PROCEEDINGS OF THE INTERNATIONAL CONFERENCE
ON FIFTH GENERATION COMPUTER SYSTEMS 1988,
edited by ICOT. © ICOT, 1988

MECHANISMS FOR CONCURRENT COMPUTING

William J. Dally

Artificial Intelligence Laboratory and
Laboratory for Computer Science
Massachusetts Institute of Technology
Cambridge, Massachusetts

ABSTRACT

Concurrent computing is fundamentally different than sequential computing. Task size is orders of magnitude smaller making synchronization and scheduling major concerns, the critical resources are communication and memory, and programs distribute tasks rather than looping. Conventional hardware and operating system mechanisms are highly evolved for sequential computing and are not appropriate for concurrent systems. This position paper examines the mechanisms required by concurrent systems and the structure of a system incorporating these mechanisms.

1 FUNDAMENTAL PROBLEMS

1.1 Primitive Mechanisms

A fundamental hardware problem is to identify a set of primitive mechanisms that efficiently support a broad range of concurrent execution models. Sequential machines have evolved stacks for memory allocation, paging for memory management, and program counters for instruction sequencing. Concurrent machines have very different demands in each of these areas; the sequential mechanisms are no longer appropriate. However, no concurrent mechanisms have yet evolved to take their places. Today's concurrent computers either interpret their execution model using sequential mechanisms or are hardwired for a single execution model.

The message-driven processor (MDP) [4] [6] is designed to evaluate concurrent execution mechanisms for communication, synchronization, and naming. A SEND instruction and hardware message reception and buffering allow efficient communication of short messages across a high-speed network [7]. Synchronization is supported by a dispatch mechanism that creates a new process to handle a message in a single clock cycle. A general purpose translation mechanism supports naming. These mechanisms provide the primitive support required by many concurrent models of computation including dataflow [9], actors[1], and communicating processes [11].

1.2 Resource Management

At the operating system level, a key problem is to develop resource management techniques suitable for concurrent systems. In a concurrent system, communication bandwidth and memory capacity are the limiting resources; processor cycles are almost free. This situation is the opposite of the sequential case where processor cycles are considered the critical resource and communication is not a consideration. To complicate the situation the resources are physically distributed. Objects and processes must be placed in a manner that balances memory and processor use across the machine and reduces communication. The JOSS operating system [14] [15] is designed to satisfy these unconventional requirements.

Methods must also be developed to regulate concurrency. Many programs have too much parallelism and thus generate more tasks than can be accommodated in the available memory. To avoid the resulting deadlock, the system must regulate programs allowing them to generate sufficient concurrency to make use of all available processors, but reverting to more sequential execution before exhausting memory. Examples of regulation include controlled unrolling of loops [2] and adaptive (FIFO vs LIFO) scheduling [10].

To make efficient use of the communication resources, memory and tasks must be allocated in a manner that exploits locality. Placing objects near each other to improve locality is often at odds with the need to distribute objects for load balancing. Also there are some cases where communication bandwidth can be increased by spreading out a computation to make more channels available. For static computations min-cut placement techniques similar to those used to place electronic components [12] work well. Dynamic computations rely heavily on heuristics (e.g., placing an object near the object that created it) supplemented by reactive load balancing.

1.3 Overhead

To make use of a computer with thousands of processors,

a program must be decomposed into many small tasks. Each task consists of only a few instructions. In conventional systems, however, the overhead of scheduling, synchronization, and communication is many hundreds of instructions per task. This overhead restricts conventional multicomputers to operating at a very coarse grain size – thousands of instructions per task. Concurrency is reduced because there are fewer large tasks. Also, the resource management problems become harder as resources are allocated in larger chunks.

Overhead can be reduced to just a few instructions per task. The JOSS operating system, using the primitive mechanisms provided by the MDP, can create, suspend, resume, or destroy a task in fewer than ten instructions [15]. This efficient management of fine-grain tasks is achieved without sacrificing protection. Each task executes in its own naming environment.

2 CONCURRENT COMPUTER ORGANIZATION

To make the most efficient use of projected VLSI technology, general purpose concurrent computers will be constructed from a number of fine-grain processing nodes [5] connected by a low-latency, wire-efficient interconnection network [3].

2.1 Fine-Grain Processing Nodes

The *grain size* of a machine refers to the physical size and the amount of memory in one processing node. A coarse-grain processing node requires hundreds of chips (several boards) and has $\approx 10^7$ bytes of memory while fine-grain node fits on a single chip and has $\approx 10^4$ bytes of memory. Fine-grain nodes cost less and have less memory than coarse-grain nodes, however, because so little silicon area is required to build a fast processor, they need not have slower processors than coarse-grain nodes.

VLSI technology makes it possible to build small, powerful processing elements. A 1M-bit DRAM chip has an area of $256M\lambda^2$ (λ is half the minimum line width [13].). In the same area we can build a single chip processing node containing:

A 32-bit processor	$16M\lambda^2$
A floating-point unit	$32M\lambda^2$
A communication controller	$8M\lambda^2$
512Kbits RAM	$128M\lambda^2$

Such a single-chip processing node would have the same processing power as a board-sized node but significantly less memory per node. The memory capacity of the entire machine is comparable to that of a coarse-grained machine. We refer to a machine built from these nodes as a *jellybean machine* as it is built with commodity part (jellybean) technology [8].

A fine-grain processing node has two major advantages: density and memory bandwidth. Several hundred single-chip nodes can be packaged on a single printed circuit board permitting us to exploit hundreds of times the concurrency of machines with board-sized nodes. With on-chip memory we can read an entire row of memory (128 or 256 bits) in a single cycle without incurring the delay of several chip crossings. This high memory bandwidth allows the memory to simultaneously buffer messages from a high bandwidth network and provide the processor with instructions and data.

Fine grain machines are area efficient. Area efficiency is given by $e_A = A_1 T_1 / A_N T_N$ (where A_i is the area of i processors, T_i is execution time on i processors and N is the number of processors). Many researchers have measured their machines effectiveness in terms of node efficiency, $e_N = T_1 / N T_N$ Proponents of coarse-grain machines argue that a machine constructed from several thousand single-chip nodes would be inefficient because many of the processing nodes will be idle. N is large, hence e_N is small. A user, however, is not concerned with N, but rather with machine cost, A_N, and how long it takes to solve a problem, T. Fine-grain machines have a very high e_A because they are able to exploit more concurrency in a smaller area.

2.2 Wire-Efficient Communication Networks

VLSI systems are wire limited. The cost of these systems is predominantly that of connecting devices, and the performance is limited by the delay of these interconnections. Thus, an interconnection network must make efficient use of the available wire. The topology of the network must map into the three physical dimensions so that messages are not required to *double back* on themselves, and in a way that allows messages to use all of the available bandwidth along their path. Also, the topology and routing algorithm must be simple so the network switches will be sufficiently fast to avoid leaving the wires idle while making routing decisions. Our recent findings suggest that low-dimensional k-ary n-cube interconnection networks [3] are capable of providing the performance required by fine-grain concurrent architectures.

3 TRANSITION TO MAINSTREAM CONCURRENT COMPUTING

Select areas of mainstream computing will switch to concurrent computers when (1) concurrent software has matured to the point that it can support a large evolving application and (2) the performance advantage of these machines is sufficient to justify an investment in new software. Concurrent machines are appropriate for applications that are (1) limited by CPU performance (e.g., sci-

entific computing and signal processing) and (2) limited by memory system bandwidth (e.g., transaction processing). It is also expected that the availability of these machines will create new applications that were not previously possible.

REFERENCES

References

[1] Agha, Gul A., *Actors: A Model of Concurrent Computation in Distributed Systems*, MIT Press, Cambridge, MA, 1986.

[2] Arvind, and Culler, D., "Managing Resources in a Parallel Machine", Massachusetts Institute of Technology Laboratory for Computer Science CSG Memo 257, 1985.

[3] Dally, William J. "Wire Efficient VLSI Multiprocessor Communication Networks," *Proceedings Stanford Conference on Advanced Research in VLSI,* Paul Losleben, Ed., MIT Press, Cambridge, MA, March 1987, pp. 391-415.

[4] Dally, W.J. et.al, "Architecture of a Message-Driven Processor," *Proc.* 14th *ACM/IEEE Symposium On Computer Architecture,* 1987, pp. 189-196.

[5] Dally, W.J., "Fine-Grain Concurrent Computers", *Proc.* 3rd *Symposium on Hypercube Concurrent Computers and Applications,* 1988.

[6] Dally, W.J. et.al, *Message Driven Processor Architecture, Version 11,* MIT VLSI Memo, 1988.

[7] Dally, W.J., "Performance Analysis of k-ary n-cube Interconnection Networks," *IEEE Transactions on Computers,* To appear.

[8] Dally, W.J., "The J-Machine", to appear.

[9] Dennis, Jack B., "Data Flow Supercomputers," *IEEE Computer*, Vol. 13, No. 11, Nov. 1980, pp. 48-56.

[10] Halstead, R., "Parallel Symbolic Computation," *IEEE Computer,* Vol. 19, No. 8, Aug. 1986, pp. 35-43.

[11] Hoare, C.A.R., "Communicating Sequential Processes," *Comm. ACM*, Vol. 21, No. 8, August 1978, pp. 666-677.

[12] Kernighan B.W. and Lin, S., "An Efficient Heuristic Procedure for Partitioning Graphs," *Bell System Technical Journal,* Vol. 49, No. 2, Feb. 1970, pp. 291-307.

[13] Mead, Carver A. and Conway, Lynn A., *Introduction to VLSI Systems,* Addison-Wesley, Reading, Mass., 1980.

[14] Totty, B.K., *An Operating Environment for the Jellybean Machine,* MIT AI-Memo, 1988.

[15] Totty, B.K., and Dally, W.J., "JOSS: The Jellybean Operating System Software," to appear.