

Classification and Performance Evaluation of Hybrid Dataflow Techniques With Respect to Matrix Multiplication

Martin Beck*, Theo Ungerer** and Eberhard Zehendner***

*Department of Mathematics, University of Jena, Leutragraben 1, O-6900 Jena, Germany
Email: beck@mathematik.uni-jena.dbp.de

**Department of Computer Science, University of Karlsruhe, P.O.Box 6980, W-7500 Karlsruhe 1, Germany
Email: ungerer@ira.uka.de, ungerer@uni-augsburg.de

***Department of Mathematics, University of Augsburg, Universitaetsstr. 2, W-8900 Augsburg, Germany
Email: zehendner@uni-augsburg.de, Phone +49-821-5982176, Fax +49-821-5982200

Abstract

This paper classifies hybrid dataflow techniques due to the instruction issuing technique. A software simulation is conducted to compare fine-grain dataflow to several hybrid dataflow techniques: multithreaded dataflow with direct token recycling as used in Monsoon, multithreaded dataflow with consecutive execution of the instructions within a thread as used in the Epsilon processors and in EM-4, dataflow with complex machine operations as proposed for the SIGMA-1, and large-grain dataflow presuming a RISC processor respectively a superscalar processor in the execution stage.

All dataflow techniques show good scalability and effectively compensate delays caused by the network and by structure access, provided that load is sufficient. The achieved performance accelerations differ as follows: Large-grain dataflow proved superior to all other techniques provided that a superscalar processor is used, and performs at least equal to the other techniques with a RISC processor. Multithreaded dataflow is an improvement over fine-grain dataflow but suffers from a large overhead and thus does not achieve a considerable speedup. This is even worse for the cycle-by-cycle interleaving technique of Monsoon. Complex machine operations perform slightly better than large-grain dataflow with a RISC processor. It can be a useful enhancement to other techniques.

1. Introduction

Multiprocessors are well suited for exploiting task level parallelism provided that communication is sparse and granularity of tasks coarse. However, an architectural design based on extensive exploitation of parallelism should not neglect the potential fine-grain parallelism that might be available in an algorithm but cannot be efficiently exploited by coarse-grain techniques. Utilizing fine-grain parallelism, i.e., instruction level parallelism or sequential threads with about 20 instructions, can bridge remote memory access or synchronization delays by switching the PE of a multiprocessor to a different thread. The main problem for the exploitation of fine-grain parallelism is that most conventional processors incur a very large overhead during a context switch. Multithreaded von Neumann architectures [1, 2, 3] and dataflow architectures [4, 5, 6, 7, 8, 9] are two approaches to exploit fine-grain parallelism.

In a multithreaded von Neumann architecture each processing element is equipped with a von Neumann processor that can perform a very fast context switch between several threads already loaded. Creation of threads is not too costly. Instruction level parallelism is not supported.

When using the dataflow scheme, programs are compiled into dataflow graphs that represent the data dependencies among instructions. Scheduling is data-driven: an instruction is ready to execute as soon as all required operands are available. The availability of operands is signaled by tokens that conceptually are propagated on the arcs of the dataflow graph. Dataflow architectures use token matching prior to instruction execution. This synchronization scheme is able to exploit all possible parallelism at instruction level but, unfortunately, leads to superfluous control overhead when executing sequences of instructions.

To reduce the number of synchronization events in dataflow architectures three techniques have been proposed:

- threaded dataflow techniques [7, 8, 9] statically schedule instructions for efficient sequential execution;
- complex machine operations [10, 11, 12, 13, 14], for instance vector operations [10] or relational data bank operations [11], are used as dataflow macro actors that are processed in SIMD mode but activated by the dataflow scheme;
- large-grain dataflow joins the von Neumann and the dataflow model of computation by processing sequences of instructions in von Neumann style, but activating them as dataflow macro actors [15, 16].

A study of these *hybrid dataflow techniques* with respect to performance acceleration and scalability seems worthwhile. Our main goal is an assessment of the possible reduction of the overhead caused by the instruction level synchronization in fine-grain dataflow that can be reached by these instruction issuing techniques. Consequently, this contribution does not cover the latency caused by access to non-strict data structures [5, 9] and by synchronization loads [17]. Multithreaded von Neumann architectures are not included in the analysis because they do not use dataflow techniques; in particular they do not apply matching as an instruction issuing technique, and their processor pipeline stages differ from those of dataflow architectures.

Section 2 of this paper classifies hybrid dataflow architectures due to the three techniques: multithreaded dataflow, dataflow with complex machine operations, and large-grain dataflow. These techniques are contrasted to fine-grain dataflow.

We have evaluated these hybrid dataflow techniques by a software simulation. Section 3 presents the common architectural model used as a basis of our simulator and deviations of the model to implement the different techniques. The matrix multiplication algorithm used as sample program for the simulations is described, succeeded by alterations for the implementations of the regarded instruction issuing techniques.

Section 4 shows the results of the software simulation with varying number of PEs, network delays and structure access delays. The simulation results are summarized and assessed in Section 5.

2. Classification Scheme

This section classifies dataflow architectures according to their instruction issuing technique as fine-grain dataflow, multithreaded dataflow, dataflow with complex machine instructions, or large-grain dataflow. A typical dataflow architecture using one of these techniques consists of a number of identical PEs connected through a communication network to a globally addressed structure store, designed to cope with large data structures. Each PE is organized as a cyclic pipeline composed of a firing stage and an execution stage, and buffered by a token queue. This pipeline is synchronous for fine-grain and multithreaded dataflow, and usually FIFO-buffered between firing and execution stages for the other two techniques. The token queue contains tagged tokens produced by previously executed instructions.

The firing stage contains a matching unit and an instruction fetch unit. For tokens, destined for dyadic operations, a search for a matching partner token is performed by the matching unit. If no match is found, the token is stored to await the arrival of its partner. Otherwise the token pair is forwarded to the execution stage, together with an opcode retrieved by the instruction fetch unit. If a token is destined for a unary operator the matching phase is bypassed. Depending

on the specific architecture, the matching unit precedes the instruction fetch unit [4, 5], succeeds it [9], or both work simultaneously on the same token [6].

The execution stage contains an ALU and a destination unit. The destination unit combines new tags for the result tokens with a value computed by the ALU. Each result token is forwarded to the local token queue, to another PE if the destination address is non-local, or to a structure store if the operation is an access to a structure.

2.1 Fine-Grain Dataflow

In fine-grain dataflow architectures [4, 5, 6] each instruction denotes an operation of low complexity. The issuing of a monadic instruction consumes one token from the token queue, the issuing of a dyadic instruction demands two tokens. In the latter case two matching operations have to be performed by the firing stage to issue a single instruction to the execution stage. The following problems arise with fine-grain dataflow:

- For a dyadic instruction the first matching operation does not trigger the issuing of the instruction, leaving a bubble in the execution stage of the synchronous pipeline.
- A dataflow graph can be degenerate to a sequence of instructions, or may exhibit only a low degree of instruction level parallelism. In this case passing tokens through the circular pipeline induces a high delay since the successive instruction is activated after a token has been passed through the whole cycle. Therefore fine-grain dataflow performs poorly with sequential code, especially when load is low.
- A context switch occurs after each instruction execution, thus no use of registers is possible. Use of registers could optimize the access time to data values, avoid pipeline bubbles caused by dyadic instructions, and reduce the total number of tokens during program execution.

2.2 Multithreaded Dataflow

To solve the problems of fine-grain dataflow, each subgraph that exhibits a low degree of parallelism can be identified within a dataflow graph and transformed into a sequential thread. By multithreaded *dataflow* we understand a technique where a thread of instructions is issued consecutively by the matching unit without matching further tokens except for the first instruction of the thread. Multithreaded dataflow covers the repeat-on-input technique in the Epsilon-1 and Epsilon-2 processors [7], the strongly connected arc model of EM-4 [8, 18], and the direct recycling of tokens in Monsoon [9]. Data passed between instructions from the same thread is stored in registers instead of writing them back to memory. These registers may be referenced by any succeeding instruction in the thread. Thereby single-thread performance is improved. The total number of tokens needed to schedule the instructions of a program is reduced thus saving hardware resources. Pipeline bubbles are avoided for dyadic instructions within a thread.

Two multithreaded dataflow execution techniques can be distinguished: the direct token recycling, or consecutive execution of the instructions of a single thread. The first technique, used by the Monsoon dataflow computer, allows a particular thread to occupy only a single slot in the 8-stage pipeline, which implies that at least 8 threads must be active for a full pipeline utilization to be achieved. This cycle-by-cycle instruction interleaving of threads is also used by some multithreaded von Neumann computers like HEP [1].

To optimize single-thread performance the Epsilon-processors and the EM-4 execute instructions from a thread consecutively. The circular pipeline of fine-grain dataflow is retained. However, the matching unit has to be enhanced with a mechanism that, after firing the first instruction of a thread, delays the matching of further tokens in favor of issuing all instructions of the thread consecutively. By this mechanism cycling of tokens through the pipeline for the activation of the next instruction is suppressed.

2.3 Dataflow with Complex Machine Operations

Another technique to reduce the instruction level synchronization overhead is the use of complex machine instructions, for instance vector instructions. These instructions can be implemented by pipeline techniques as in vector computers. Structured data is referenced in block rather than element-wise, and can be supplied in a burst mode. This deviates from the I-structure scheme [5, 9] where each data element within a complex data structure is fetched individually from a structure store. A further advantage of complex machine operations is the

ability to exploit parallelism at the subinstruction level. Therefore the machine has to partition the complex machine operation into suboperations that can be executed in parallel. The use of a complex machine operation may spare several nested loops.

The use of a FIFO-buffer allows to decouple firing stage and execution stage to bridge the different execution times within a mixed stream of simple and complex instructions issued to the execution stage. As a major difference to conventional dataflow architectures tokens do not carry data (except for the values "true" or "false"). Data is only moved and transformed within the execution stage. This technique is used in the Decoupled Graph / Computation Architecture [10], the Stollmann Dataflow Machine [11], the LGDG Machine [12], and the ASTOR architecture [19]. These four architectures combine complex machine instructions with large-grain dataflow, described in the next subsection.

The structure-flow technique [14] proposed for the SIGMA-1 enhances an otherwise fine-grain dataflow computer by structure load and structure store instructions, that move for instance whole vectors from or to the structure store. The arithmetic operations are executed by the cyclic pipeline within a single PE.

2.4 Large-Grain Dataflow

As with multithreaded dataflow, sequences of instructions can be compounded to a thread. The thread is represented as a macro dataflow actor in the dataflow graph. Large-grain dataflow computers [10, 12, 15, 16] activate macro dataflow actors by the dataflow principle but execute the represented sequences of instructions following the von Neumann principle. Off-the-shelf microprocessors can be used as implementations of the execution stage, that is capable of functioning as a conventional processor running a single thread in von Neumann style.

3. Simulation Model

A software simulation has been conducted to reveal a performance comparison of the hybrid dataflow techniques to fine-grain dataflow. We emphasize that we did not intend to compare specific architectures, but to derive some insight into the differences between the four instruction issuing techniques described above.

Our simulation determines the elapsed time for the execution of the complete program, measured in simulation time steps. This reflects our view that performance gain should be measured by the achieved reduction of the execution time, and distinguishes our approach from other surveys where performance is measured as utilization of the PEs.

The simulator emulates dataflow multiprocessors with PEs characterized by an 8 stage cyclic dataflow pipeline, an explicit token store, and an instruction set as implemented in Monsoon [9]. Due to the abstract level of the architectural features, we disclaimed precise timing characteristics. Nearly all organizational or scalar arithmetic instructions are supposed to persist for a single time step in each pipeline stage. However, the memory allocation instruction "get-frame", as well as all instructions accessing a structure store or traversing the network have to be weighted differently. For each "get-frame" instruction that allocates an activation frame of fixed size in the explicitly addressed token store of the matching unit [9] we account for 10 time steps.

When distributing function calls over the various PEs of the multiprocessor we have to consider network delays. The delay of a single token transmission over the network depends on the network bandwidth, network bottlenecks, and the appearance of network contention.

The delay of structure access is even more difficult to assess. Memory access conflicts, non-strict data structures, synchronization loads, and the distinction between local and remote structure storage have to be modeled in excess to the network delays.

The delays caused by the network and the structure access usually are non-deterministic. We simplify the simulation model by using fixed network and structure access delays within a single run of the simulator, but varying these parameters over several runs. Network delays are assumed to be orders of magnitudes higher than the execution time of a single instruction; we

chose to vary it between 10 and 10000 time steps. Structure access delays are varied from 1 time step, simulating local access to on-chip structure storage, up to 10000 time steps for a remote memory access.

The simulation model is adapted to the various instruction issuing techniques as follows: Fine-grain dataflow uses no threads and no registers. When simulating multithreaded dataflow we distinguish two different approaches: the 8 stage cycle-by-cycle interleaving of threads and use of 8 sets of 3 registers each in the Monsoon processor, versus the consecutive execution of the instructions within a thread as in the Epsilon-processors and in the EM-4.

When simulating dataflow with complex machine operations we adopt the structure-flow processing scheme of the SIGMA-1 [14]. When a structure-flow generation instruction is executed in the PE, a single trigger token is passed to the structure store, and a structure-flow consisting of all demanded data values is returned to the matching unit of the PE.

Large-grain dataflow architectures use conventional von Neumann processors as execution stages. Large-grain dataflow simulation deviates from the principal simulation model with respect to the pipeline organization and the memory access. We assume that data is already loaded in the local memories of the PEs and in consequence we do not cover remote structure access. Due to the varying execution times of instruction sequences we further assume an elastic pipelining of the units in the firing stage, i.e., when the execution stage is busy executing a sequential code block, the units of the firing stage are supposed to idle. FIFO-buffering of executable instruction packets between firing stage and execution stage may be a slightly better technique. The instruction set and the execution time model of the execution stage is highly processor-dependent. We consider both an instruction set that is designed according to the RISC philosophy and a superscalar instruction set and execution model according to the IBM/R6000 [20].

As a sample program we use an algorithm whose implementations are capable of duly reflecting the peculiarities of the different instruction issuing techniques. Multiplication of two square ($n \times n$)-matrices using the inner product technique seems to be an adequate choice since the code contains nested loops and structure accesses, and the algorithm has a single size parameter n . Moreover, the algorithm is deterministic and its control flow does not depend on the input data.

In all cases we use concurrent dataflow loops for the two outer loops. A uniform distribution over the PEs is done at compile-time for the iterations of the outermost loop only. The innermost loop is adapted to the different instruction issuing techniques as follows:

- In fine-grain dataflow the innermost loop is a concurrent dataflow loop, executed by a single PE.
- For multithreaded dataflow we distinguish two approaches:
For the direct token recycling model of Monsoon we follow the matrix multiplication algorithm of Traub [21], replacing fan-out of tokens by threads in the outer loops and a sequential ordering of the iterations of the inner loop due to a 1-bounded loop approach. All iterations of the inner loop execute sequentially within a single frame, thus saving the execution of get-frame instructions. Each time two tokens are produced by an instruction in the dataflow graph of the inner loop, a fork instruction is introduced and a thread is initiated. By this strategy many very small threads are created with the goal to hold the pipeline within a single PE busy by executing threads (at least 8 threads are needed).
The second approach simulates the repeat-on-input model of the Epsilon processors, where the instructions of a thread are executed consecutively. Therefore threads should be as long as possible. So we use the same algorithm as in the Monsoon model with the exception that the inner loop iterations are divided into two threads only.
- For dataflow with complex machine operations the whole innermost loop is replaced by two vector-load instructions, an inner-product operation and a scalar-store instruction. The execution times of the load and inner product instructions grow linearly in the size parameter n .
- In large-grain dataflow the innermost loop is mapped to a sequential loop code that can be executed on a von Neumann (RISC or superscalar) processor, accessing only local memory.

The code is processor-dependent. We assume $8*n+6$ time steps for the execution of an inner loop iteration in case of the RISC-model, and $2*n+11$ time steps for the superscalar model as proposed in [20].

4. Simulation Results and Analysis

In the sequel we discuss the simulation results with regard to the following questions:

- What degree of runtime acceleration is achieved by the various hybrid dataflow techniques relative to fine-grain dataflow? How does the acceleration depend on the number of PEs used?
- How does the network delay influence the results?
- How does the structure access delay influence the results?

We visualize the simulation results for a fixed matrix size parameter $n=16$. The number of PEs has been varied between 1 and 16. Figure 1 shows the elapsed time versus the number of the PEs used with fixed structure access delays of 1 time step and network delays of 100 time steps; figure 2 shows the achieved efficiency relative to large-grain dataflow with a superscalar processor and 1 PE. For a small number of PEs (up to 4) all techniques show a nearly linear speedup that degrades for a larger number of PEs; this degradation is due to the reduced load on the individual PEs. Thus we expect good scalability for all techniques provided the problem size is large enough.

However, for any fixed number of PEs the degree of runtime acceleration depends heavily on the used instruction issuing technique, for large-grain dataflow also on the type of processor used as execution stage.

In particular, the Monsoon-type of multithreaded dataflow does not achieve a speed improvement compared to fine-grain dataflow. This phenomenon can be explained viewing the Monsoon-type multithreaded dataflow code that contains a large overhead due to the use of short threads. This overhead is caused by load and store instructions, because each thread has to load operands from the frame memory into registers of the execution stage and store results back to the frame memory before activating a new thread. The load and store instructions are not necessary in fine-grain dataflow.

Multithreaded dataflow following the Epsilon-2 approach performs better than the Monsoon technique due to its more powerful instruction set that leads to more efficient machine code. In particular the efficiency is nearly insensible to changes in the number of PEs.

Performance results for the large-grain dataflow techniques depend heavily on the type of processor used as execution stage. The speed can be nearly doubled by a powerful superscalar processor that is able to exploit fine-grain parallelism by its internal hardware structure in combination with an optimizing compiler.

Complex machine operations perform slightly better than large-grain dataflow with a RISC processor. It can be an useful enhancement to other techniques.

Figure 3 shows the elapsed time versus the network delay for a fixed number of 8 PEs and a structure access delay of 1 time step. Network delay shows to be of minor influence except for very high delay values (10000 time steps). This proves that dataflow with any instruction issuing technique effectively compensates network delays by its rapid context switching capabilities. However, if network delay is assumed to be very high, idle times cannot be avoided because load is too low.

Figure 4 shows the elapsed time versus the structure access delay where the number of PEs is 8 and network delay is fixed to 100 time steps. Large-grain dataflow is not considered in this investigation because all data is kept local in the large-grain dataflow model. The results show that structure access delays of up to 1000 time steps per delay are fully compensated by the various dataflow models.

5. Conclusions

We evaluated fine-grain dataflow against different hybrid dataflow techniques with respect to a matrix multiplication program. We measure performance by the elapsed time for the execution of the complete program and not as utilization of the PEs. Performance gains of different architectural approaches are assessed using a single algorithm that is slightly adapted to the specific techniques. This distinguishes our approach from other surveys where several problems are simulated for a single architecture. Results of these surveys cannot be compared to each other.

As results of our simulations we would like to emphasize the following points:

- Split-phase memory access combined with rapid context switching as used in fine-grain dataflow or in hybrid dataflow prove as adequate methods to bridge delays caused by the network or by structure access. This is a strong argument for the use of dataflow techniques because the performance of conventional von Neumann processors either suffers from a high context switching overhead or from idling.
- Large-grain dataflow proved superior to all other techniques provided that a superscalar processor is used, and performs at least equal to the other techniques with a RISC processor. This makes large-grain dataflow a very promising approach since its execution stage can be implemented with off-the-shelf microprocessors and therefore will automatically profit from advances in microprocessor technology.
- Multithreaded dataflow is an improvement over fine-grain dataflow but suffers from a large overhead and thus does not achieve a considerable speedup. This is even worse for the cycle-by-cycle interleaving technique of Monsoon.
- Complex machine operations can be an useful enhancement to other techniques.
- All techniques show good scalability provided that load is sufficient; the Epsilon model of multithreaded dataflow scales best among all regarded techniques when load is low.

Our sample program avoids several problems that may emerge in other classes of algorithms. Therefore it is inevitable to use further algorithms as simulation load that address in particular problems as for instance computing intensive versus I/O bound computations, storage access conflicts, read-before-write-races when implementing non-strict data structures, synchronization loads, load balancing, or network contention.

References

1. Smith, B.J.: A Pipelined, Shared Resource MIMD Computer. Proc. 1978 Int. Conf. Parallel Processing, Bellaire, MI, 1978, pp. 6-8.
2. Dally, W.J., et al.: The J-Machine: A Fine-Grain Concurrent Computer. In: Ritter, G.X. (ed.): Information Processing 89, North-Holland 1989, pp. 1147-1153.
3. Agrawal, A., et al.: APRIL: A Processor Architecture for Multiprocessing. 17th Ann. Int. Symp. Comp. Arch., Seattle, 1990, pp. 104-114.
4. Gurd, J.R., Kirkham, C.C., Watson, I.: The Manchester Prototype Dataflow Computer. Communications of the ACM, Vol. 28, No. 1, January 1985, pp. 34-52.
5. Arvind, Nikhil, R.S.: Executing a Program on the MIT Tagged-Token Dataflow Architecture. IEEE Transactions on Computers, Vol. 39, No.3, March 1990, pp. 300-318.
6. Hiraki, K., Sekiguchi, S., Shimada, T.: Status Report of Sigma-1: A Dataflow Supercomputer. In: Gaudiot, J.-L., Bic, L. (eds.): Advanced Topics in Data-Flow Computing. Prentice Hall, Englewood Cliffs, NJ, 1991, pp. 207-223.
7. Grafe, V.G., Hoch, J.E.: The Epsilon-2 Multiprocessor System. J. Parallel and Distributed Computing 10 (1990) pp. 309-318.
8. Yamaguchi, Y., et al.: An Architectural Design of a Highly Parallel Dataflow Machine. In: Ritter, G.X. (ed.): Information Processing 89, North-Holland 1989, pp. 1155-1160.
9. Papadopoulos, G.M., Culler, D.E.: Monsoon: an Explicit Token-Store Architecture. 17th Ann. Int. Symp. Comp. Arch., Seattle, 1990, pp. 82-91.

10. Evripidou, P., Gaudiot, J.-L.: The USC Decoupled Multilevel Data-Flow Execution Model. In: Gaudiot, J.-L., Bic, L. (eds.): Advanced Topics in Data-Flow Computing, Prentice Hall, Englewood Cliffs, 1991, pp. 347-379.
11. Glueck-Hilltop, E., Ramlow, M., Schuerfeld, U.: The Stollmann Dataflow Machine. In: Odijk, E., Rem, M., Syre, J.-C. (eds.): Proc. PARLE '89, Parallel Architectures and Languages Europe, Eindhoven, 1989, Vol. 1, pp. 433-457.
12. Dai, K., Giloi, W.K.: A Basic Architecture Supporting LGDF Computation. 1990 Int. Conf. Supercomputing, Amsterdam, 1990, pp. 23-33.
13. Zehendner, E., Ungerer, T.: A Large-Grain Dataflow Architecture Utilizing Multiple Levels of Parallelism. 6th Annual European Computer Conference (COMPEURO 92), The Hague, 1992, pp. 23-28.
14. Hiraki, K., Sekiguchi, S., Shimada, T.: Efficient Vector Processing on a Dataflow Supercomputer SIGMA-1. Supercomputing 88, Orlando, 1988, pp. 374-381.
15. Iannucci, R.A.: Toward a Dataflow / von Neumann Hybrid Architecture. 15th Ann. Int. Symp. Comp. Arch., Honolulu, 1988, pp. 131-140.
16. Bic, L.: A Process-Oriented Model for Efficient Execution of Dataflow Programs. J. Parallel and Distributed Computing 8, 12 (Dezember 1990), pp. 42-51.
17. Nikhil, R.S., Papadopoulos, G.M., Arvind: *T: A Multithreaded Massive Parallel Architecture. 19th Ann. Int. Symp. Comp. Arch., Gold Coast, Australia, 1992, pp. 156-167.
18. Sato, M., et al.: Thread-based Programming for the EM-4 Hybrid Dataflow Machine. 19th Ann. Int. Symp. Comp. Arch., Gold Coast, Australia, 1992, pp. 146-155.
19. Ungerer, T., Zehendner, E.: Threads and Subinstruction Level Parallelism in a Dataflow Architecture. Proc. CONPAR 92 - VAPP V, Lyon, France, 1992, pp. 731-736.
20. Oehler, R.R., Groves, R.D.: IBM RISC System/6000 Processor Architecture. IBM Journal of Research and Development 34, 1, January 1990, pp. 23-36.
21. Traub, K.R.: Multi-thread Code Generation for Dataflow Architectures from Non-Strict Programs. In: Hughes, J. (Hrsg.): Functional Programming Languages and Computer Architecture. 5th ACM Conference, Cambridge, Ma., August 1991, pp 73-101.

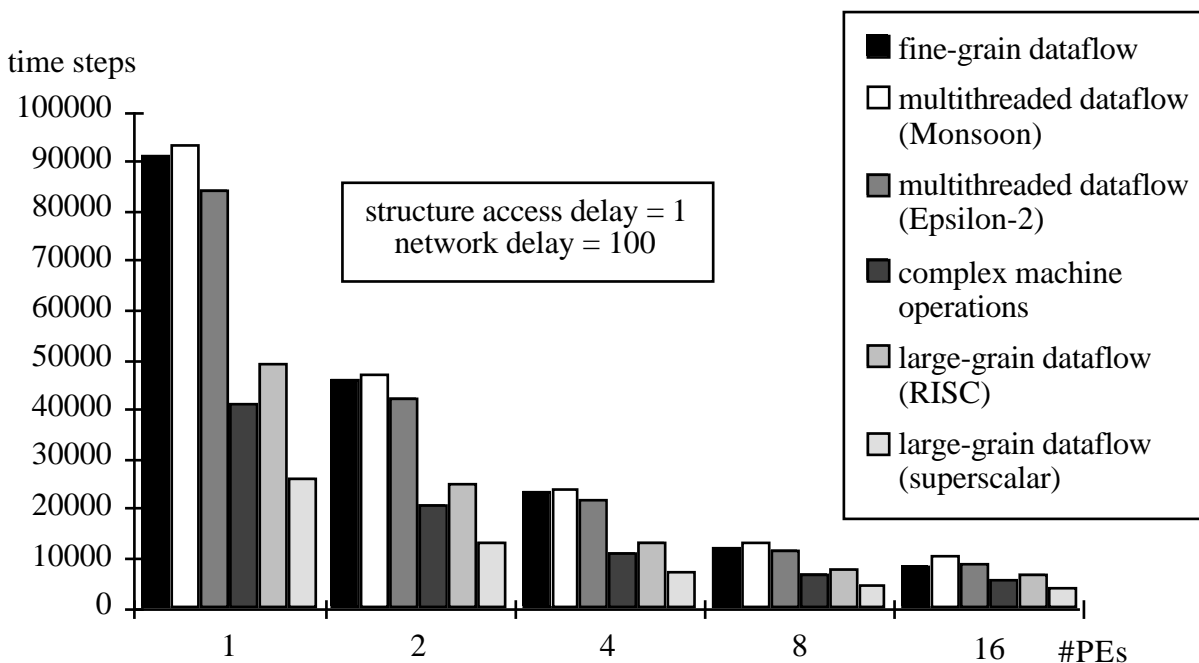


figure 1

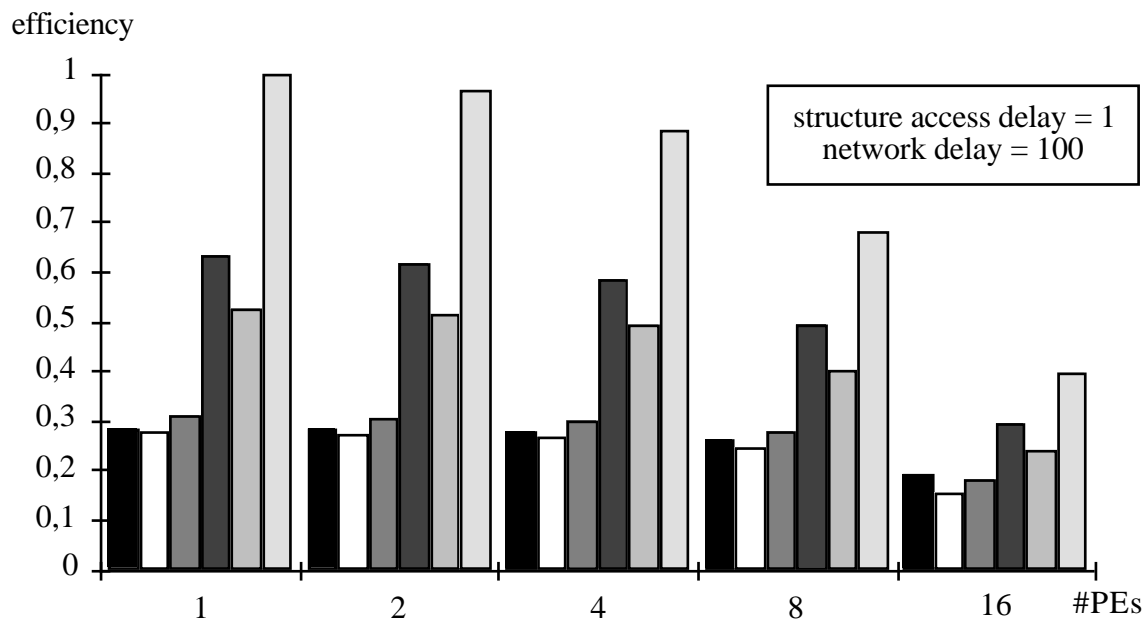


figure 2

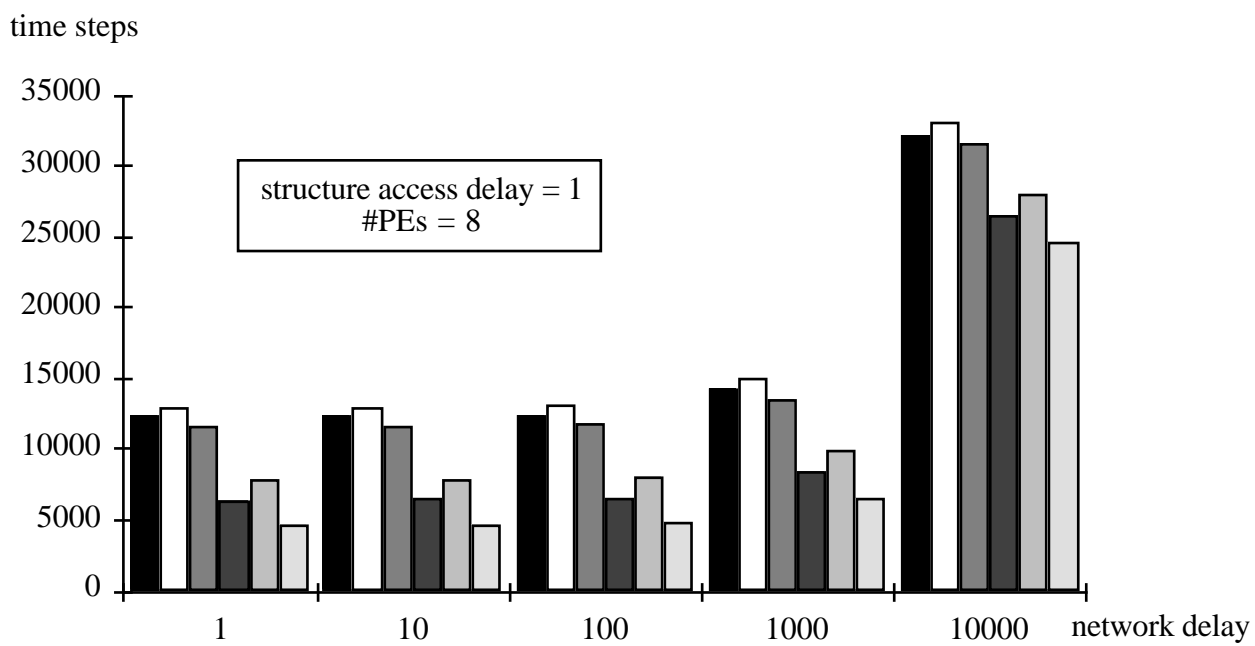


figure 3

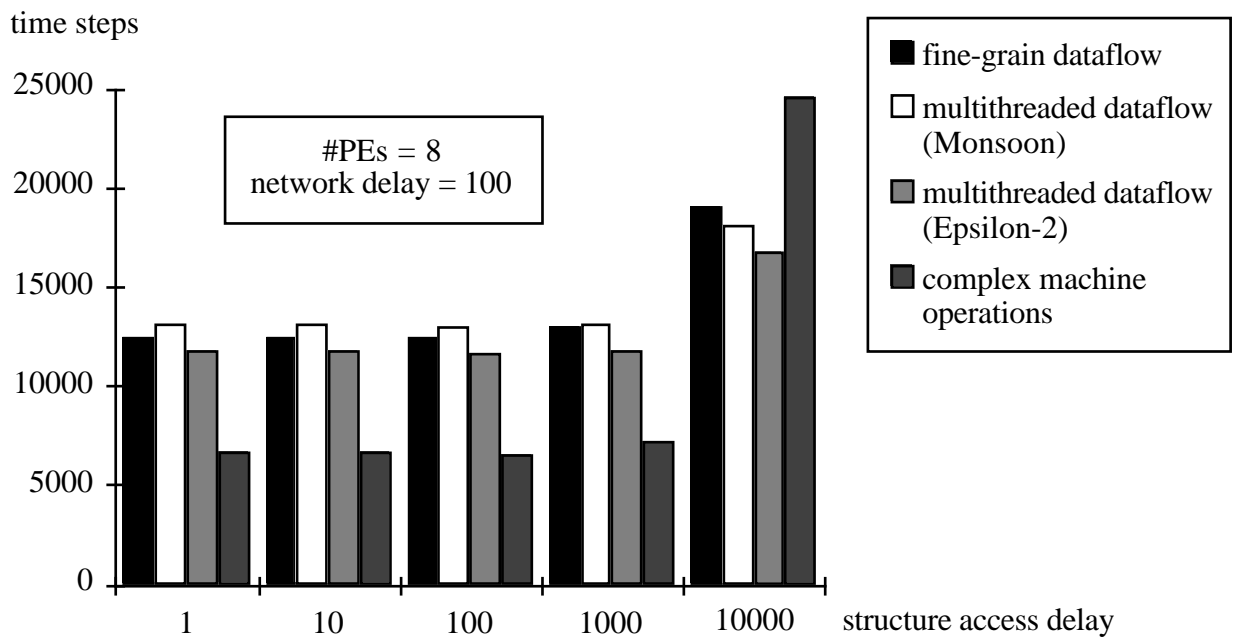


figure 4