing elements. (The parameter h is the height of the FFP Machine's connection network, which is $O(\log(e + f))$.) The algorithm makes no assumptions about the structure of A and requires no pre-processing. The pivot order may be given in advance, or it may be chosen at run-time by the Markowitz heuristic with only a linear increase in cost. We also present results of simulations on sample problems, both randomly generated and from the Boeing-Harwell set. The results of the simulations, in operation counts, are used to estimate the performance of an FFP Machine hardware prototype.

The Problem

Matrix problems are encountered in such disciplines as physics, engineering, econometrics and operations research. Common to many of these problems is the occurrence of matrices that are sparse, i.e., many elements of the matrices are zero.

The matrices associated with problems from physical sciences and engineering, in addition to being large and sparse, are frequently structured. For example, they may be symmetric, diagonally dominant, positive definite or banded. Hence, they lend themselves to an efficient solution by a variety of special techniques.

[*]This work was supported in part by NSF grant MIP-8702277, and by the Office of Naval Research, Contract N00014-86-K-0680.

In contrast, problems in such areas as operations research, non-linear optimization or management can yield unstructured sparse matrices. As a consequence, more general sparse matrix techniques have been developed for less structured problems [3]. Such techniques are characterized by relatively few operations per data element and an unpredictable, dynamic growth of data structures. These issues, as summarized in [2], necessitate dynamic storage management and efficient data structure handling methods.

In the prevailing paradigm of parallel computation, we address these issues by considering a parallel form of the direct Gaussian elimination method, augmented by the Markowitz heuristic to establish pivot ordering, for the solution of systems of linear equations. The parallel approach we take is based on a computational model of a fine-grain, distributed-memory, network-based MIMD computer called the *FFP Machine* (*FFPM*). In keeping with our interest in less structured problems, we make no assumptions about the structure of the coefficient matrices. We give a brief description of the FFPM architecture in the next section. The algorithm and results of its theoretical and experimental analyses are presented in the following sections.

FFP Machines

FFP Machines are a family of small-grain, parallel computers [7] designed to execute the FFP languages of Backus [1]. An FFPM, as shown in Figure 1, consists of a linear array of PEs, called the *L-array* of *L-cells*, connected to each other and to an interconnection network of *T-cells*. Each L-cell is a small, programmable computer with an ALU and a very small memory. There is also a *front-end machine* that handles I/O, but for the most part FFP execution takes place in the L-cells and T-cells. In a simple FFPM, as shown in Figure 2, the T-cells are organized as a binary tree with an L-cell at each leaf and the front-end machine above the root. A useful FFPM would contain at least a few thousand L-cells.

FFP's primary data structure is the sequence, and the FFPM treats sequences as *dynamic arrays* [6]. That is, it is possible to randomly access the elements of a sequence, and at the same time easy to add or delete elements at arbitrary positions. Moreover, elements of a sequence may be accessed either by relative position in the sequence or by content, as in associative memory[5]. Many of these operations correspond to FFP's primitive functions, but an FFPM can support functions not in Backus's original language [8,9]. This paper shows how an FFPM can provide operations on sequences that are well suited to sparse matrix computation. Such operations may be added as new FFP primitive functions to be used in Gaussian elimination and other computations.

An FFP expression is placed in the L-array, each symbol in a different L-cell, and the FFPM evaluates the expression by rewriting innermost function applications, known as *reducible applications* or *RA*s, until no more applications remain. The FFPM operates in machine cycles of partitioning, execution and storage management. *Partitioning* creates an independent sub-machine for each RA, consisting of the L-cells holding the RA and a binary tree of T-cells, as shown in Figure 2. During *execution*, the sub-machine's L-cells rewrite their RA by performing local computations and exchanging messages. Message packets are

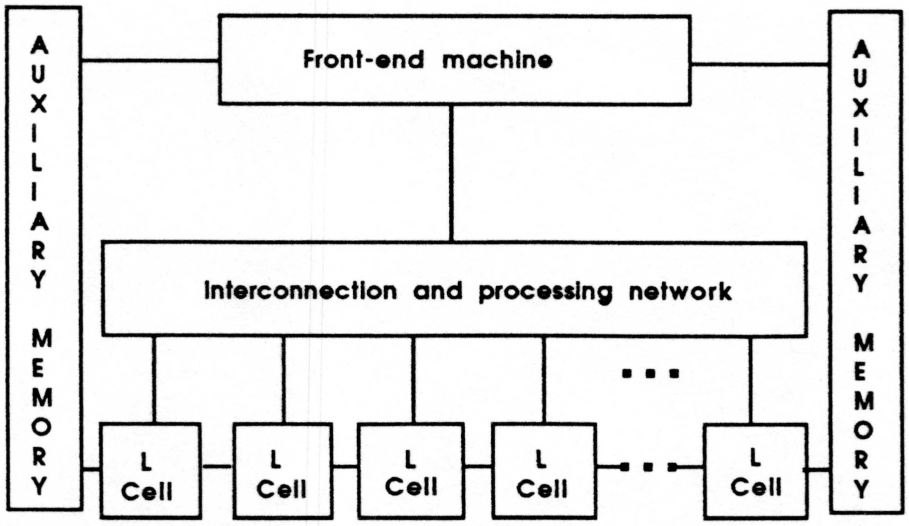

Figure 1: The components of an FFP Machine.

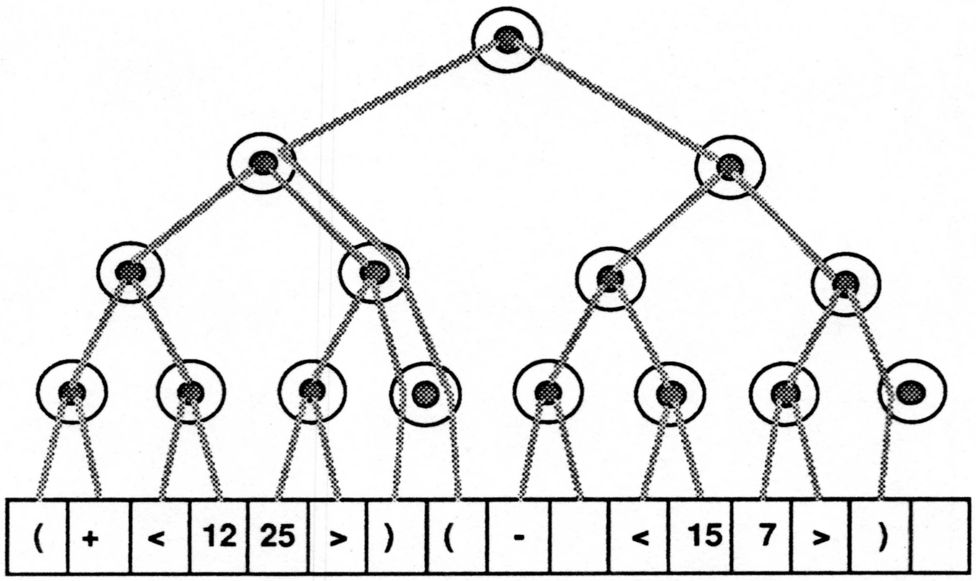

Figure 2: Partitioning creates a submachine for each RA.

sent from the L-cells and contain instructions on how the T-cells shall treat them. The sub-machine's T-cell network can select or sort messages, broadcasting the result to all L-cells in the sub-machine. The T-cell network can also perform parallel prefix operations. A sub-machine might request extra space and suspend execution, as occurs when an FFP expression grows during evaluation. During *storage management*, the contents of the L-cells, *the L-cell images*, are shifted through the L-array, retaining their left-to-right order, to make empty L-cells available where needed, as shown in Figure 3. The shifted L-cell images obtain a new T-cell network in the next partitioning stage, and execution continues.

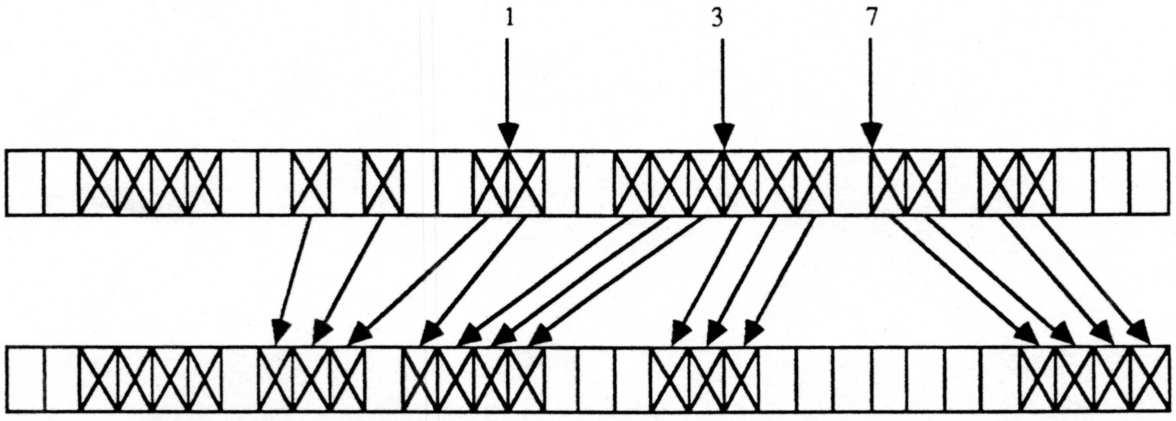

Figure 3: During storage management, L-cell images are shifted to provide empty L-cells where requested.

The Algorithm

The system of equations $Ax = b$ is given row-wise, as an FFP sequence

$$<row_1, \ldots, row_n>$$

and each row_i is of the form

$$<a_{i,j(i,1)}, \ldots, a_{i,j(i,k_i)}, b_i>$$

where the i^{th} row of A contains k_i nonzero entries in columns $j(i,1), \ldots, j(i,k_i)$. The entries $b_1, \ldots, b_n$ are from the vector b. Each $a_{i,j(i,l)}$ contains the corresponding coefficient in A along with its row and column numbers, and it has space for some additional values used during the computation. (For example, values used in choosing pivot elements by the Markowitz heuristic.) The number of values required per entry is independent of the matrix size, and for this reason, an entry might be an FFP atom and reside in one L-cell. In that case, new primitive FFP functions would be required to operate on its components. Or, it could be a small FFP sequence, manipulated by the standard FFP functions. The choice would influence time and space performance on a particular FFPM, but it has no effect on the algorithm or its analysis.

We will describe the algorithm in three parts. First, and in the greatest detail, we present the basic Gaussian elimination algorithm. Second, we describe how the solution is obtained via back-substitution. Finally, we discuss how to modify the basic algorithm to choose pivots at runtime, using the Markowitz heuristic.

Gaussian elimination: Initially all rows are "active". We choose a (non-zero) pivot element in an active row and subtract the appropriate multiple of that row from the other active rows, so that afterwards they have 0s in the pivot element's column. Then we mark the pivot element's row as inactive and repeat the process. When there are no non-zero elements in active rows – after n steps, if A is non-singular – we have the system in (permuted) upper triangular form, and we can find the solution easily by back-substitution.

The algorithm proceeds by two types of operations: *global operations*, where the whole matrix is contained in one RA, and *row operations*, where each row is contained in its own RA. The global operations choose the pivot, broadcast the pivot row and update the values for the Markowitz heuristic. The row operations subtract the appropriate multiple of the pivot row, and create fillins. Since Gaussian elimination is so familiar, it suffices to describe in detail the "inner loop", as it applies to one active, non-pivot row.

Suppose we have a 5×5 system of equations (neither large nor sparse) that contains the following:

$$\begin{array}{cccccl} 3x_1 & & +x_3 & -4x_5 & = 3 & \qquad (\text{row } 2 \ = \ \text{pivot row}) \\ & x_2 & +2x_3 & & = 5 & \qquad (\text{row } 3) \end{array}$$

and that $a_{2,3}$ has been chosen as the pivot. We will trace the effect of a single step on the third row. Figure 4 (a) shows row 3 at the beginning of the step. The column numbers of the entries in the pivot row are broadcast to the entire matrix in a global operation, and each active element counts those less than or equal to its own, shown as *temp* in Figure 4 (b). Each element also records whether one of these matches its own column number, shown as *hit*. Next, each entry computes the number of elements in the pivot row between itself and its left neighbor. (This is done separately in each active row, by a parallel prefix computation.) This value, minus 1 if *hit* is *true* (*temp* in Figure 4 (c)) gives the number of fillins to be created to the left of the entry. The fillin entries are created after an FFPM storage management cycle. The elements who are going to be "hit" on this step compute their relative order within the row, *temp* in Figure 4 (d). (This is another parallel prefix computation, done separately in each active row.) Now, in another global operation, the values in the pivot row

(a)

value	1	2	5
row	3	3	3
col	2	3	b
active	T	T	T
p-row	F	F	F
p-col	F	T	F
hit	-	-	-
temp	-	-	-

(b)

value	1	2	5
row	3	3	3
col	2	3	b
active	T	T	T
p-row	F	F	F
p-col	F	T	F
hit	F	T	T
temp	1	2	4

(c)

value	1	2	5
row	3	3	3
col	2	3	b
active	T	T	T
p-row	F	F	F
p-col	F	T	F
hit	F	T	T
temp	1	0	1

(d)

value	0	1	2	0	5
row	3	3	3	3	3
col	-	2	3	-	b
active	T	T	T	T	T
p-row	F	F	F	F	F
p-col	F	F	T	F	F
hit	T	F	T	T	T
temp	1	-	2	3	4

(e)

value	0	1	2	0	5
row	3	3	3	3	3
col	1	2	3	5	b
active	T	T	T	T	T
p-row	F	F	F	F	F
p-col	F	F	T	F	F
hit	T	F	T	T	T
temp	3	-	1	-4	3

(f)

value	-6	1	0	8	-1
row	3	3	3	3	3
col	1	2	3	5	b
active	T	T	T	T	T

Figure 4:
Snapshots of one row of a system of equations during one step of Gaussian elimination. Each column shows the contents of one entry (one L-cell). The coefficient, row and column number are shown as *value*, *row* and *col*, respectively. ("*col* = *b*" identifies elements of vector *b*.) Booleans *active*, *p-row* and *p-col* show that this is an active row, not the pivot row, with an entry in the pivot column. The Boolean *hit* shows which entries correspond to (non-zero) entries in the pivot row. The values in *temp* are used for creating fillins in (b)–(d), and are the pivot row entries in (e).

are broadcast once again, sorted by column number in the T-cell network, and each entry to be hit receives the column number and pivot row coefficient indicated by its index from the previous step. This is shown in Figure 4 (e), where *temp* now is the value from the pivot row. Finally, in another row operation, the element in the pivot column broadcasts the quotient *value* $\div$ *temp* and every "hit" entry performs the multiplication and subtraction.

A subtle point of the algorithm is worth mentioning. The newly created fillins do not have column numbers until the pivot row is broadcast the second time. (Shown by "-" in Figure 4 (d).) Before the pivot row is broadcast the first time, there is no way, in general, for another row to know how many fillins it will need. Nor is there any way to know where they will be created. As we remarked earlier, an L-cell is fairly small. It is reasonable to assume that it can store several small integers, but it is not reasonable to assume that it can store arbitrarily many column numbers. So, the reason for the second broadcast of the pivot row is simply that we want to create all fillins for one step of the Gaussian elimination in one FFPM storage management cycle, rather than deciding as each pivot row entry arrives whether it requires a fillin.

We analyze the Gaussian elimination algorithm as follows. An L-cell does a bounded number of arithmetic operations for each value it receives, so we may restrict our attention to the time required for communication. Suppose there are e entries in the original system and that, for the sequence of pivots we choose, f fillins are created. (For convenience we assume A is non-singular, so that $e \geq n$ and so that Gaussian elimination will take n steps.) Clearly, we need $O(e + f)$ L-cells, and the height h of the binary tree of T-cells needed to connect them is $O(\log(e + f))$. The cost of shifting in the original problem and of making room for fillins (over all n steps) is $O(e+f)$, since the amount of space requested bounds the cost of storage movement. Each entry or fillin is in a pivot row exactly once, so it is broadcast twice for that purpose. Thus $O(e + f)$ messages go through the root of a submachine of height $O(h)$ in n message waves. That requires $O(n \times h)$ time to fill the T-cell pipeline n times, plus $O(e + f)$ time for the messages to arrive in sequence. Thus there is a total of $O(h \times (e + f))$ time for broadcasting the pivot rows. There are a fixed number of row operations for each step, and these only broadcast single values and perform parallel prefix computations. Each of these requires time $O(h)$, so over n steps this is also $O(h \times (e + f))$. We conclude that the entire Gaussian elimination, then, requires $O(h \times (e + f))$ time.

Back-substitution: Once we have the system in permuted upper-triangular form, it is easy to finish solving the system by back-substitution. Each row (in an independent row operation) determines if it has the value for a variable by seeing if it has exactly one non-zero $a_{i,j}$ entry. (One parallel prefix operation can count the number of entries.) If so, it computes the value of x_j (by one division) and broadcasts it on the next global operation and becomes inactive. Rows that remain active receive the x values and eliminate those variables from their equation. Eventually, assuming the system has a unique solution, every row will have found the value of one variable. The time for this is $O(h \times n)$, which is again $O(h \times (e + f))$.

Choosing pivots: We modify the Gaussian elimination algorithm to choose pivots at runtime by the Markowitz heuristic, as follows. Let each entry keep two additional values, nr and nc. The number of other non-zero elements in an entry's row will be nr, and the number of other non-zero elements in its column will be nc. We can initialize the nr and nc values by broadcasting the row and column numbers of the original matrix entries, and letting each entry count the number of matches. Every time an entry becomes inactive (or zero), it globally broadcasts its row and column numbers, and other (active) entries decrement their nr and nc values, respectively, if the row or column numbers match. Every time a fillin is created, it globally broadcasts its row and column numbers, and other (active) entries increment their nr and nc values, respectively. To choose a pivot, each active entry computes the product of its nr and nc values, and the T-cell network chooses a minimal one. The additional time required for this is $O(h \times (e + f))$, with the analysis much like that for the basic algorithm.

The Simulations

Our simulations supplement the analysis presented in the previous section. The set of sample problems contains both randomly generated matrices and matrices from various real-life applications (obtained from the Boeing-Harwell sparse matrix collection[4]). The simulator was written in C and run on a Convex C-220 system (two processors), with an implicit vectorizing/parallelizing compiler.

The simulator is in two parts. The first part is used to study fillin behavior and to count the number of floating point operations in triangularizing a coefficient matrix. It omits the back-substitution computation, but it does count the work required to choose pivots by the Markowitz heuristic. The output of the first part of the simulator is input to the second part, which uses parameters for an FFPM hardware prototype to provide performance estimates. In the remainder of this section, we describe the two parts of the simulator and summarize its results.

The coefficient matrices are transformed into a 0/1 representation, i.e., non-zero entries are replaced by 1s and zero entries by 0s. Due to the relatively small size of our sample problems and the simplicity of programming, we use dense matrix representations for storing the 0/1 matrices. We note, however, that the sparse nature of the problem is maintained in that only operations for non-zero entries are counted. The simulator follows the basic Gaussian elimination technique. The Markowitz heuristic, used to choose pivots, tends to minimize the number of fillins, retaining the sparsity of a matrix. Numerical stability issues were not taken into account, but one can add a threshold based criterion for selecting a numerically stable pivot without changing the complexity of the algorithm.

The results of the first part of our simulations on randomly generated matrices and matrices from the Boeing-Harwell collection are presented in Tables 1 and 2, respectively. The number "entries" is the number of non-zero entries in the initial coefficient matrix, "fillins" is the number of entries created during Gaussian elimination, and "sequential operations" is the sum of divisions, multiplications, and subtractions performed. The results presented in Table 1 were obtained by averaging the results of 5 separate trials on random matrices with

order	entries	fillins	mean size of pivot row	sequential operations
100	516	746	7	9955
200	1995	9380	29	548937
300	4466	31332	60	3560740
400	7922	68512	96	11749105
500	12421	125662	139	29876633

Table 1: Randomly generated matrices.

order	discipline	entries	fillins	mean size of pivot row	sequential operations
180	astrophysics	2659	145	7	30628
199	stress analysis	701	703	4	5003
541	chemical kinetics	4285	11442	14	301068
600	oil recovery	13760	30304	37	2060332
822	linear programming	4790	1701	5	15132

Table 2: Matrices from the Boeing-Harwell collection.

an average density of 5%.

Next we present the performance estimates produced by the second part of the simulator. Due to the pipelined nature of the FFPM architecture, there is a considerable overlap between useful computation and communication. Therefore, it is most meaningful to account for the net solution time for a problem. Messages are typically made up of several packets, based on data size and the format required by the T-cells. The speed of operation is directly governed by the parameters of hardware employed. The parameters used in this model were derived from the specifications of a transputer (20 MIPS) operating at the clock rate of 20 MHz. The communication channels in the tree-network were modeled to be bit-serial with peak throughput of 10 Mbps. The communication channels among cells in the L-array were considered to be byte wide with a peak data rate of 10 MBps.

The results are shown in Tables 3 and , corresponding to the entries in Tables 1 and 2. Here "parallel operations" is the sum of all divisions, multiplications, and subtraction operations performed in each disjoint sub-machine, and corresponds to the operations performed on the last message received by an L-cell. We have presented the results of total solution time for the problems under two distinct situations, the first where the pivoting sequence was defined before elimination, and the second where Markowitz heuristic was employed to determine the pivoting sequence during the computation. (Both use the same ordering for each matrix.) The cost of choosing pivots at runtime is surprisingly high, but this is due, in part, to the size of the message packets required in the design being simulated.

order	parallel operations	time (msec)	
		given order	runtime pivoting
100	1338	18	61
200	7084	102	486
300	19989	293	1497
400	41042	601	3172
500	72579	1064	5708

Table 3: Performance estimates (random matrices).

order	parallel operations	time (msec)	
		given order	runtime pivoting
180	2675	36	128
199	2108	27	77
541	12078	161	693
600	25682	394	1861
822	8151	128	352

Table 4: Performance estimates (Boeing-Harwell matrices).

Conclusions

We have described and analyzed an algorithm for performing Gaussian elimination on sparse matrices with an FFPM. This work is relevant to FFPM development in two ways. First, it demonstrates how the basic FFP language can be extended with new primitive operations, to better support operations on sparse matrices. Secondly, it provides performance estimates for an FFPM hardware prototype on a real-life problem.

Acknowledgements

We would like to thank Dr. A.M. Erisman and Dr. R.G. Grimes of Boeing Computer Services, Seattle, Washington, for providing us with Boeing-Harwell collection of sparse matrices. Mr. Mike Padrick and Mr. Larry Mason, of Academic Computing Services at UNC, provided assistance with systems related issues on the Convex computer. In the Department of Computer Science, Vernon Chi, William Partain and Donald Stanat provided encouragement and valuable suggestions, and Deborah Stogner produced the figures.

References

[1] J. Backus. Can programming be liberated from the von Neumann style? A functional style and its algebra of programs. *Communications of the ACM*, 21(8):613–641, 1978.

[2] I.S. Duff. The use of vector and parallel computers in the solution of large sparse linear equations. In *Large Scale Scientific Computing*, Birkhäuser, 1986.

[3] I.S. Duff, A.M. Erisman, and J.K. Reid. *Direct Methods for Sparse Matrices.* Clarendon Press, 1986.

[4] I.S. Duff, R.G. Grimes, and J.G. Lewis. *Sparse Matrix Test Problems.* Technical Report, Computer Science and Systems Division, Harwell Laboratory, 1987.

[5] G.A. Magó. Data sharing in an FFP Machine. In *Conference Record of the 1982 ACM Symposium on LISP and Functional Programming*, pages 201–207, 1982.

[6] G.A. Magó and W. Partain. Implementing dynamic arrays: a challenge for high-performance machines. In *Proceedings of the Second International Conference on Supercomputing*, pages 491–493, 1987.

[7] G.A. Magó and D.F. Stanat. The FFP Machine. In *High-Level Language Computer Architecture*, Computer Science Press, 1988.

[8] D. Middleton and B.T. Smith. FFP Machine support for language extensions. In *Proceedings of the 19th Hawaiian International Conference on Systems Sciences*, pages 59–65, 1986.

[9] B.T. Smith and D. Middleton. Exploiting fine-grained parallelism in production systems. In *Proceedings of the Seventh Biennial Conference of the Canadian Society for Computational Studies of Intelligence*, pages 262–270, 1988.