TDFL: A TASK-LEVEL DATA FLOW LANGUAGE

Paul A. Suhler*, Jit Biswas, and Kim M. Korner

Department of Computer Sciences
The University of Texas at Austin
Austin, Texas 78712-1188

TR-87-44 November 1987

Abstract

The Task-Level Data Flow Language (TDFL) is a graphical programming language intended for the writing of new programs and the adaptation of existing ones. A computation is represented as a directed graph. As each node contains a subroutine-sized task, this is considered to be coarse-grained parallelism. The task functions are written in standard sequential high-level languages. Two versions have been implemented, one supporting static and one supporting dynamic computation graphs. This paper discusses previous data flow languages, presents the definition of TDFL, describes its implementations on the Sequent Balance shared-memory multiprocessors, and presents several programs and their performance figures.

Keywords: data flow, coarse-grain parallelism, graphical programming

*Department of Electrical and Computer Engineering.

TDFL:
A TASK-LEVEL DATA FLOW LANGUAGE

INTRODUCTION

Various data flow languages have been proposed for parallel programming. The languages differ according to the designers' perceptions of the programmers' needs and of the target machine architectures. The Task-Level Data Flow Language (TDFL) is a data-driven data flow language in which each node contains a function written in C, Fortran, or Pascal. Tokens contain data organized as arbitrary structures. Because it is data-driven and has no central scheduling mechanism, it is easy to implement on a variety of architectures.

The design and implementation of TDFL are intended to achieve several objectives: architecture independence (i.e., the ability to transport a program among different systems without modifying the source code); absence of programmer concern with synchronization, communication, and resource control; ease of use both for writing new programs and for adapting existing programs in conventional languages; and experimentation with different resource control mechanisms. We are pursuing these objectives through a graphical programming environment, in which programs are represented as directed graphs. In this paper, we analyze previous parallel programming languages in terms of these objectives, show how the objectives led to the current design and implementations of TDFL, and present some example programs.

The original version of TDFL was implemented by Nicolas Graner, now of IRIA, on a dual CDC Cyber 170 in 1985 [Graner 86]. The static version of TDFL presented here was implemented by Paul Suhler in 1986 and the dynamic version by Kim Korner in 1987, both on Sequent Balance multiprocessors. A simplified version of static TDFL has been implemented by Rajeev Gulati on an Intel iPSC.

OTHER DATA FLOW LANGUAGES

Some purely textual data flow languages have been designed, some intended for data flow machines (VAL and Id) and some for more general architectures (SISAL [McGraw 83]). Others have been adapted from existing languages for parallel or distributed processing, such as CSP and DP, by adding features to support parallel data-driven execution [Patnaik 86]. These languages all tend to be aimed at small-grained parallelism and all require programmers to learn a wholly new language.

Other language designs have recognized the need to adapt existing sequential language programs and have done so by adding special operations to functions written in standard languages, such as Fortran. Some require the programmer to explicitly perform scheduling or inter-node synchronization, which we regard as an unnecessary burden ([Gokhale 86] and DOMINO [O'Leary 86]). Some, such as Large Grained Data Flow [Babb 86] and Loral Data Graph Language [Kaplan 86] permit the programmer complete generality in defining each node's firing rule (i.e., token consumption and production characteristics). The ease of programming with programmer-defined firing rules versus programming with a fixed set of firing rules, as we advocate, will need future study. Finally, some languages (e.g., LDGL) do not permit nodes to retain state from one firing to the next; this can greatly increase program efficiency and can be done through the use of self-loop arcs, as we explain later.

TDFL DEFINITION

TDFL is a data flow language in which programs are graphs similar to those developed by Karp and Miller [Karp 66]. Each node is an encapsulated function written in a conventional language (in particular, C, Fortran, or Pascal). The node functions contain no operations to perform communications or scheduling. All data is passed to and from the functions as parameters and scheduling of ready-to-fire nodes is handled by the runtime system. There is a fixed set of node categories, each of which operates differently upon its tokens. Each arc of the graph carries tokens of a specified type.

While TDFL is static in that the structure of a graph cannot change at run time, the paths taken by successive tokens flowing through the graph can vary in a data-dependent manner. Furthermore, while all of the tokens on an arc are of the same structure, they may consist of arrays, thus permitting the amount of data to vary between one token and the next, depending upon the computation. This permits the implementation of algorithms in which data sets are partitioned into blocks containing varying numbers of elements.

The encapsulation of standard language functions in data flow nodes simplifies learning TDFL and converting existing sequential programs. Placing inter-node communication and resource control in the runtime system relieves the programmer of those problems and makes the program independent of the architecture of whatever machine it is executing on. The existence of multiple node types and parameterized tokens allows for flexibility in programming.

Execution

To illustrate the execution of a TDFL program, we use an adaptive quadrature program which applies Simpson's Rule to compute the definite integral of some function over an interval. The computation involves repetitively halving the partitions of the interval and computing an approximation of the integral until the approximations converge. This leads to a graph with an input phase, a repetitive calculation, each step of which has parallelism, and an output phase.

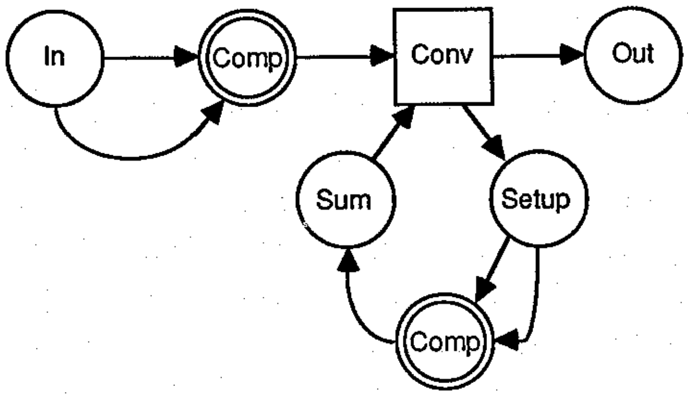


Figure 1. Adaptive Quadrature Program

The program consists of seven nodes of three different types (which will be explained shortly). "In" reads in the interval and directs the next node, "Comp", to compute approximations for partitions of the interval into one and two subintervals. These results are sent to the node marked "Conv", which compares the two approximations for convergence. If the computation has not converged, "Conv" sends the approximations and the interval to "Setup." "Setup" doubles the last number of subintervals and sends an array with one element per subinterval, as well as the number of subintervals, to a second node called "Comp". This "Comp" applies Simpson's rule to each subinterval in parallel to generate a set of approximations and then passes the data to "Sum". "Sum" computes the new overall approximation from the separate approximations and passes it to "Conv", which tests for convergence again. The cycle is repeated with increasingly fine partitions until convergence is achieved, at which time "Conv" sends the results to "Out" for output and then accepts a new interval from "In".

This example illustrates three types of nodes. "In", "Setup", "Sum", and "Out" are General nodes – they are ready to execute when they have at least one token waiting on each input arc. A node with no input arcs, such as "In", is always ready to fire. "Conv" is a Loop node. It is first ready to fire when there is a token on its main input – the arc from "Comp". If the user-defined function in "Conv" returns a value of **repeat** to the runtime system, a token is written onto the feedback output – the arc to "Setup" – and the Loop node will not fire again until a token is received on the feedback input – the arc from "Sum". When convergence is achieved, the function returns a value of **finished** and the next token must come from the main input. Finally, the "Comps" are DoAll nodes, in which multiple processors execute the node's function in parallel, corresponding to a Fortran DO-loop in which all of the iterates are independent. One of the inputs to the DoAll contains the number of iterates to be executed. Like a General node, a DoAll node is ready to fire when it has a token on every input arc.

Multiple waves of data tokens flow through the graph until the first node in the graph decides to terminate execution. As long as "In" has generated valid data, its function must return a value of **alive** to the runtime system, and the token will be given type "Regular". However, when there are no more intervals to process, "In" will return a value of **dead**, the token generated will be of type "EOS" (end-of-sequence), and "In" will die. When other nodes receive EOS tokens, their functions are not executed and they die, generating EOS tokens on all of their outputs. Because each arc handles tokens in a FIFO manner, all Regular tokens will have their data processed before the EOS token is encountered. EOS tokens are handled by the runtime system and the programmer is not involved with testing for them. All that the user code must do is ensure that all nodes with no incoming arcs will return a value of **dead** when they have no more computation to do.

Thus, there are three kinds of information communicated between the node functions and the TDFL runtime system: values in data tokens, special information for the function, and instructions to the system on token handling. The first two kinds are received as function arguments – pointers to the data tokens, pointers to space to be used for the output tokens, and the value of special information, such as the particular iterate number for a DoAll node or the source of a token for a Loop node. Token handling instructions are returned to the system as function values, such as the **alive** or **dead** indication from a General node or the **repeat** or **finished** indication from a Loop node. As the scope of the information available to and the control exerted by a node's function is restricted to the state of that node only, inter-node synchronization and communication remain within the data flow paradigm.

Other Node Types

There are three other types of nodes. The first is the Merge, which fires when there is a token on any input arc. If multiple arcs have tokens, one is selected nondeterministically. This nondeterminism facilitates programming of operating systems and real-time applications.

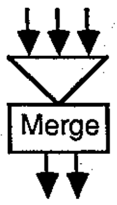


Figure 2. Merge Node

The Case and EndCase nodes are used to obtain parallelism when processing successive tokens in a different manner. In the following example, tokens A, B, and C arrive in order at the Case node. Applying the node's function to A reveals that it requires processing in the second branch from the node. The function copies the token to the second output and returns the value "2" to the runtime system, which writes a token with that value to the special output arc. Similarly, token B goes to the third output and C to the first.

The EndCase node, seeing a token containing a 2 on the control input, will not fire until a token is present on the second input. The EndCase node's function is then applied to that token and the resulting token written to the output. Similarly, tokens B and C will follow A, with the original order preserved by the ordering of the tokens on the control arc. One alternative to the Case-EndCase construct would be a single general node whose function contains a case statement. However, this would not permit the simultaneous processing shown here.

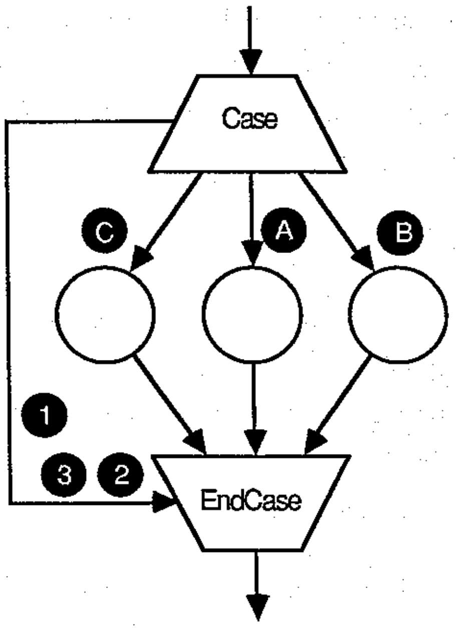


Figure 3. Case-EndCase Node Usage

State Retention

It is frequently useful for a node to retain state from one firing to the next, such as to accumulate information across several firings. As functions' variables do not retain their values from one execution to the next, it must be accomplished with a self loop, an arc going from one node to itself. This is consistent with the data flow paradigm, in which the state of the program resides entirely upon its arcs. At initialization, one token with a null value is automatically placed on every self loop.

Determinacy

With the exclusion of the merge operator, which deliberately introduces nondeterminism, TDFL program graphs are purely deterministic. In other words, every time a program is given the same set of inputs on each input arc of a program graph, irrespective of the order of scheduling or the relative speeds of execution of the tasks, the outputs will be the same. This observation directly follows a theorem of Kahn [Kahn 74], which states that as long as each task in a computation graph behaves as a function on streams, the computation graph in its totality behaves as a function on streams. This property of the language is very convenient for the task scheduler, since all it must do is ensure that tokens on an arc are delivered in the proper order (i.e., as a stream). As we shall see, the scheduler is quite simple.

Dynamic TDFL

In the course of developing TDFL programs, it became obvious that some programs are inefficient when forced into the mold of fixed-size tokens and fixed graphs. Simulation, logic programming, and recursive algorithms are examples of areas better implemented with graphs that can change at runtime. As an alternative, we have developed and implemented a dynamic version of the language which permits creation and destruction of nodes and arcs.

From the user's perspective, Dynamic TDFL (DTDFL) is static TDFL extended by the provision of runtime routines to determine the current topology of the computation graph, determine arc and task characteristics (arc token size, tokens currently buffered by an arc, node type, user code body, etc.), and create new task nodes and arcs. Care has been taken not to involve the user in scheduling and synchronization abstractions – only the topology and structural characteristics of the computation graph are accessible through the runtime routines.

In order to preserve this abstraction, users are not allowed to explicitly destroy nodes and arcs (to do so would require involvement in synchronization, i.e., one node's being aware of the status of another). Instead, the EOS token (previously described) is used to kill nodes. An enhancement has been made in that nodes no longer only return only an **alive** or **dead** status – a node may now return a **killing** status in which it remains alive while propagating EOS tokens on all its output arcs. Dynamic TDFL has remained upward compatible with static TDFL; existing static TDFL programs will execute under DTDFL. Furthermore, the property of determinacy continues to hold for programs not using Merge nodes.

Dynamic TDFL is expected to prove useful in parallel simulation and dynamic functional and logic programming applications. Wave front approaches also lend themselves to parallelization of matrix computations [O'Leary 85]. In such approaches, a matrix of task nodes is created and wavefronts of calculations and data propagate through the matrix. In static TDFL, a different computation graph would have to be compiled for each uniquely sized data matrix. In dynamic TDFL, a root node can initiate creation of an appropriately sized matrix, flow initial data values into the matrix, propagate the calculation wave fronts, and at their completion, destroy the matrix (with EOS tokens). This can be repeated as often as desired in the same computation sequence for new data sets. This is illustrated in Figure 4.

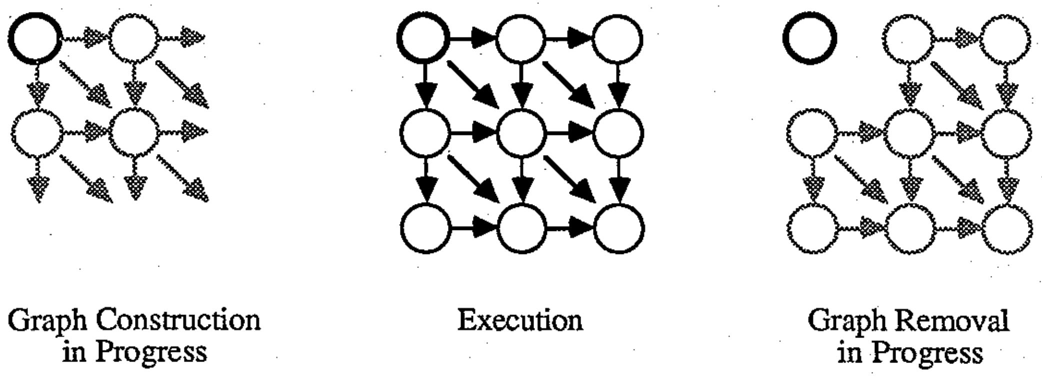


Figure 4. Wave Front Program

IMPLEMENTATION

There were a number of objectives of the implementation which guided the choices made in developing the TDFL runtime system. In approximate order of importance they were: ease of use by application programmers, speed of execution, minimal memory usage, use of different work managers (processor allocation mechanisms), and production of useful performance information. In this section, we describe the target system and tell how TDFL programs are specified. We then discuss our task execution mechanism (microscheduling) and show how the data flow firing rules are implemented by our communication and node scheduling mechanisms.

Target System

The Balance is a shared memory multiprocessor using NS32032 microprocessors with floating point coprocessors. All memory is shared among all processors and is accessed via a 32-bit wide bus. Each processor has a write-through cache with inter-cache consistency maintained through bus watching. The system has two hardware synchronization mechanisms. The System Link and Interrupt Control bus is used to distribute interrupts to processors and to enforce mutual exclusion on certain operating system operations; such a synchronization operation requires about fifty microseconds. Higher-speed synchronization for the user is obtained through a set of hardware locks implementing an atomic test-and-set operation. System calls use these locks to implement spinlocks, semaphores, and barriers. The Dynix™ operating system is an implementation of 4.2BSD and AT&T System V Unix™ and obtains concurrency at the process level.

Program Specification

Eventually, users will program in TDFL using a graphical interface. Arcs and nodes will be drawn on a workstation screen and windows opened to type in specifications for tokens and high level language code for task bodies. At the moment, however, the user must prepare several text files using an editor. These specify node behavior, graph connectivity, token structures, files to be accessed, and task body code.

Task Execution

We considered two different approaches for the Balance implementation of TDFL. The first was to create one Dynix process for each node in a TDFL program graph and to perform communication using sockets. While this would have permitted Dynix to completely handle resource allocation and would have made transporting TDFL to other Unix-based systems easier, there were several disadvantages, such as the high overhead in process switching, the complexity of connecting an arbitrary graph with sockets, and the difficulty of experimenting with different scheduling mechanisms.

We chose a different technique, known as microscheduling, in which a fixed set of identical processes are used, with one process per processor. The functions representing task bodies are linked with the runtime system code and a single copy of the resulting code text is used by all processes. The Dynix operating system then places each process on a different processor, where it can remain for the duration of the program's execution. Task state and scheduling information (such as a ready queue) are saved in shared memory and are available to all processes, with mutual exclusion for consistency enforced using locks. Processes not executing tasks perform busy waits on the ready task queue; caching ensures that this generates main memory references only when the queue changes.

The following figure shows microscheduling schematically with a set of seven processors executing a ten-node program. Processors A, B, and C have acquired and are executing one task each. Processor D has obtained exclusive access to the ready task queue and is about to remove the first task for execution. The remaining processors are waiting to access the queue. Four tasks are neither on the queue nor in execution because they are waiting for their firing rules to be satisfied. The single queue is a point of serialization and reduces the parallelism which might otherwise be obtained. We are experimenting with other work management structures which allow concurrent access by multiple processors; the runtime system was designed to permit easy substitution of different work managers.

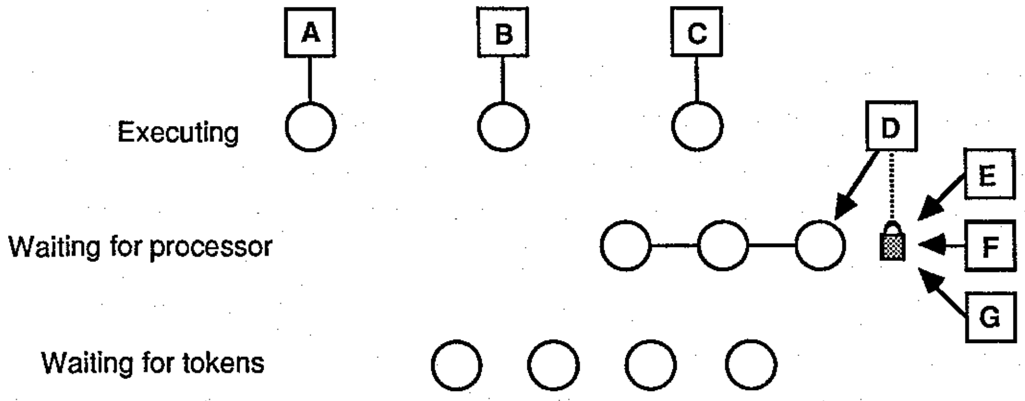


Figure 5. Microscheduling

Microscheduling has the advantages of reducing the amount of memory consumed, reducing the overall task switch time, permitting us to have nearly complete control over processor allocation, and avoiding the problem of performing socket connections in general graphs with large numbers of arcs and a small number of available sockets. It also permits easy changing of the number of processors, as the number can simply be an argument to the command invoking the runtime system.

Scheduling

A node is placed on the ready queue when its firing rule is satisfied – all of its required input tokens are available, each output arc has room to receive at least one token, and the node is not already on the queue. To ensure correctness, nodes are checked for readiness to run by any processor modifying their tokens. Whenever a token is written to an arc, the receiving node is checked by the writer. Similarly, whenever a token is removed from an arc, the reader checks the sending node. Finally, whenever a node completes execution, the processor that executed it checks to see if it can run again. By testing upon every potential change to a node's state, the system ensures that a ready task will always be scheduled for execution.

Data Structures

In static TDFL, all buffers for arcs and control blocks for tasks are allocated at initialization and remain until program termination, whether in use or not. Implementing dynamic TDFL involved conversion of most of the data structures to dynamically modifiable forms. All dead nodes and their input arcs are removed from system data structures and garbage collected.

EXAMPLE PROGRAMS

As a means of testing the usefulness of coarse-grained data flow as a programming paradigm, we have converted a number of existing application programs into TDFL and have completely reprogrammed other applications. Some of this has been done by students in a parallel programming class. We are interested in determining how dependencies in computations affect partitioning their data sets into sequences of tokens, how intuitive the data flow paradigm is, and whether significant sections of existing codes can be reused or whether programs must be completely rewritten. These programs can then become test cases for experiments in resource allocation and performance tuning. The static TDFL examples shown in Figure 6 and described below are not intended to represent a complete spectrum of possible application programs.

Particle Orbit Code

The particle orbit code (POC), is a Fortran program that has been run extensively on supercomputers (e.g., Cray and CDC). It solves a system of differential equations giving the position of particles in a tokamak containing fluctuations. At the heart of the program is a differential equation solver that computes the final coordinates of a particle given initial conditions and system parameters. A large sample of particles is integrated for many oscillation periods. The calculations for individual particles are independent and thus can proceed in parallel. In analyzing the POC, we observed that the program breaks up naturally into three independent modules: a control module that generates the initial and final values for each time step of the integration, a module that performs the integration over a vector of particle coordinates, and a module that computes the final statistics.

A* Heuristic Search

This program follows the same form of the adaptive quadrature program – repeated parallel processing of a data set until a goal is achieved. In this case, the parallel processing is the computation of all successors of the current node in a search of a graph. The graph used in this example had thirty nodes.

Minimal Spanning Tree

This is an implementation of the Prim-Dijkstra MST algorithm. The computation inside the loop has three phases: a parallel generation of prospective next arcs, a serial selection of one arc for addition to the tree, and a parallel updating of the tree. The tree processed in this example had eighty nodes.

Odd-Even Transposition Sort

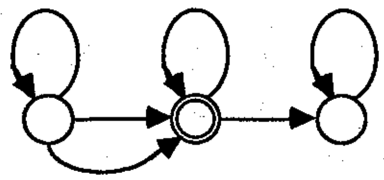
This program begins with a partition of an array of numbers. The first DoAll node performs the odd-even phase of the sort and the second performs the even-odd phase. Each time around the loop, adjacent partitions are merged pairwise and the size of each block doubles. We will present performance figures for sorting 1000 integers.

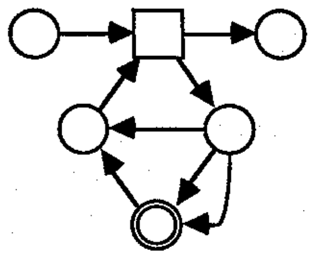
Triangular Matrix Solver

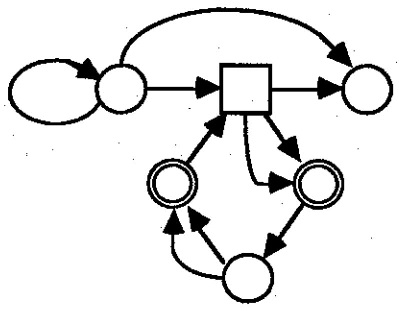
This program solves a system $Ax = b$ of linear equations by back substitution (A is a lower triangular matrix – all superdiagonal elements are zero). The matrix is broken into a three-by-three grid of blocks of size $n/3$ x $n/3$. One node reads in the blocks of matrix A and distributes them to the appropriate arcs; a second node does similarly for constant vector b. Another node writes out the resulting blocks of vector x. The actual solution uses two different task bodies, one for the blocks on the diagonal and another for the subdiagonal blocks. Performance is described for a 128 x 128 matrix of 64-bit reals.

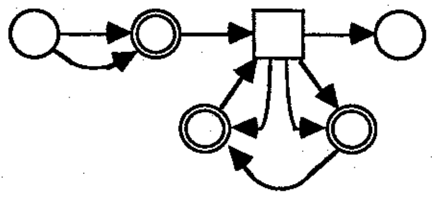
Form4P

Form4P is a quantum dynamics program for computing the motions of diatomic molecules. While the TDFL version is still in development, it is included here to illustrate the language's utility in programming applications with irregular structure.


Particle Orbit Code


A* Heuristic Search


Minimal Spanning Tree
(Prim-Dijkstra)


Odd-Even Transposition Sort

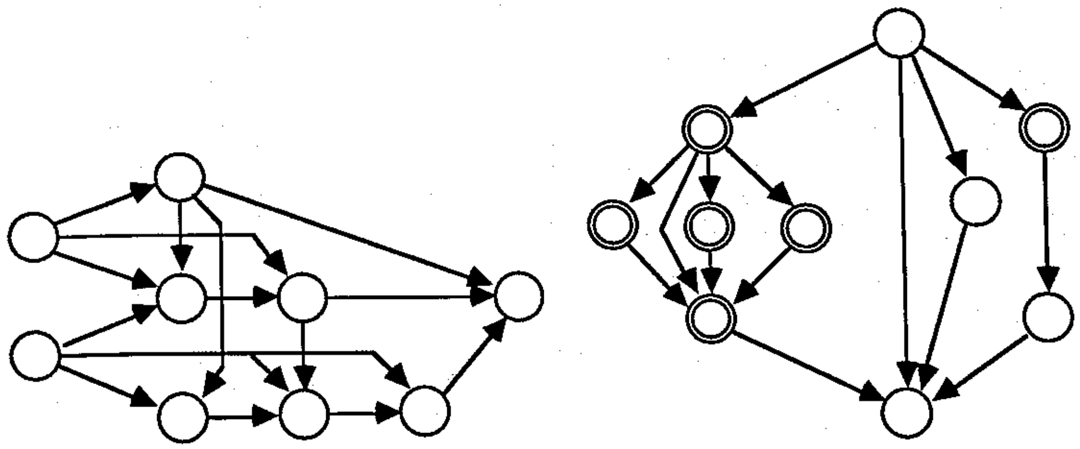


Figure 6. Static TDFL Programs

Performance

Table 1 shows the execution times obtained for each program by varying the numbers of processors used. These execution times yield the speedups shown in Figure 7. In no case did the source code require recompilation when the number of processors changed. The most successful program was the particle orbit code, which did a great deal of computation in each iterate of its DoAll node. The steps in its speedup curve reflect the granularity of the computation. A sequential Fortran version of the POC took 152 seconds, 1 second longer than the TDFL version using one processor. This counterintuitive result is due to the Fortran runtime system's taking longer to start up than does the C language system under which TDFL runs. This difference is slightly greater than the TDFL initialization and internode communication overhead. The total times spent executing the Fortran functions were equal to within a fraction of a second.

The other programs had smaller granularity, resulting in a higher proportion of overhead to useful computation. In particular, the programs using Loop nodes were not constructed to minimize the transmission of data around the loop, resulting in copying of whole data sets with each firing. Finally, the dependencies in the matrix solver's graph never allowed more than two nodes (the subdiagonal ones) to execute at the same time; decomposing the array into more nodes would increase the parallelism. In many cases these programs were the first attempts of students at parallel programming, and the inefficiencies are not surprising. Typically, the most parallelism was obtained through the use of DoAll nodes.

Program \ Processors	1	2	3	4	5	6	7	8
Particle orbit code	151.	77.	57.	40.	39.	30.	30.	21.
Adaptive quadrature	0.95	0.53	0.41	0.34	0.31	0.29	0.27	0.27
Transposition sort	3.39	2.04	1.60	1.38	1.24	1.18	1.13	1.08
Minimal spanning tree	129.	106.	98.	94.	93.	93.	93.	90.
A* heuristic search	1.91	1.51	1.40	1.36	1.35	1.36	1.38	1.39
Triangular matrix solver	7.24	6.25	6.25	6.25	6.25	6.25	6.25	6.25

Table 1
Program Execution Times

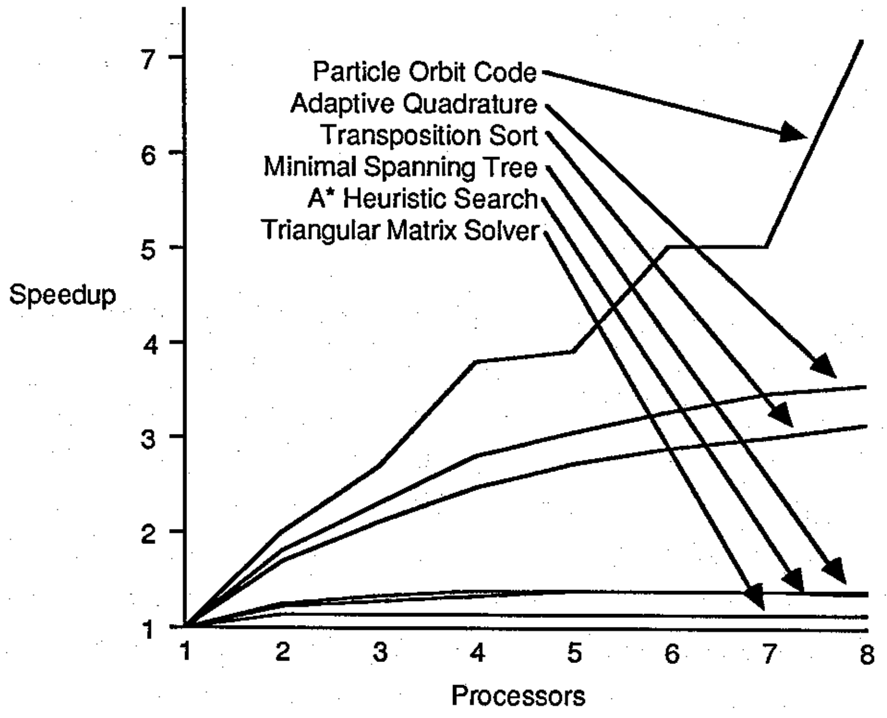


Figure 7. Program Speedups

Program Conversion

In converting the particle orbit code to TDFL we demonstrated the feasibility of converting an existing program from standard Fortran to TDFL. The conversion took more than three weeks; however, most of this time was spent in learning the application program and debugging errors in the TDFL runtime system. It is expected that a programmer more familiar with his/her particular code would be able to do the conversion faster. The portion of the code that was doing the control had to be modified considerably in order to fit it into the data flow framework. However, the smaller subroutines that did the actual integration were left virtually untouched.

SUMMARY

TDFL presents the programmer with a complete language in which he must cast his algorithm in a data flow paradigm, with the data set broken into sequences of tokens. He can then write the computation node task bodies in a standard sequential language without worrying about special scheduling, synchronization, or communication operations. The language provides a fixed set of node types with different firing rules. The static version of TDFL requires that a program be expressed as a fixed graph, while dynamic TDFL permits the programmer to create and destroy new sections of the graph at runtime.

The Sequent Balance multiprocessor implementations of TDFL and DTDFL use one identical Unix-style process per processor to execute the nodes. Node states and data tokens are kept in shared memory. To support program development and experimentation, the TDFL system includes an execution monitor and a single-step execution mode.

A number of programs have been written in static TDFL; some are adaptations of existing programs and some are new implementations of well-known algorithms. The speedups achieved with these programs depend upon the ability of the programmer to structure the code and data to keep the amount of overhead low relative to the amount of useful computation.

Future directions. A graphical front-end for TDFL is being developed for the Sun workstation, to permit programmers to make more direct use of the graphical nature of the language. TDFL has been implemented on the Intel iPSC and studies of program portability have begun. The language is also serving as a testbed for experiments with parallel data structures for work management and with program performance tuning through coordinated scheduling and program-to-architecture mapping.

Acknowledgements. A number of the language's features of have been suggested by our associates and visitors, such as Greg Pfister of IBM (DoAll nodes), Randall Dow, formerly of Sequent, (microtasking), and Clement Leung. Chris Erickson of Sequent has been very helpful in explaining the workings of Dynix. The particle orbit code was provided by and adapted with the assistance of Wendell Horton and Lee Leonard of the University of Texas Department of Physics. Other programs were written by Ashok Adiga, David Roch, James Knight, Kiran Somalwar, and V. N. Rao.

We are grateful to J. C. Browne and Geoffrey Brown for their comments on this paper.

This work was supported by the Cockrell Centennial Chair in Engineering and by grants from Control Data Corporation (4502-0012585), Department of Energy (DE-FG05-85ER25010), Office of Naval Research (N00014-86-K-0763), and DARPA (N00038-86-C-0167).

REFERENCES

[Babb 86] Babb, R. G., L. Storc, and W. C. Ragsdale, "A Large-Grain Data Flow Scheduler for Parallel Processing on CYBERPLUS," *Proceedings of the 1986 International Conference on Parallel Programming*, 19 - 22 August 1986, pp. 845 - 848.

[Browne 85] Browne, J. C., "Formulation and Programming of Parallel Computations: A Unified Approach," *Proceedings of the 1985 International Conference on Parallel Programming*, 20 - 23 August 1985, pp. 624 - 631.

[Edler 85] Edler, J., A. Gottlieb, C. P. Kruskal, K. P. McAuliffe, L. Rudolph, M. Snir, P. J. Teller, and J. Wilson, "Issues Related to MIMD Shared-memory Computers: the NYU Ultracomputer Approach," *Proceedings of the 12th Annual International Symposium on Computer Architecture*, 17 - 19 June 1985, pp. 126 - 135.

[Gokhale 86] Gokhale, M.B., "Macro vs. Micro Data Flow: A Programming Example," *Proceedings of the 1986 International Conference on Parallel Programming*, 19 - 22 August 1986, pp. 849 - 852.

[Graner 86] Graner, N. and J. Biswas, "User Reference Manual for Task Level Data Flow Language: Version 1," TR 86-05, Department of Computer Sciences, University of Texas at Austin, January 1986.

[Kahn 74] Kahn, G., "The Semantics of a Simple Language for Parallel Programming," *Proceedings of IFIP Congress 74*, North-Holland Publishing Co., 1974, pp. 471 - 475.

[Kaplan 86] Kaplan, Ian, "The LDF 100 Data Graph Language," Loral Instrumentation, 11 December 1986, San Diego, CA.

[Karp 66] Karp, R. M., and R. E. Miller, "Properties of a Model for Parallel Computations: Determinacy, Termination, Queuing," *SIAM Journal of Applied Mathematics*, Vol. 14, No. 6, November 1966.

[McGraw 83] McGraw, *et al.*, "SISAL: Streams and Iteration in a Single-Assignment Language. Language Reference Manual Version 1.1," 20 July 1983, Lawrence Livermore National Laboratory Technical Report M-146.

[O'Leary 85] O'Leary, D. P., and G.W. Stewart, "Data-Flow Algorithms for Parallel Matrix Computations," *Communications of the ACM*, August 1985, Vol. 28, No. 8, pp. 840-853.

[O'Leary 86] O'Leary, D. P., G. W. Stewart, and R. van de Geijn, "DOMINO: A Message Passing Environment for Parallel Computation," TR-1648, University of Maryland, April 1986.

[Patnaik 86] Patnaik, L. M., and J. Basu, "Two Tools for Interprocess Communication in Distributed Data-Flow Systems," *The Computer Journal*, Vol. 29, No. 6, 1986, pp. 506 - 521.