Evaluation of Mechanisms for Fine-Grained Parallel Programs in the J-Machine and the CM-5

Ellen Spertus†, Seth Copen Goldstein‡, Klaus Erik Schauser‡, Thorsten von Eicken‡,
David E. Culler‡, William J. Dally†

†MIT Artifical Intelligence Laboratory
545 Technology Square
Cambridge, MA 02139
{ellens,billd}@ai.mit.edu

‡Computer Science Division — EECS
University of California
Berkeley, CA 94720
tam@cs.berkeley.edu

Abstract

This paper uses an abstract machine approach to compare the mechanisms of two parallel machines: the J-Machine and the CM-*5. High-level parallel programs are translated by a single optimizing compiler to a fine-grained abstract parallel machine, TAM. A final compilation step is unique to each machine and optimizes for specifics of the architecture. By determining the cost of the primitives and weighting them by their dynamic frequency in parallel programs, we quantify the effectiveness of the following mechanisms individually and in combination. Efficient processor/network coupling proves valuable. Message dispatch is found to be less valuable without atomic operations that allow the scheduling levels to cooperate. Multiple hardware contexts are of small value when the contexts cooperate and the compiler can partition the register set. Tagged memory provides little gain. Finally, the performance of the overall system is strongly influenced by the performance of the memory system and the frequency of control operations.*

Keywords: *Parallel Processing, Performance Analysis, Compilation.*

1 Introduction

Several experimental parallel architectures have been developed in recent years to demonstrate novel hardware mechanisms that may enhance the performance of programs written in emerging parallel languages. For example, Monsoon focuses on Id90, the J-Machine on CST, Alewife on Mul-T, the CM-5 on Fortran90, and Dash and KSR-1 on extensions to C and Fortran. All of these architectures provide a family of mechanisms that collectively support the requirements of the parallel language, are universal enough to support any of the other language paradigms, and are real enough to be constrained by the traditional technology forces. Thus, it would seem that the time has come for parallel architecture research to begin the shift from "big new ideas" to careful quantitative analysis of the effectiveness of various mechanisms. In this paper, we seek to evaluate the set of mechanisms in the MIT J-Machine with respect to the implicitly parallel language Id90 and draw a quantitative comparison with the CM-5.

At the current state of parallel computing, a completely satisfactory quantitative analysis of mechanisms is difficult to achieve because there is no well-established body of machine-independent software reflected in a standard set of benchmarks. There is not even a consensus on the programming languages of choice. Where benchmarks exist, they have been developed specifically for the machine that they are intended to evaluate [14, 9] or specifically avoid emerging languages and the novel mechanisms which could bring them within practical reach [3]. It is also difficult to obtain high-quality compilers for such new languages on more than one machine, yet it is well understood that the architectural support can only be evaluated in the context of sophisticated compilation, rather than direct execution of high-level constructs. Finally, the machines reflect substantially varying engineering budgets and designer capabilities, which should be factored out of the evaluation of the architectural contribution. Simply comparing execution times gives only a crude and noisy calibration, failing to isolate the reasons for the differences.

The method of analysis employed in this paper is as follows. We consider two recent parallel machines: the J-Machine, developed at MIT as a study in universal mechanisms for fine-grained parallelism, and the CM-5, developed at Thinking Machines Corp. as a commercial product supporting data-parallel programs. We take as a basis for comparison a powerful machine-independent parallel language, Id90, which was not the primary target for either architecture, but for which a high-quality compilation sys-

tem exists. The compiler performs a variety of high-level optimizations in translating the language down to code for a simple abstract machine, TAM [6, 13]. The TAM code is identical for the two machines, controlling for effects of high-level optimizations. The translator from TAM code to machine language, however, employs a variety of machine-specific optimizations reflecting the most advantageous use of the available mechanisms. The performance of isolated mechanisms is reflected in the cost of the individual TAM primitives on the machine. The overall effectiveness of the family of mechanisms is determined by weighting each of the primitives by its frequency in a suite of programs. The J-Machine essentially provides direct hardware support for every aspect of TAM; however, TAM does not use all the mechanisms in the machine. The CM-5 provides a variety of mechanisms for data-parallel programming, which are not useful to TAM. What remains is a very reasonable baseline machine, essentially a collection of workstation-class processors on a dedicated network. Thus, we can compare a sophisticated set of mechanisms against a familiar baseline architecture with respect to the dynamic load presented by Id90 programs compiled to TAM.

Section 2 describes the two architectures under study and explains the salient aspects of TAM. TAM-level dynamic instruction frequencies are produced for a variety of programs to serve as a basis for comparison. Section 3 corrects for a set of architectural and engineering factors that have a significant impact on execution time for the two machines, but for which conventional wisdom (and hindsight) applies. The remaining sections deal with architectural aspects that are unique to parallel computing. Section 4 examines the impact of the processor/network coupling on message-passing cost. Section 5 looks at three mechanisms related to asynchronous message arrival that interact with dynamic scheduling. Section 6 considers the utility of tagged memory words and Section 7 ties together our observations. Two important lessons arise from the study. First, novel mechanisms do not substitute for solid engineering of the processor pipeline and storage hierarchy. Second, mechanisms should not be evaluated in isolation, but in how they work together in the compilation framework for the programming language.

2 Background

2.1 CM-5

The CM-5 [16] is a massively-parallel MIMD computer based on the Sparc processor, interconnected in two identical disjoint "hypertree" networks. Each node consists of a 33 MHz Sparc RISC processor chip-set (including FPU, MMU and 64 KByte cache), 8 MBytes of local DRAM memory and a network interface to the hypertrees and broadcast/scan/prefix control networks. (The node may also contain vector units with additional memory, but we will not address the vector capability.) The network interface consists of a pair of memory-mapped FIFO queues for each of the two data networks. Messages are limited to a maximum of five 32-bit words in length. Message delivery is reliable, but no guarantee is made on ordering. The study uses a 128-node CM-5, although machines of 1024 nodes are currently in the field.

2.2 J-Machine

The J-Machine is a massively-parallel MIMD computer based on the Message-Driven Processor (MDP) interconnected by a 3-D mesh network. The MDP is a single-chip processing node composed of a 16 MHz 32-bit integer unit, a 4K by 36-bit static memory, a closely integrated network interface, a packet router, and an ECC DRAM controller. The on-chip memory is augmented with a 256K by 36-bit off-chip memory. The 36-bit words include 4-bit tags, which indicate data types such as booleans, integers, and user-defined types. Two special tag values *future* and *cfuture* cause a trap when accessed. The MDP has three separate priority levels: background, 0, and 1, each of which has a complete context, consisting of an instruction pointer, four address registers, four general-purpose registers, and other special-purpose registers. A 512-node J-Machine has been built, and a 1024-node machine is planned.

The MDP implements a prioritized scheduler in hardware. When a message arrives at its destination node, it is automatically written into a message queue, consisting of a fixed-size ring buffer in on-chip memory. Background execution is interrupted by priority 0 message reception, which in turn may be interrupted by priority 1 message reception.

2.3 TAM

TAM defines a fine-grained parallel execution model used as a compilation target for Id90. Although it grew out of work on dataflow, it defines a simple model of self-scheduling threads that can be implemented on any machine. The key ways in which TAM differs from "thread packages" are that TAM threads are even lighter weight, the scheduling is integrated with aspects of compilation, such as register allocation, and there is no external scheduler.

A TAM program consists of a collection of *code-blocks*, which typically represent functions or loops in the source program. Each code-block consists of a collection of *threads*, which correspond roughly to basic blocks. Two instructions appear in the same thread only if they can be statically ordered and if no operation whose latency is unbounded occurs between them.

The TAM execution model centers on the *activation frame*, which is the analog of a stack frame for parallel calls. To invoke a code-block, a frame is allocated on a processor and initialized, and arguments are sent to the

frame. Initialization consists of setting the values of *synchronization counters* stored within the frame. A thread is allowed to run only when all its antecedents have been executed. To detect the completion of antecedents, a synchronization counter is associated with each thread. The counter is omitted for threads that have only one antecedent, *i.e., unsynchronizing* threads. For each frame, a stack of instruction pointers, called the *continuation vector* (CV), holds the list of threads that are ready to run. The arguments to the code-block, results from subordinate calls, and responses to global heap accesses are received by *inlets*. Inlets are compiler-generated message handlers that copy the arguments into the frame and enable computation dependent on the message. In order to process requests from the network quickly, inlets are small and run at a higher priority than threads.

Maintaining the thread queue in the frames provides a natural two-level scheduling hierarchy. When a frame is scheduled, the *remote* continuation vector (RCV) is copied into the *local* continuation vector (LCV), from which enabled threads are executed until the LCV is empty. The set of threads that run during this time is called a *quantum*. Each processor maintains a queue of *ready* frames with non-empty CVs. A new frame is activated from the queue when a quantum completes.

Global data structures in TAM provide synchronization on a per-element basis to support I-structure and M-structure semantics [10]. In particular, reads of empty elements are deferred until the corresponding write occurs. Accesses to the data structures are split-phase and are performed via special instructions: `ifetch` reads an element by sending a message to the processor containing the data which returns the value to an inlet, `istore` writes a value to an element, resuming any deferred readers, and `ialloc` and `ifree` allocate and deallocate I-structures.

In the current implementation of TAM, instructions are primarily three address, where the operands are constants, registers, or frame locations. TAM registers and frame slots are statically typed into integers, floats, various pointers, and generals. Generals are sufficiently large to contain any TAM type but do not identify the type. Correct compilation ensures that the producer and consumer of a general agree on the type of the contained value. No fixed limit is placed on the number of TAM registers, although the compiler tries to use them as efficiently as possible. The translator from TAM to a target machine is responsible for mapping TAM registers to physical registers or spill areas.

The key issues presented by TAM are the parallel call, dynamic synchronization of computation with asynchronous responses from both remote requests and calls, split-phase remote operations, and the overlap of computation with communication.

2.4 Mapping to the machines

The basic mapping of TAM to the two machines is relatively straightforward. Program code is placed on every processor, but a given code-block invocation takes place on a single processor. Because the compiler may pull loops out into separate code-blocks, these can be spread across the machine to implement parallel loops [7]. The memory on each processor is divided into two areas. One holds small arrays and activation frames. The other holds large arrays, which are spread across all the processors such that logically consecutive elements are on different processors. Memory is managed explicitly through library routines.

The J-Machine implementation of TAM [15] makes direct use of the hardware support for different priority levels. Threads run at the background priority level, allowing them to be quickly interrupted by messages arriving in the priority 0 queue. (Priority 1 is currently not used.) Because each priority level has its own register set, inlets do not interfere with thread execution. An address register is set aside in each register set to hold the frame pointer. Threads use an additional general-purpose register to hold the address of the top of the LCV. Two general-purpose registers are used as temporaries to hold memory operands and to implement complex TAM instructions. The remaining general-purpose register is used to hold one TAM register. All other TAM registers are mapped to the base of on-chip memory, a region that can be addressed easily. Frames are stored in main memory.

A similar approach is followed on the Sparc with inlets using a new register window. However, due to the tight coupling between threads and inlets, it proves to be more efficient to simply partition a single window. The CM-5 implementation [8] uses 32 registers divided into three classes: *global registers* which hold frequently-used values, *TAM registers* which are preserved for the duration of a quantum, and *inlet registers*, used during inlet execution and to pass information from threads to inlets. The CM-5 translator attempts to keep as many TAM variables as possible in the TAM registers and spills the rest into the frame.

2.5 Benchmarks

The empirical basis for comparison is provided by six benchmark programs described below. TAM-level dynamic instruction distributions are collected by running an instrumented version of the program on the CM-5. The translator inserts in-line code to record roughly a hundred specific statistics on each processor, which are combined at the end of the program.[1] These are grouped into the basic

[1] The Benchmark programs, raw data, and tools to process the data can be retrieved by anonymous FTP from ftp.cs.berkeley.edu under /ucb/TAM/isca93.tar.Z.