N90-16447

Dynamically allocating sets of fine-grained processors to running computations

David Middleton

ICASE, NASA Langley Research Center
Hampton, VA 23665.

Abstract

We explore an approach to using general purpose parallel computers which involves mapping hardware resources onto computations instead of mapping computations onto hardware. Problems such as processor allocation, task scheduling and load balancing, which have traditionally proven to be challenging, change significantly under this approach and may become amenable to new attacks. We describe the implementation of this approach used by the FFP Machine whose computation and communication resources are repeatedly partitioned into disjoint groups that match the needs of available tasks from moment to moment. Several consequences of this system are examined.

Keywords: *reconfigurable computers, partitionable computers, variable granularity, fine granularity.*

Mapping hardware resources onto computation structures

A standard part of parallel computation is mapping the computations onto the given structure of the hardware resources. The FFP Machine supports an alternative approach of mapping hardware resources onto the structure of running computations. We describe the method it uses and discuss some of the consequences of this approach.

The motivation for this inversion is that, although computations are more mutable than hardware, they are also highly dynamic in their structure. For example, an inner product computation begins with many small operations, the individual multiplications, which might be performed simultaneously given a fine-grained hardware structure. This is followed by a single, potentially large, summation (which may have a finer structure) better suited to more coarsely grained hardware. As a result of computations' highly variable structure, a fixed hardware structure will likely encounter difficulties implementing some of their stages efficiently. The philosophy of mapping hardware onto computations leads to a hardware design with a more flexible structure, which may reduce mismatches between the structures of the hardware and of the various computation stages.

Operations such as program decomposition, task scheduling and load balancing, which have traditionally been both essential for good performance and highly sensitive to several disparate parameters, heavily reflect the "mapping computations onto hardware" philosophy. The "mapping hardware onto computation" view ought to recast these problems drastically, opening the way to new methods for solving them.

As part of this approach, the FFP Machine implements the concept of a *virtual machine*, defined as an abstract entity created to perform a single task and consisting of many processing elements connected by a tightly-coupled message-passing combining network. The FFP Machine *partitions* its hardware elements into disjoint *resource groups* such that each virtual machine is provided with one entire resource group dedicated to its progress.

Properties of Partitioning

The FFP Machine is a reconfigurable fine-grained MIMD computer consisting of a linearly ordered set of processors which communicate through a tree-structured network of communication nodes. Each resource groups created by partitioning consists of a contiguous set of the processors connected by a tree of message processors embedded in the physical communication network. The following properties of the partitioning process are noteworthy.

Partitioning is very fast, yet still provides the flexibility usually associated with late binding.

From the innermost reduction rule of Backus's FP language [1], computations are defined as parenthesized expressions, with innermost parentheses delimiting computations that can proceed immediately. The expressions reside in the processors and resource groups are constructed by creating "breaks" where parentheses occur. Partitioning involves a single wave of messages that passes up through the tree network. The messages contain three bits; one indicates the presence of parentheses in the subtree and is sufficient for partitioning; the other two are used to determine which resource groups are delimited by a balanced pair of parentheses and so contain a virtual machine that can immediately proceed with its task. Each tree node calculates a logical sum, two logical products and sets its three communication channel switches. Figure 1 shows the internal structure of a communication node with the channel switches and the message processor which is allocated to one of the resource groups. Figure 2, Color Plate I (p. 693), demonstrates the result of partitioning, emphasizing the distinction between the physical tree structure and the tree structures of the resource groups. Partitioning takes logarithmic time, although with additional interlocks, constant time can be achieved through pipelining because the lower portions of the resource groups can be used while their upper portions are still being configured. Because of its simplicity, partitioning should add little overhead to machine operation and so may be performed frequently.

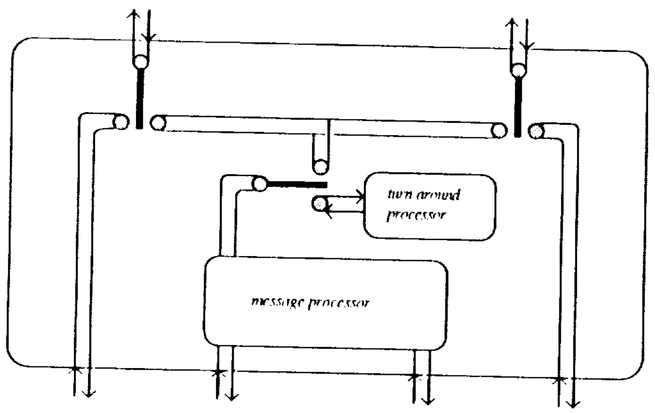

Figure 1. Communication node

- Apart from a virtual machine's determination of the pattern of parentheses it leaves in its result, no planning is required by either compiler or programmer to control the creation, activation or deletion of virtual machines. The size, placement and lifetime of the virtual machines is completely determined by the partitioning process, according to the positions of parentheses. These arise as the results of immediately preceding virtual machines without any prior calculation or storage of information.

- There are no size or alignment constraints placed on the virtual machines. Most notably, this avoids wasting resources through fragmentation, so, for example, a subtree of the physical machine with a thousand processors can support without help two virtual machines, one needing six hundred processors and the other, four hundred. (As a consequence of the non-alignment, the average depth of a virtual machine with n processors is $(lg(n) + 2.0)$ rather than $\lceil lg(n) \rceil$.) Virtual machines are *variably-grained*: their size can be tailored to reflect closely their individual needs without regard for other virtual machines. In particular, the size of a virtual machine relates closely to the size of its operands; identical functions applied to different data will generally be performed by different sized virtual machines.

- Resource groups are constrained to contain non-overlapping groups of processors. This imposed locality provides an upper bound on the number of resource groups (three) that a communication node may be required to support, allowing the design of the communication node to provide dedicated hardware.

  No contention or interference arises between communication operations occurring in different virtual machines; however, no communication can occur either. There does remain a communication bottleneck local to each resource group due to its tree structure. A richer interconnection in each resource group could be provided were the interconnection structure of the FFP Machine similarly enhanced [3, 5].

- Virtual machines can grow during their operation, with the communication network acquiring additional processors in a consistent fashion. This growth is achieved by shifting the contents of the processors so that more of them separate the pair holding the delimiting parentheses. In the current design, this *storage management* takes linear time and is the primary situation where one virtual machine can affect the operation of others, by potentially requiring that the contents of neighboring machines' processors also be shifted to make room.

- Multiple levels of parallelism are exploited. Concurrent virtual machines execute simultaneously, each internally exploiting fine-grained parallelism. (The term MSIMD has been used to describe parallel computers in which multiple SIMD machines execute simultaneously; by comparison, this might be described as an MMIMD machine).

- The message processor networks in the resource groups support *combining operations without* requiring costly associative memories in the switch nodes [2]. Each resource group has a tightly-coupled circuit-switched network of ALUs which can perform such cumulative operations (possibly within *groups* [7]) at hardware speeds rather than at the processors' instruction speeds. Cumulative operations provide a powerful mechanism capable of performing data permutations and parsing operations useful to the FFP Machine, without suffering from the bottleneck in a resource group. Other implementations of combining networks in general purpose machines have resulted in switch nodes that are too costly, probably due to the range of possibilities that the nodes must handle. By assuming that any given task involves closely coordinated actions by the processors, the complexity of the communication nodes is greatly reduced.

The potentials of flexible virtual machines

These virtual machines as supported by the resource groups defined above have a flexibility that provides opportunities for accomplishing tasks in new ways. Because partitioning creates virtual machines so cheaply, a task can profitably use many of them. In each of its stages, the resources already allocated to a task can be restructured into a different set of groups, so that, for example, stages that exhibit fine-grained parallelism can use many disjoint machines operating simultaneously. The following list demonstrates some of the ways that this flexibility can be used.

- It is possible to alternate between virtual machines that allow long-distance communication within the task, with the attendant communication bottleneck, and isolated virtual machines performing localized operations that communicate with no or greatly reduced contention.
- The TRAC machine could avoid explicit communication by reconnecting memory banks to different processors and so transferring data implicitly [4]. In an analogous fashion, some explicit communication in the FFP Machine can be avoided by reconfiguring the processors holding data into different resource groups, so that they belong to different virtual machines at different times. This is the standard method for passing results between functions when executing FFP programs.
- Computations structured as pipelines, or more generally, data-flow graphs (possibly with complex computations at the nodes) can be implemented by alternating between a set of virtual machines specialized to the individual nodes, and a set of virtual machines that perform the communication along the arcs of the graphs.

These uses, together with others, can be combined freely depending on the particulars of the task. We present one abbreviated example to show the possibilities [8].

OPS5 is a Production System language. When specified patterns can be found among subsets of known facts, corresponding actions are performed. Finding such patterns consumes a large majority of the processing time in OPS programs. The RETE algorithm, the best current technique for matching facts to the rule patterns, uses a discrimination network in which the nodes store partial matches found so far and compare them with new partial matches that arrive along their input arcs.

The discrimination network can be naturally implemented using virtual machines, as shown in Figure 2. A node has four parts including the local memory for storing partial matches and input and output buffer areas, each occupying as many processors as necessary. Pattern matching in each node uses a three stage cycle. With the node organized as a single virtual machine, a new pattern is broadcast from the input buffer to the processors holding partial matches. In the second stage, these processors are divided into many small machines each of which compares the new pattern with one previous partial match. In the third stage, successful comparisons cause a combined match to be placed in the output buffer.

Interleaved with the operation of virtual machines corresponding to nodes in the discrimination network is a set of machines corresponding the network arcs, which transmit successful matches from output areas of some nodes to the input areas of their descendants.

Figure 2 shows a discrimination network and its mapping as a hierarchy onto the linear array of processors. The graph is laid out as a series of nodes, each having four parts; individual processors within those parts are too small to be seen. Beneath the linear array of processors, bars show the groupings of processors into virtual machines for different phases in the matching operation. The first three rows correspond to the stages of the network node virtual machines and the last row corresponds to those for the arcs.

The arcs being able to send simultaneously relies on two facts: the discrimination network was created as a skew tree, and the input and output buffers could be placed within each node so as not to interfere. These choices display an important part of efficiently using virtual machines. Given the "logic in memory" and associative processing style of operation [6], it is less important that data be organized with regard to access methods that reflect sequential styles of algorithms. Instead, the data need to be organized so as to provide locality, in some sense, at appropriate stages in the tasks.

The ability of virtual machines to have different sizes depending on the amount of data is particularly important since the memories in the discrimination network nodes display a high variance with different input values.

Conclusions

The approach of mapping hardware resources onto computation structures, rather than vice versa, provides many novel opportunities for performing tasks. The implementation of this philosophy embodied in the FFP Machine has a significant affect on the way in which computations are viewed and organized.

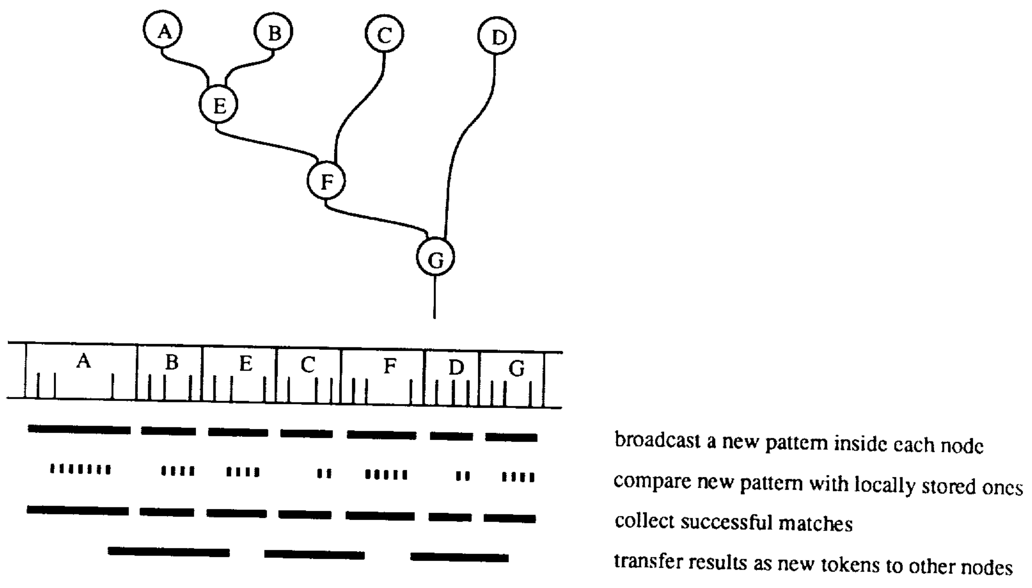
A
B
C
D
E
F
G
A B E C F D G
broadcast a new pattern inside each node
compare new pattern with locally stored ones
collect successful matches
transfer results as new tokens to other nodes

Figure 2. Virtual machine implementation of OPS5 discrimination network

Programming effort concentrates on arranging that data are organized to be clustered when they are combined or otherwise manipulated in a task. The "logic in memory" character by which the data reside in processors removes the requirement that the data be sorted and stored in structures that reflect the physical problem. "Associative programming" techniques allow data to be stored "out of order", in some sense, but with descriptors that determine when and how the data participate in operations [6].

Tasks are defined syntactically. The programmer's control over scheduling concentrates on creating and deleting the parentheses that delimit virtual machines. Barrier synchronization derives naturally from the partitioning mechanism; virtual machines delimited by non-innermost parentheses do not begin operation until those inner computations have completed and the parentheses are removed. Other synchronization and scheduling mechanisms can be created with little additional effort.

References

[1] J. Backus, "Can programming be liberated from the von Neumann style? A functional style and its algebra of programs", Communications of the ACM, Volume 21 No. 8, pp. 613-641, August 1978.

[2] A. Gottlieb, R. Grishman, C.P. Kruskal, K.P. McAuliffe, L. Rudolph and M. Snir, "The NYU Ultracomputer - Designing an MIMD Shared Memory Parallel Computer", IEEE Transactions on Computers, Volume C32 No. 2, pp. 175-189, February 1983.

[3] J.N. Kellman "Parallel Execution of Functional Programs" Master's Thesis. University of California at Los Angeles. 1983.

[4] G.J. Lipovski and A. Tripathi, "A reconfigurable varistructure array processor", Proceedings of the 1977 International Conference on Parallel Processing, pp. 165-174, August 1977.

[5] D.A. Plaisted (a) "An Architecture for Fast Data Movement in the FFP Machine" Proceedings of the 1985 Conference on Functional Programming Languages and Computer Architecture. Springer-Verlag, LNCS 201. pp. 147-163. September 1985. Nancy, France.

[6] J.L. Potter, "Programming the MPP", pp. 218-229, in "The Massively Parallel Processor", edited by J.L. Potter, MIT Press, 1985.

[7] J.T. Schwartz, "Ultracomputers", ACM Transactions on Programming Languages and Systems, Volume 2 No. 4, pp. 484-521, October 1980.

[8] B.T. Smith and D. Middleton, "Exploiting fine-grained parallelism in Production Systems", Proceedings of the 7th Biennial Conference of the Canadian Society for Computational Studies of Intelligence, pp. 262-270, edited by R. Goebel, Edmonton, Alberta, Canada, June 6-10 1988.