Missouri University of Science and Technology
Scholars' Mine

Computer Science Faculty Research & Creative Works

Computer Science

01 Feb 1988

An Inherently Parallel Large Grained Data Flow Environment

Roger E. Eggen

John R. Metzner
Missouri University of Science and Technology

Follow this and additional works at: https://scholarsmine.mst.edu/comsci_facwork

Part of the Computer Sciences Commons

Recommended Citation

R. E. Eggen and J. R. Metzner, "An Inherently Parallel Large Grained Data Flow Environment," *Proceedings of the 1988 ACM 16th Annual Conference on Computer Science, CSC 1988*, pp. 551 - 557, Association for Computing Machinery (ACM), Feb 1988.
The definitive version is available at https://doi.org/10.1145/322609.323134

This Article - Conference proceedings is brought to you for free and open access by Scholars' Mine. It has been accepted for inclusion in Computer Science Faculty Research & Creative Works by an authorized administrator of Scholars' Mine. This work is protected by U. S. Copyright Law. Unauthorized use including reproduction for redistribution requires the permission of the copyright holder. For more information, please contact scholarsmine@mst.edu.

An Inherently Parallel Large Grained Data Flow Environment

Roger E. Eggen
Division of Computer and Information Sciences
University of North Florida
Jacksonville, Florida 32216

John R. Metzner
Department of Computer Science
University of Missouri-Rolla
Rolla, Missouri 65401

Abstract

A parallel programming environment based on data flow is described. Programming in the environment involves use with an interactive graphic editor which facilitates the construction of a program graph consisting of modules, ports, paths and triggers. Parallelism is inherent since data presence allows many modules to execute concurrently. The graph is executed directly without transformation to traditional representations. The environment supports programming at a very high level as opposed to parallelism at the individual instruction level.

Introduction.

Parallel computers are now in existence. These new machines require support software that allow programmers to use them efficiently and without difficulty. Some computer scientists are questioning the practicality of parallel machines since each has a set of unique features that require programmer interaction [2]. The programming environment described is designed to allow easy, efficient construction of parallel programs. Programming in the environment consists of constructing a graph of modules and interconnecting data paths. Program logic becomes apparent due to the pictorial representation. Sequential programming skills are used to construct modules employing top down design and forcing modularity. Parallelism occurs when modules execute concurrently.

Many parallel programming languages concentrate on fine grain parallelism, that is, parallelism at the individual instruction level [3,5,7]. This environment differs by programming with high level verbs that represent modules containing numerous instructions. Modules are system resident at all times, each waiting for data presence to satisfy conditions for activation and if sufficient processors exist, all can be active concurrently. The environment is capable of being implemented on computers with any number of processors. If a single processor exists, one module will execute at a time, however, program development and representation does not change.

A program consists of a network of modules and data paths. The environment is constructed so that programmers can

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specfic permission.

© 1988 ACM 0-89791-260-8/88/0002/0551 $1.50

take advantage of multiprocessing systems without significant retraining. Each program is constructed as a directed graph where modules and paths represent vertices and edges, respectively. The system executes the graph directly, taking advantage of the parallelism expressed. Implementing an algorithm in this environment consists of two distinct tasks: (1) determining required modules and module interconnections; (2) writing the instructions inside the module. Since data flow is the specification of the directed graph, this new exercise is not difficult. At the lowest level, modules are made from available programming languages allowing use of current skills. Since program construction and maintenance is the greatest cost of systems [10], this environment will improve programmer efficiency, with a minimum of retraining, while using new parallel architectures.

> "Parallel programming sounds like a good idea, except that it's impossible or at least too hard to be worth the effort. Parallel computers are becoming increasingly widespread. The question now is what to do with them. Can programming-language researchers make the power of parallelism accessible to programmers?" [2]

The programming environment presented is designed to address the above criticism and questions. This environment provides a good abstraction vehicle for algorithms, which is the basic requirement of programming systems.

What is being offered?

Envision a programming environment composed of a variety of prewritten modules of a general nature to do common tasks such as data division, sorting, searching and merging. An algorithm is constructed by interconnecting modules with paths. The paths indicate the source and destination of data messages.

The computing environment consists of a program of graphs waiting in the system for data presence to allow module activation. Data first entered into the system contains a module destination identifier. Upon data arrival at a module, activation criteria is evaluated. The module begins execution if sufficient processors exist and data presence has satisfied the criteria. Data presence is the sole criteria for determining module activation and execution. Control flow is not used to activate modules.

Many other systems use control flow [8, 6, 4]. These systems vary widely in the degree of expression power. Some offer to express a program as a data flow graph exposing the parallelism involved, but after the parallelism is clear the program is rewritten in a traditional control form language [8]. Other approaches seek to enhance existing languages, such as Fortran, making them become parallel languages [1]. Our approach differs from these in that all program specification is made by modules, ports and paths to represent a large grained highly parallel program. The data flow network is given to the operating system for execution with no further programmer intervention.

Many data flow systems require all paths to contain data before activation of a module is possible [1]. This requirement is circumvented by allowing each module to separate data instances with markers and using activation criteria to determine if the module should be activated.

Many systems require handshaking between modules to pass data, that is, both the sending and receiving modules must be active [4,5]. This requires unnecessary concern by the programmer since both sender and receiver must be considered. Modules can be developed independently in the proposed environment since operating system supported buffers exist to store data at the receiving module.

Information hiding and top down design are natural program methodologies in this environment. The programmer is challenged with specifying the modules and data paths, while the underlying

operating system supports data passage and resource allocation.

A set of modules, such as the merge, sort and count exist within the environment. Data requirements are documented so that program designers use modules without modification. Programming with prewritten modules produces more reliable code in less time since all internal details have previously been considered. Modules that are unique for a given problem are constructed by the programmer. A program graph is specified by the paths interconnecting modules. The code inside the module consists of sequential instructions or a fine grained parallelism. A programmer has two distinct tasks when specifying an algorithm in this environment: the creation of the data flow network and the code associated with modules.

Data Flow Environment - Modules.

A module is a set of instructions performing a task such as sort, merge or other operations required by an algorithm. Each module consists of a number of instructions so that scheduling of parallel operation does not degrade system performance. Information hiding occurs since the module exists within the data flow network and depends only on data paths connected to its ports. Each module containing local nonshared memory executes independently of other modules. A module is in one of two states: idle or active. An idle state indicates that data arrival has not satisfied the modules triggers (activation criteria) and therefore resources have not been assigned to allow activation. A module becomes active when data presence satisfies its trigger and resources are assigned to the module to support execution. Resources include memory for buffer space supporting data passage and processor allocation.

Program networks can be nested so that individual instructions exist in a module only at the lowest level. The merge module in Figure 1 exemplifies this principle. The 4-way merge consists of three 2-way merge modules as shown in Figure 2. The programmer uses existing 2-way merge modules to construct the 4-way module. Again, programmer efficiency is enhanced while using parallelism.

Data Flow System - Ports.

A data flow network graph consists of modules interconnected by paths associated with module's ports. Three basic port types exist: request, input, and output. Supplying modules write data to output ports. The operating system passes data to input ports of receiving modules as indicated by paths. A module contains instructions to write data to an output port. The operating system takes data from the output port and places it in buffers of all paths associated with the port.

Input ports pass data from paths into the module upon command. The data is received from the buffer associated with each input port and each request port. There is no required hand shaking between modules, each module exists as an independent group of executable instructions.

A request port is used to support data flow upon demand. A module requests data causing a port to perform like the output port, to write the request, and an input port, to receive the response.

Data Flow System - Paths.

Multiple paths can be associated with all ports. The module is not concerned with data in paths, as normally encountered in data flow networks, but rather the existence of data at its port. Paths can be added to a port without module modification.

Paths may fan-out from an output port causing data to be alternated between the paths. That is, one data message is placed in each path in a round robin fashion. This technique supports parallelism by replication. Path fan-in and fan-out supports replication and partition algorithm designs. Figure 1

shows an example of a program represented in this environment. Parallelism by replication and partition is shown with the search modules. The document is partitioned by letter and divided to the replicated search modules.

Data Flow System - Triggers.

Associated with each module is a set of triggers that specify the data presence conditions necessary for module activation. Depending on the trigger, data is not required to be present at all input ports before the module can be activated.

Triggers are dynamic. The module can modify its trigger during execution. Commonly, the last instruction, before the module becomes inactive, writes a new trigger to the operating system. When data arrives at a port, the operating system examines the trigger to determine if activation is possible. The trigger is specified by a set of boolean expressions with associated module entry points. The module can perform a variety of tasks warranting multiple entry points.

Data Flow System - Example.

The example (Figure 1) shows construction of the directed graph representing a program to check spelling. Generic modules are being used including distribute, search, keep sorted and merge. Count and strip are inverses of each other; count associates a number with each word and strip removes the number, leaving the document intact. Distribute modules allow parallelism by separating the document, effectively employing the divide and conquer strategy (parallelism by partition). The distribute modules are identical, each receiving half of the document from the path that fanned-out as previously explained. Each distribute module supplies words A-M to port B and N-Z to port C. A search module checks the document against its dictionary, which contains a subset of words to be checked and receives only words associated with that subset. A more efficient search can be employed since the data dictionary in each module is smaller. For example, search 1 and 3 check words beginning with A-M and search 2 and 4 check words N-Z.

When sufficient resources are available, modules can remain continuously active. The module keep sorted, for example, is sorting data as it arrives, providing sorted data immediately, in contrast to traditional operating system sorts, which accumulate data in a file, sorting only upon request.

Data Flow System - Execution.

The execution environment compliments the program development environment. A graphic system for editing and entering programs with windows to allow magnification of a section of the network is required. Upon completion of the network, the program graph is released for execution. The operating system establishes a routing table for data destinations and a table to support a module's trigger. A master portion of the distributed operating system controls the environment. The distributed portion of the system supports individual modules by controlling the buffer, passing data through the port and evaluating triggers. The master section controls the overall system by allocating resources, thereby eliminating deadlock related problems. The network graph is interpreted directly without further programmer intervention.

Acknowledgment.

This work is partially supported by an AMOCO Research Foundation Grant.

Conclusion.

The described environment enhances programmer efficiency and productivity. Parallel algorithms are easily developed without significant retraining since most sequential programming skills are still used. The environment promotes

parallelism without compromising reliability. Each module can be developed individually. Programs are represented graphically, making logic clearer and data dependencies obvious. The dynamic features of triggers allow varying data conditions to activate a module. The environment provides a vehicle for programmers to use parallel computers with a minimum of retraining and simplifies parallel program development. The environment bridges the gap between sequential and parallel program development.

References.

1. Babb II, R.G., Ragsdale, W.C. "A Large-Grain Data Flow Scheduler for Parallel Processing on CYBERPLUS". Proceedings of the 1986 International Conference on Parallel Processing, Aug. 1986, pp. 845-848.

2. Gelernter, D. Guest Editor's Introduction: "Domesticating Parallelism". Computer, Vol. 19, No. 8, Aug. 1986, pp. 12-16.

3. Hillis, D.W., Steele, G.L. Jr. "Data Parallel Algorithms". Communications of the ACM, Vol. 29, No. 12, Dec. 1986, pp. 1170-1183.

4. Hoare, C.A.R. "Communicating Sequential Processes". Communications of the ACM, Vol. 21, No. 8, Aug. 1978, pp. 666-677.

5. Mundle, D.A., Fisher, D.A. "Parallel Processing in Ada". Computer, Vol. 19, No. 8, Aug. 1986, pp. 20-25.

6. Rettberg, R., Thomas, R. "Contention Is No Obstacle To Shared Memory Multiprocessing". Communications of the ACM, Vol. 29, No. 12, Dec. 1986, pp. 1202-1212.

7. Ryder, B.G., Paull, M.C. "Elimination Algorithms for Data Flow Analysis". ACM Computing Surveys, Vol. 18, No. 3, Sept. 1986, pp. 277-316.

8. Standley, H.M. "A Very High Level Language For Large-Grained Data Flow". Fifteenth Annual Computer Science Conference, Feb. 1987, pp. 191-195.

9. Stanfill, C., Kahle, Brewster. "Parallel Free-Text Search On The Connection System". Communications of the ACM, Vol. 29, No. 12, Dec. 1986, pp. 1229-1239.

10. Stevens, W.P. "How Data Flow Can Improve Application Development Productivity". IBM Systems Journal, Vol. 21, No. 2, Feb. 1982, pp. 162-178.

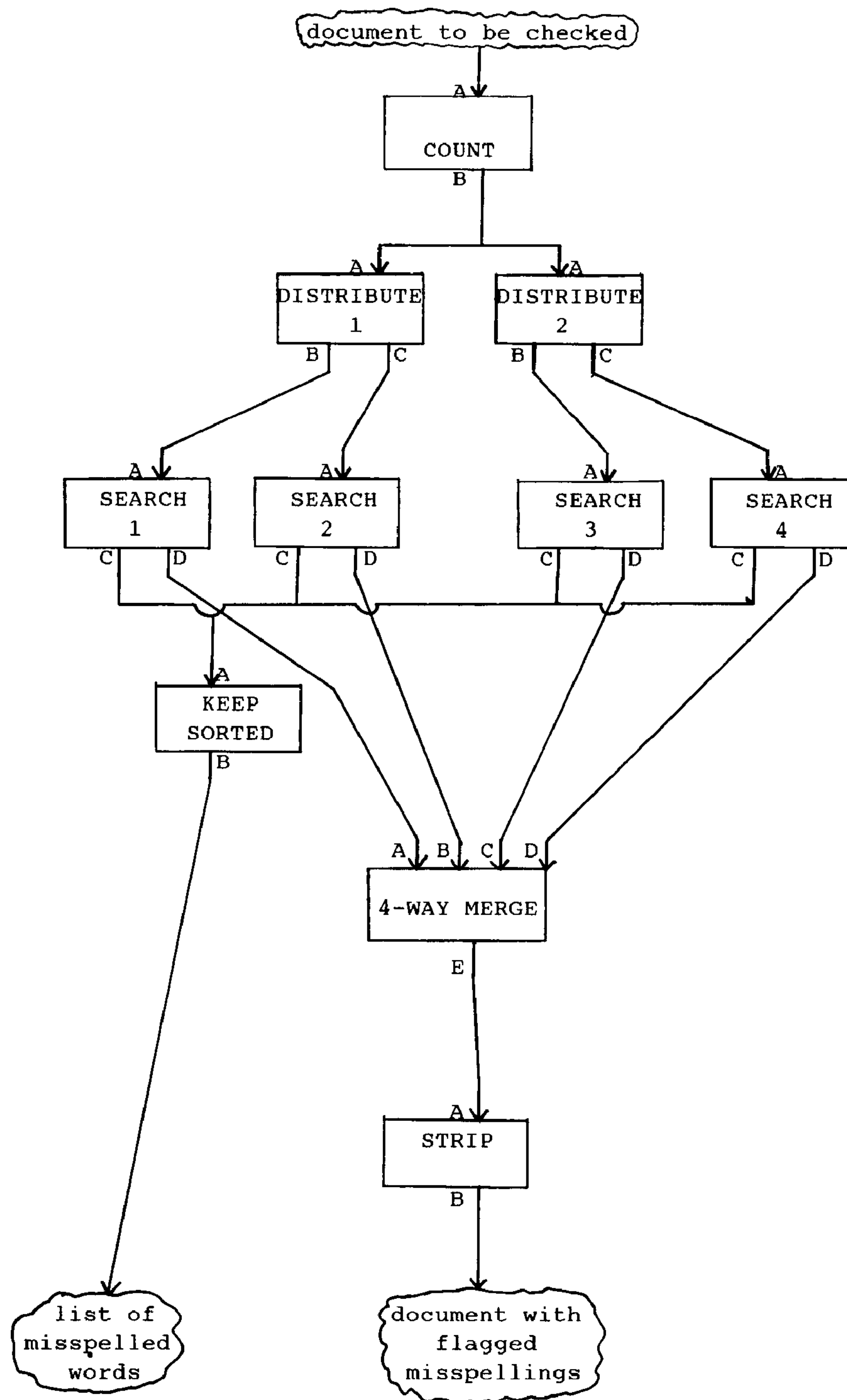
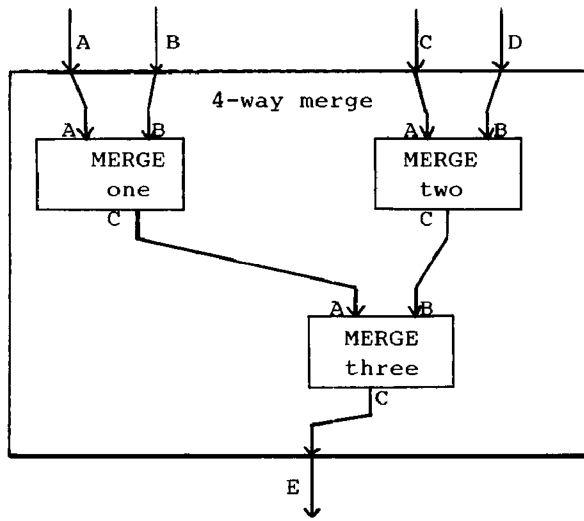


FIGURE 1
SPELLING CHECKER



4-WAY MERGE
Figure 2