Research Monographs in
Parallel and Distributed Computing

GREGORY M. PAPADOPOULOS

Implementation of a General-Purpose Dataflow Multiprocessor

RESEARCH MONOGRAPHS IN PARALLEL AND DISTRIBUTED COMPUTING

Gregory M. Papadopoulos
MIT Laboratory for Computer Science

Implementation of a General-Purpose Dataflow Multiprocessor

Pitman, London

The MIT Press, Cambridge, Massachusetts

PITMAN PUBLISHING
128 Long Acre, London WC2E 9AN

First published 1991

Available in the Western Hemisphere and Israel from
The MIT Press
Cambridge, Massachusetts (and London, England)

ISSN 0953-7767

British Library Cataloguing in Publication Data
Papadopoulos, Gregory M.
Implementation of a general-purpose dataflow multiprocessor.—(Research monographs in parallel and distributed computing : ISSN 0953-7767
1. Computer systems. Multiprocessors
I. Title II. Series
004.35

ISBN 0-273-08835-1

Library of Congress Cataloging-in-Publication Data
Papadopoulos, Gregory Michael.
Implementation of a general purpose dataflow multiprocessor / Gregory Michael Papadopoulos.—1st MIT Press ed.
p. cm.—(Research monographs in parallel and distributed computing)
ISBN 0-262-66069-5 (pbk.)
1. Multiprocessors. 2. Computer architecture. 3. Parallel processing (Electronic computers) I. Title. II. Series.
QA76.5.P29 1990
004′.35—dc20

Reproduced and printed by photolithography
in Great Britain by Biddles Ltd, Guildford

Contents

List of Figures

To my parents,
Imogen and Michael Papadopoulos

1 General Purpose Multiprocessing

Across the diverse range of multiprocessor architectures, from small collections of supercomputers to thousands of synchronous single-bit processors, all seem to share one undesirable property: they are hard to use. Programming has taken a giant step *backwards*. The application writer must consider detailed and hard-to-measure interactions of the program with the machine; the resulting codes are difficult to debug [37], are of questionable reliability [46], and are far from portable [31]. Even after arduous work in converting an application for parallel execution, the actual improvement in performance is frequently disappointing. Perhaps we do not yet understand how to express parallelism in a way that is machine independent. Perhaps we need more sophisticated compilers and associated debuggers to better exploit a machine's parallelism. Perhaps we are building the **wrong** machines.

It is a pervasive belief that our lack of real success in general purpose multiprocessing is a software problem. Machine architects adapt sequential processors to the parallel setting by providing an interprocessor communication medium and an *ad hoc* set of synchronization mechanisms, like locks or fetch-and-add. Then the compiler, or worse yet the programmer, is expected to partition the application into tasks that can run in parallel using the supplied synchronization primitives to ensure deterministic behavior. While we certainly share the belief that there **is** a software problem, we are convinced that there are equally serious defects in the underlying machines. There needs to be a fundamental change in processor architecture before we can expect significant progress to be made in the use of multiprocessors. The nature of this change is the deep integration of synchronization into the processor instruction set [12]. The instruction set must constitute a **parallel machine language**, in which parallel activities are coordinated as efficiently as instructions are scheduled.

The instruction set of a dataflow machine [6] forms such a parallel machine language. An instruction is executed only when all of its required operands are available. Thus, low-level synchronization is performed for every instruction and at the same rate as instructions are issued. It is easy for a compiler to use this fine grain synchronization to produce code which is highly parallel but deterministic for any runtime mapping of instructions onto processors. While there is consensus that dataflow does expose the maximum amount of parallelism, there is considerable debate surrounding efficiency of the execution mechanism. This criticism centers on three points: (1) the number of instructions executed, (2) the relative power of a dataflow instruction and (3) the cost and complexity of a data driven processor.

Recently there has been significant progress in compiling scientific codes for dataflow machines [7]. Substantial programs written in the high-level language Id [41][40] and compiled for a dataflow machine yield dynamic instruction mixes (*e.g.* percentage of floating point operations) that are nearly equivalent to the same algorithms compiled from FOR-

TRAN and executing on a reduced-instruction set sequential uniprocessor [20] [8]. Moreover, the dataflow program executes essentially the same number of instructions independent of the number of processors, whereas the parallelization of a program for a conventional multiprocessor invariably incurs non-trivial execution overhead (*e.g.* synchronizing through barriers, task creation [8]) and typically yields *far less* parallel activity.

But how are we to compare the cost, in terms of processor complexity, of executing a dataflow instruction versus executing an instruction from a sequential stream? First, the operation performed by a dataflow instruction is similar in power to an operation on a conventional load/store machine, *i.e.* `ADD`, `MULT`, `LOAD`, `STORE`, `BRANCH` *etc.*[1] The difference lies in the way instructions are scheduled. In the von Neumann model, the operands for an instruction are assumed to be available when the program counter points to the instruction. In a dataflow machine an instruction is *asynchronously* scheduled only when its operands have been produced by some other parallel activities, so a dataflow machine must have an **operand matching** mechanism for detecting when an instruction has its required operands. Several general purpose dataflow machines have been built (*e.g.* ETL Sigma-1 [26], Manchester Machine [22]) or extensively simulated (*e.g.* M.I.T. TTDA [14]). But it is clear that these machines are far from commercial practicality[2]. A primary reason for this, and a key criticism of dataflow machines, is the complexity of the operand matching mechanism [21].

Our goal is to discover implementation techniques that improve the cost/performance ratio of dataflow processors. Central to the work presented here is a new approach to operand matching. An **Explicit Token Store (ETS)** machine directly executes dataflow graphs while incorporating a new model of storage. The ETS allows the operand matching storage for the execution of a function invocation to be coalesced into an activation frame which is explicitly managed by the compiler. This enables implementation of the operand matching store with conventional (as opposed to content-addressable) memory technology and permits the realization of well-balanced pipelines.

Although this work focuses on the derivation of the ETS within the realm of dataflow architectures, we take the opportunity by way of this introduction to build a stronger case for machines which support fine grain synchronization. We believe that parallel machine architects eventually will have to apply the same concern for the efficient coordination of parallel activities as they presently do for fast sequential execution within a processor. Only then can we expect significant progress in exploiting parallelism in the general purpose setting.

1.1 Why Fine Grain Synchronization

Most multiprocessors are very bad at managing parallelism. Programmers and compiler writers are painfully aware of this fact. The more finely a program is divided into tasks, the greater the opportunity for parallel execution. However, there is a commensurate increase in the frequency of inter-task communication and associated synchronization. So exposing

[1] Here we are restricting our attention to the instruction set of the M.I.T. Tagged-Token Dataflow Architecture (TTDA).

[2] The ETL Sigma-1 is the best engineered of the group. Presently, a 128 processor 640 MIPS engineering prototype is now operational at the MITI Electro-Technical Laboratory in Japan.

more parallelism, by way of identifying more tasks, does not obviously make the program run faster. In fact, we claim that is largely **unprofitable** to expose most of the latent parallelism in programs unless synchronization and task management are as efficient as the scheduling of primitive operations like add and multiply.

For a given machine, there is a fundamental tradeoff between the amount of parallelism that is profitable to expose and the overhead of synchronization. Sarkar[48] articulates this tradeoff as the competing contributions of the **ideal parallel execution time,** the amount of time required to execute the program in the absence of overhead, versus the **overhead factor**, the extra work required to schedule and coordinate the tasks. The ideal parallel execution time is multiplied by the overhead factor to yield the **actual parallel execution time,** the amount of time required to complete the problem for a given task granularity in the presence of scheduling overhead.

Figure 1.1 illustrates the general characteristic of the parallelism–overhead tradeoff for a typical program running on a machine with ten processors. This plot is suggestive of what was experienced running various programs on contemporary multiprocessors, but it does not express the data from a specific machine or application. The normalized execution time is the ratio n/s where n is the number of processors and s is the actual speedup relative to a single processor. The normalized ideal parallel execution time increases from 1 to n as the task granularity increases from 1, a single instruction, to 100,000 when the entire program executes as a single task. The task overhead factor is given by $(g + o)/g$ where g is the task size in instructions and o is the per-task overhead, in this case 1000 instructions[3].

In this example the actual execution time is a minimum for a task size of about 2,000 instructions yielding a normalized execution time of three — that is, three times slower than the best ideal time. And this plot is optimistic. It is not possible, in general, to partition a given program into tasks of equal size, and the task overhead is a complicated expression that depends upon how the program was partitioned into tasks (*e.g.* the communication overhead component is a function of the number and sizes of data structures shared by two tasks). Thus, achieving optimal performance in the presence of overhead is much more difficult than simply finding the intersection of the ideal execution and overhead factor curves in Figure 1.1.

1.1.1 Tasks Should be Cheap

Look again at Figure 1.1. We think there is something fundamentally wrong — why should the task overhead be so extraordinarily large? Is it an inherent aspect of parallel computation or an artifact of the processor architecture used to construct the multiprocessor? We believe the latter. It seems counterproductive to force a programmer or compiler to expend so much effort working around what amounts to a basic deficiency in the processor architecture. If the task overhead were essentially zero, more parallelism would be exposed and exploited and compiling would be far easier. In fact, the entire partitioning and scheduling problem solved by Sarkar for the functional language SISAL [38] would be moot on a machine that had very low task overhead.

[3]The example employs a per-task overhead of 1,000 instructions which is very *low* as compared to the tasking costs on current commercial multiprocessors and operating systems.

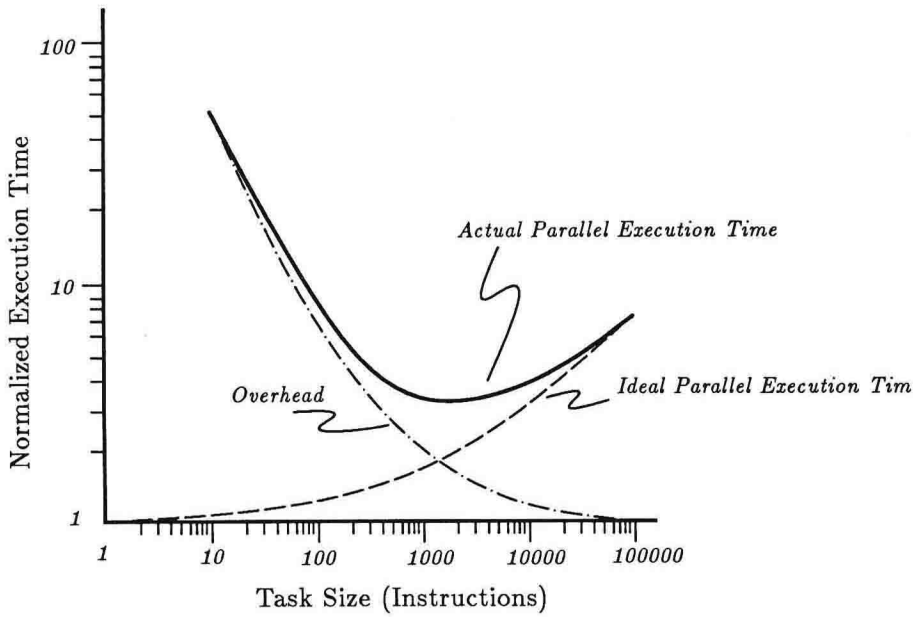

Figure 1.1: The Parallelism–Overhead Tradeoff (from Sarkar)

By "low overhead" we mean on the order of one to ten instructions, basically **orders of magnitude** better than contemporary multiprocessors. We think the parallelism–overhead tradeoff should look like Figure 1.2. In this ideal world, task overhead is not a first order issue. Instead, the objective is to expose the maximum amount of parallelism by subdividing the computation as finely as possible.

1.1.2 A Virtual Memory Analogy

In a way, our argument for hardware support of fine grain synchronization is not unlike the case for architectural changes in order to efficiently implement demand-paged virtual memory. Demand-paging is a convenience for the programmer. It is very tedious for a programmer to manage overlays of code or data, and overlays clash with the semantics of modern programming languages. When data structures get large and access patterns are unpredictable, the programmer using overlays essentially emulates a virtual memory system, with the attendant loss of efficiency — both of the machine and the programmer.

While it is possible to look for the compiler-*forte* that can sift through the access patterns of programs and insert page management code, the problem is so intractable that we have come to insist on hardware support. The essential changes to the hardware are simple to describe, but the effect on the machine architecture is pervasive. Address translation and page fault detection must correctly occur on *every* memory reference, and each instruction that touches memory must be restartable. These two requirements

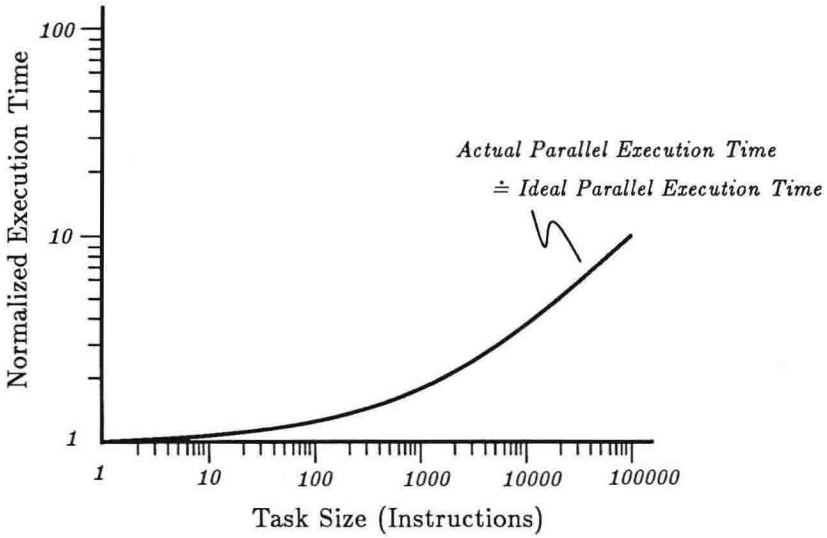

Figure 1.2: Speedup vs. Task Size When Task Overhead is Very Low

affect almost every aspect of the implementation from cache design to the basic instruction interpretation mechanisms.

Why should the management of complex interactions among concurrently executing entities be any *easier?* At least the demand paging analysis need only focus upon the address patterns of a single locus of control, whereas intertask synchronization analysis must discover the *interaction* of addressing patterns across multiple control loci. We think that a parallel machine should provide, at a minimum, real time synchronization checks on every memory load and store.

We also maintain that compiling for a parallel machine is greatly simplified when the task namespace is "virtualized"; that is, the namespace is much larger than the number of physical processors, so tasks are not bound to limited processor resources like multiple register sets.

An analogy for a machine that supports only a small fixed number of contexts (*cf* the Denelcor HEP [50]) would be a virtual memory system where the hardware supports a small number (say, 64) of simultaneous page translations, *but* where issuing an address that is not presently in the translation buffer causes an unrecoverable error (as opposed to a restartable fault). This puts the burden on the compiler to manage the translation buffer, explicitly inserting and deleting entries. It is not clear that this is any easier than performing demand paging on a machine with *no* hardware support. Similarly, a machine that provides a virtualized task namespace relieves the compiler and runtime system of the tough job of managing a small set of active tasks drawn from the large set of *possibly* active tasks.

In the remainder of this chapter we look more carefully at the overhead incurred in the parallel execution of a program. We build an informal model of parallel computation based upon concurrently executing tasks that can communicate through shared memory. While a

conventional processor executes the sequential portion of these tasks very well, it provides little support for moderating the interaction among tasks. Our objective is to sensitize the reader to two of our axioms: (1) a machine should support lots of simultaneous tasks and (2) synchronizing and scheduling these tasks should be very cheap. Not surprisingly, we believe that dataflow machines possess these properties. This work presents a dataflow processor architecture which, we believe, is a simple and appropriate building block for general purpose multiprocessors.

1.1.3 An Informal Task Model of Parallel Computation

We present a model of parallel execution to build our intuition about the cause of high overhead in parallel computation. We do this to motivate architectural support for fine grain synchronization, *not* to put forward an exact, or even complete, performance model for multiprocessors.

We are strongly biased towards programming parallel machines in high level languages which support a dynamic model of storage (*i.e.* a heap). At present, we do not see how modern languages can be effectively compiled for machines that do not directly supply a uniform address space without forsaking the ability to freely reference shared objects from within the language. In **message passing** machines (*e.g.* the Cosmic Cube [49] and Intel iPSC) the only way to share data is for the programmer to explicitly code commands to move data from one processor to another, so references to shared objects are emulated by the programmer and are not part of the machine language[4]. We focus instead upon **uniform address space** machines (*e.g.* the BBN Butterfly [47] , IBM's RP3 [44], Alliant and Cedar [33], Cray-XMP) which provide hardware interpretation of references to addresses which are non-local, meaning there is no requirement for a compile-time distinction between local and non-local storage.

Suppose that we represent an executing parallel program as a **task graph** of inter-dependent sequential tasks. The nodes of the graph represent activations of sequential tasks, the local storage for which will be contained by an **activation frame.** The edges represent inter-task control and data dependences. We further assume that the execution dependences form a tree, although tasks can also communicate through shared objects on a heap for which references are passed as arguments and results. The shape of the task tree and the connectivity of objects on the heap, in general, cannot be determined statically, as we also permit fully recursive application and dynamic allocation of shared objects. For example, consider the following program which initially invokes `f`, which in turn allocates and returns as a result the object `A`, and makes calls to `g` and `h`;

```
def f() =
  { allocate A;
    call g(A);
    call h(A);
    return A };

def g(A) =
  { call g1();
    call g2(A)};

def h(A) =
  { call h1(A)};
```

[4]The driving concern is efficiency. It is certainly possible to *emulate* a shared address space on a message passing machine. The inefficiencies of such a scheme should be obvious, notwithstanding the efforts of intensive compile-time analysis.

We have not yet taken a position on how f is to be computed. Namely, what order is to be imposed on the computation of g and h? There are roughly three possibilities.

1. **Sequential.** The familiar sequential order: f calls g and f suspends; g computes, terminates and then continues f; f calls h and suspends; h computes, terminates and continues f; f terminates. These rules are applied recursively to g, h, g1, g2 and h1. So at any time there is a stack of activation frames, of which only the top one is active.

2. **Fringe Parallel.** The calls to g and h are made in parallel: f forks g and h as independent tasks, and f suspends; g and h compute; when *both* g and h have terminated f is continued (*i.e.* g and h are joined with f); f terminates. These rules are applied recursively to g, h, g1, g2 and h1, but notice that only the leaves of the calling tree are active simultaneously.

3. **Fully Parallel.** A parallel call is made to g and h except that f is not suspended. f terminates after g and h terminate. The recursive application of this rule enables all tasks to run concurrently in the tree of active tasks.

We are not concerned with how the programmer indicates which evaluation method to use, nor whether the programming language semantics are functional or imperative. In either the parallel call or fully parallel case, the task graph unfolds as shown in Figure 1.3. We note that if calls proceed in parallel, there is no way *a priori* to allocate and deallocate local storage for the simultaneous tasks from a stack. In this case, tasks must be given separate activation records which, in general, must be managed more nearly like a heap.

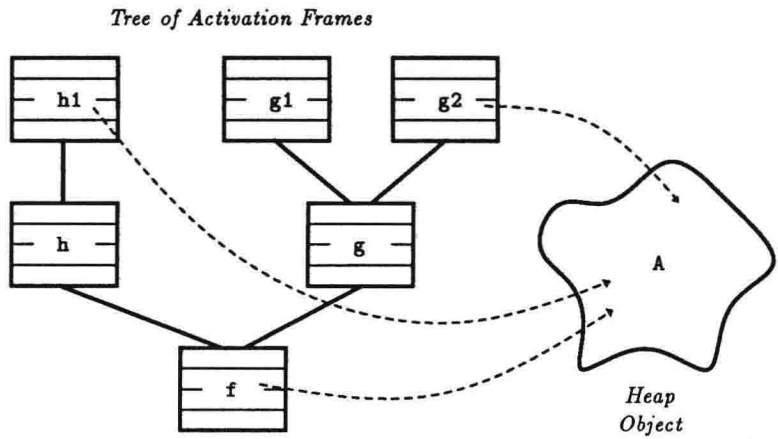

Figure 1.3: Simultaneous Tasks Referencing a Shared Object

The decomposition of the program into parallel tasks is usually accomplished explicitly by the programmer by annotating the source program [31], although functional language (*e.g.* FP [15], SISAL [38], Id [41][40]) programs can be partitioned automatically by the compiler. There is also considerable effort in the semi-automatic partitioning of programs written in sequential languages, notably FORTRAN, into parallel tasks [3][2].

1.1.4 Parallelism and Synchronization

The opportunity for parallel execution arises by noticing that all tasks whose dependences are satisfied can be freely scheduled. A dependence may either be a parameter to a task (*e.g.* the pointer to an object), a control prerequisite (*e.g.* the termination of another task, the acquisition of a resource lock), or dynamic data dependence through the heap (*e.g.* an element of an array).

The determination of the set of executable tasks is fundamentally a problem of **synchronization,** the act of translating the implicit or explicit assertion of a dependence by one task into a decision either to schedule or block another dependent task. The forms of synchronization required to support various dependences found in parallel programs include the following basic operations:

1. **Producer–Consumer.** A task produces a data structure that is read by another task. If the tasks are executed in parallel, synchronization is needed to avoid the **read-before-write** race.

2. **Forks** and **Joins.** A parallel call forks the thread of control into two tasks which is subsequently joined back together.

3. **Mutual Exclusion.** Concurrent procedures may emit requests that must be processed one at a time, *e.g.* updating an object or the serialization in the use of a resource.

All forms of synchronization require the *naming* of a synchronization event. At least one bit of state must be associated with this name, indicating whether the event has occurred or is **pending.** When the event occurs, the synchronization can complete. For producer–consumer synchronization this means that a waited-for value has become valid and the consumer can read it; for a join this means that both threads have reached the join and that one may proceed; for mutual exclusion this means that the exclusive resource has been freed for use by a pending requester. We say that a task is **blocked** when it is waiting for a pending event.

The more finely a program is divided into independent tasks, the greater the opportunity for exploiting parallelism; but there is typically a similar increase in the frequency of synchronization. Each pending synchronization event requires a unique name. Thus, as the number of potentially concurrent activities increase, so does the number of simultaneously named synchronization events. If the total number of concurrent tasks are identified dynamically (*e.g.* parallel execution of nested loops, recursive execution, or even separate compilation) then maximum number of synchronization events is difficult, or impossible, to predict statically. If the synchronization namespace is small, limited for example by the number of registers in a processor, then exposing parallelism is apt to be more difficult as the synchronization namespace must be carefully managed. This requires either static analysis at compile time (tantamount to global register allocation) or fairly expensive runtime (*e.g.* operating system) management.

Aside from how a synchronization point is named, the most important property of an implementation is how the event is *related to* the completion of synchronization. In an **event driven** system, the event directly causes the blocked task(s) to be scheduled,