Towards a Large-Grain FFP Machine

Gyula A. Magó
Raj K. Singh
Vernon L. Chi

The FFPM Project
in collaboration with the
Microelectronic Systems Laboratory
The University of North Carolina
Department of Computer Science
CB#3175, Sitterson Hall
Chapel Hill, NC 27599-3175

This work was sponsored by NSF/CISE/MIPS under grant number MIP-8702277 and ONR under contract number N00014-86-K-0680. To be presented at TENCON '89 session (IEEE region 10 conference) on "Functional Programming Languages: Theory and Applications", November 22-24, 1989, Bombay, India. The authors gratefully acknowledge the valuable contribution of the entire FFPM project team: Edoardo Biagioni, Tai-sook Han, William Partain, David Plaisted, Jan Prins, Bruce Smith and Donald Stanat of UNC, and Ting Feng, Charles Molnar, and Fred Rosenberger of Washington University.

Reference and title

TR89-018
Towards a Large-Grain FFP Machine

Publication history

Version 1.0: 24 May 1989

TOWARDS A LARGE-GRAIN FFP MACHINE[1]

G.A. Mago, R.K. Singh and V.L. Chi
Department of Computer Science
University of North Carolina
Chapel Hill, N.C. 27599-3175

1. Introduction

Linear, or almost linear, scaling within wide parameter ranges has been an important goal of many multiprocessor projects, and it is a goal that is proving to be hard to reach. A pivotal organizational issue for multiprocessors is locality of references, but a frequent choice is to ignore the issue and provide a rich interconnection network to support arbitrary references. Such high-flux networks [ULL84] will be hard to build on a large scale, as doubling the size of such networks means far more than doubling the cost. In addition, performance growth will be far less than linear as a function of the network size. Amdahl [AMD88] offers a simple, quantitative analysis to support such a claim for hypercube networks, and his analysis shows that locality of references is indeed a parameter crucially influencing how well a multiprocessor scales.

This paper is concerned with a multiprocessor whose design centers around systematically and automatically preserving locality in computations. The FFP Machine (FFPM) [MAG79] is a small-grain multiprocessor originally designed for the direct, reduction-style execution of the FFP language [BAC78]. It employs massive parallelism in its operation (a) to perform computations specified by user programs, and (b) to perform operating system functions, such as locating reducible applications and managing storage throughout the system.

[1]This work was supported in part by NSF grant MIP-8702277 and by the Office of Naval Research, Contract N00014-86-K-0680.

The operating system functions are always massively parallel in this machine (e.g., all PEs always work together to find reducible applications), whereas user programs exploit parallelism to widely differing degrees. Assuming that few application programs are inheretly massively parallel, it is probable that a large grain implementation of the FFPM (we shall call it LGFFPM) could perform as well for most applications. Only in the most massively parallel of applications programs would the original FFPM be expected to excel. Such a machine would not enjoy the benefits of massive parallelism in its operating system functions, and thus it is likely to be substantially different from the original design.

This paper briefly describes an LGFFPM, and offers various comparisons with the original FFPM design.

The FFPM has been described elsewhere [MAG79,MAG80,MAG84, MAG89], and this paper builds upon those descriptions. Some of the main ideas of the FFPM, such as the (permanent) linear representation of the FFP expression, carry over into the new design. The main difference is that the LGFFPM has a large-grain leaf processor (we shall refer to it as LP) as opposed to the small-grain leaf cell of the FFPM. All other differences are consequences of this.

2. Description of the LGFFPM

Some important characteristics of LGFFPM are the following:
(1) The leaf processor (LP) holds a substring of the FFP expression (typically hundreds or thousands of FFP symbols), and its behavior is equivalent to that of a subtree of T and L cells in the small-grain FFP Machine. In the example of Figure 1, two RAs (B and C) are fully contained in an LP and two others (A and D) are spread across more than one LP .

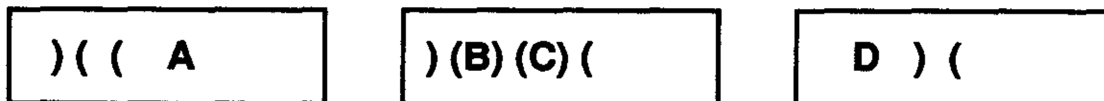

Figure 1

(2) The LP contains S *cells* (S for symbol), each of which is an FFP symbol plus related information. All S cells are of the same size, and they form a linear linked list. This list is the permanent representation [MAG89] of the FFP expression.

(3) Expressions eligible for being rewritten are the innermost applications, and are called *reducible applications* (or RAs for short). An RA fully contained in an LP is a *local RA*, otherwise it is a *global RA*.

(4) Expressions expand by having new S cells inserted into the linked list of existing S cells. The process of creating new S cells, and removing S cells not needed any more, is called *local storage management* (local because it takes place within an LP). *Global storage management* is needed only when at least one LP runs out of available storage. This is accomplished by moving the contents of LPs that are full (or about to be overfilled) to other LPs. This is similar to what is usually called load balancing.

(5) Only global RAs make use of the T cells. The linear representation of the FFP expression preserves locality, and allows effective use of the binary tree interconnection network which is dynamically partitioned to allocate parts of it to disjoint RAs; thus the root of the whole network never becomes a bottleneck. Partitioning is incremental: as soon as a global RA comes into existence, its component tree machine gets constructed with little delay (by setting appropriate switches in the T cells) without having to interrupt any of the ongoing computations in the machine. By contrast, in the small-grain FFPM partitioning is not incremental: the tree of T cells is always emptied before the network is repartitioned.

2.1. Linear Representation and Leaf Processor Capacity

The number of S cells within an LP varies during execution. The nominal number of S cells typically put into the LP by global storage management is a changeable parameter of the system, which could be something between 64 and several thousands. The value of this parameter should be chosen in such a way that space remains for a certain amount of expansion in the LP. As

execution proceeds, the LP may find itself holding fewer or more than the nominal number of S cells. Of course, too many S cells will tend to slow down execution.

The transient representation is not needed by local RAs. Most such RAs can be executed in a single pass (possibly using a stack to get through arbitrary expressions). The transient representation is identical to that in the small-grain FFPM, with possibly some pointers added to speed up access to certain S cells that are roots of subexpressions.

2.2. Overall Operation of LGFFPM

This is very different from the small-grain version for several reasons:

(1) Global storage management (STMG for short) should be done rather infrequently because (a) partitioning just after global storage management is costly: it requires a full scan of the new contents of the LP to rebuild the representation (the LP must explore the new expressions it now contains); (b) all RAs except those needing global STMG should be allowed to complete their execution before STMG shifting takes place (this avoids many complications in the reduction routines and the kernel, and results in smaller S cells to be shifted).

(2) Global storage management can be done infrequently, because the actual capacity of an LP is much larger than its nominal capacity. Thus each LP can manage its own storage for a while.

(3) Partitioning should be done frequently so as not to delay the simple RAs.

(4) Partitioning can be done frequently, if it is done incrementally, i.e., without disrupting ongoing computations.

So the overall operation can be characterized by saying that partitioning may be performed an arbitrary number of times between two successive global storage managements. Thus the notion of machine cycle is eliminated.

Subcomputations (RAs) synchronize with each other much less frequently than in the FFPM, which makes for a "more distributed," "more asynchronous," and, from the performance point of view, more responsive operation.

2.3. Partitioning

Partitioning is global, and will be done frequently, and incrementally, i.e., without emptying the tree, or disturbing ongoing computations in any way. The *global port* of LP (one of the ports that connects to the nearest T cell, briefly described in Section 2.7) always contains the current partitioning information about the contents of LP (this requires only three bits [MAG84]).

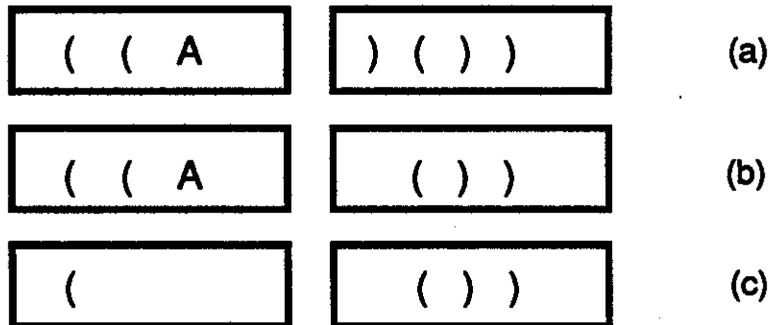

Figure 2

When the controller demands partitioning by sending down a suitable packet, all global ports respond without delay with their respective three bits. Different LPs do not synchronize with each other while doing this. Thus the global network does not necessarily "see" a well-formed expression at all times, as shown in snapshot (b) of Figure 2. The partitioning switches may be set in response to an ill-formed expression, but the part of the network connected to the ill-formed expression will not be used. This is accomplished as follows: (1) Component tree machines corresponding to RAs just finishing execution are emptied. (LPs are synchronized here.) (2) LPs involved in processing such RAs do their final rewriting, and update the partitioning information at the global ports, although not necessarily at the same time. (3) The partitioning packets pick up this new information (different LPs do not synchronize with each other

here). (4) Every LP ascertains that the partitioning packet took the new information before it begins to send into its component tree machine.

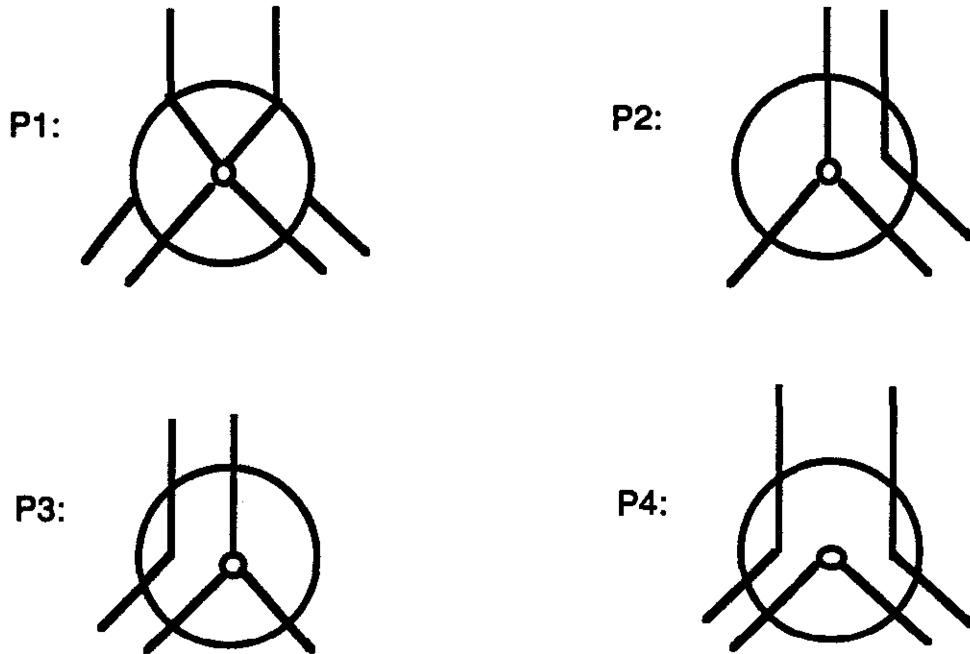

Figure 3

Incremental partitioning means that the partitioning switches of a T cell may be reset while parts of that T cell are involved in some computation. Since an arbitrary T cell has four distinct partitioning configurations (shown in Figure 3), it may, during partitioning, make one of the sixteen possible transitions among those configurations. Figure 4 shows all the possible state transitions of the T cell. During transitions P1 to P1, P2 to P2, P3 to P3, and P4 to P4 the switch settings do not change. Thus all parts of the T cell making one of these transitions are able to provide continuous, uninterrupted connection during partitioning. In transitions P2 to P4, and P4 to P2, the right side channel may serve an RA that still computes. Similarly, in transitions P3 to P4, and P4 to P3, the left side channel may serve an RA that still computes. In the remaining

eight transitions, the FFP expression under the T cell has changed sufficiently so that the tree can be assumed to be empty (rule (4), above, gurantees this).

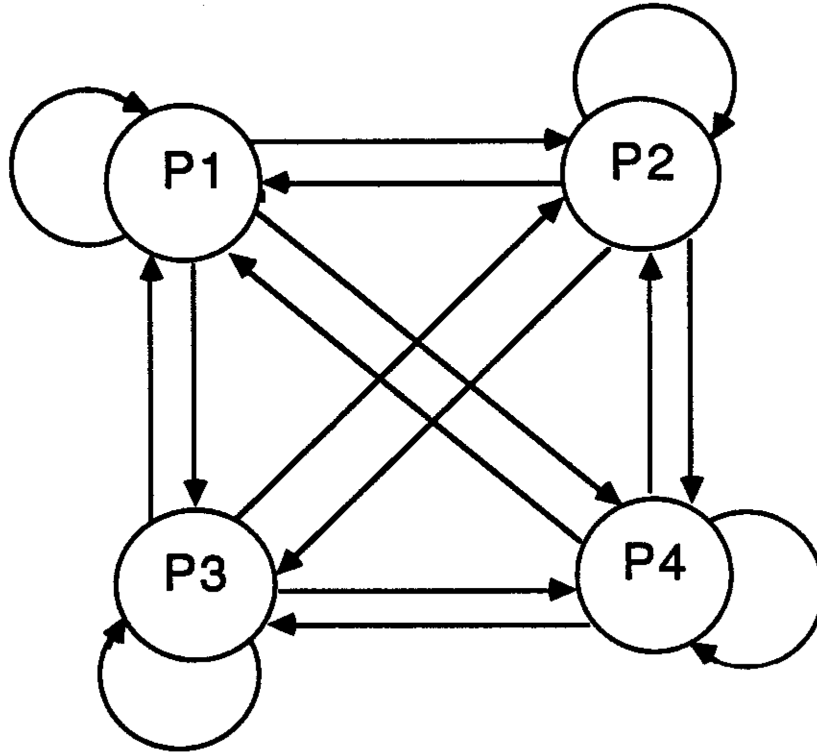

Figure 4

2.4. Storage Management

When an expression expands, the corresponding RA tries to perform local STMG first. If there is not enough room in the LP, it switches to requesting space from global STMG, and then waits till the requested space is provided.

Global STMG is needed when there is not enough space left in the LP to do local storage management. Once the controller decides that global STMG is needed, it begins to prepare for it by winding down the computations and emptying the tree. This is done by not allowing RAs to start whose execution is expected to take a long time. In particular, every RA estimates its execution time from its operator and from the number of LPs involved in it, and

informs the controller. The controller periodically broadcasts a time-limit for those allowed to start up. An RA execution starts only if estimated execution time is within the allowed limit. If no global STMG is needed, the controller broadcasts the largest integer available as the time limit. Otherwise, it broadcasts the time needed for the longest RA currently in progress to finish.

2.5. The Reduction Routines

The reduction routines define the effects of FFP functions and functionals. They differ from the small-grain versions only in details forced by the change in granularity, but these differences are considerable. Some important details of execution depend on the following: (a) whether the RA is strictly internal to an LPt; (b) whether an RA uses transient representation; (c) whether an RA uses local or global STMG.

2.6. Message Waves

Most communications among LPs are permutations: each item has a unique destination. In the small-grain version the corresponding port of the L cell is programmed to look for a unique item. In the large grain version, messages carry target addresses so that only a range check needs to be done by the LP port to decide acceptance or rejection.

Sorting is an important operation. The LP must presort its contents before sending a stream into the tree (since each T cell merges two sorted streams), thus it is not wise to interrupt sorting in order to do global storage management. (If sorting could be interrupted, then the machine would have to know which of the messages got through, and after global storage management, re-sort its contents again before resuming the sort operation.) Having to run a sort till completion, before performing global storage management, is a disadvantage, but probably not a big one.

2.7. LP hardware and software

The leaf processor (LP) hardware consists of an instruction set processor, and ports connected to one T cell and two other LPs. The T cell is the gateway to the interconnection network, which is a binary tree in the simplest case. (A richer interconnection network, such as a 2D rectangular grid, may require more ports.) There are three ports that connect LP to the T cell above. Two of the ports connect to the partitioned part of the interconnection network (corresponding to the details shown in Figure 3), whereas the third port, referred to as the *global port*, connects to the non-partitioned (also called global) part of the interconnection network. The two ports leading to two other LPs are used for global storage management only.

The software for the LP could be organized as a set of communicating sequential processes. In a (yet unpublished) solution worked out by Tai-sook Han, there is a process dedicated to each hardware port and to each of the four types of tasks shown in Figure 5.

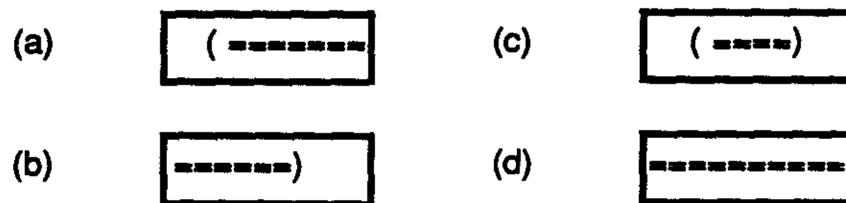

Figure 5

The LP gives top priority to processing global RAs. It is involved with at most two of them: (with the notation of Figure 5) either (d) alone, or one or both of (a) and (b). In the latter case, there may be many RAs fully contained within the LP, in which case a process corresponding to (c) in Figure 5 works on one at a time.

3. Comparison

The chief motivation for a large-grain version of the FFPM is that it requires a lot less hardware than the small-grain version, and the majority of the hardware can be off-the-shelf. It might be expected that performance would suffer greatly. Fortunately, there are many factors mitigating such a performance loss (some of them are listed below). As a result of these factors, the large-grain version might in fact be faster on certain computations, and cost/performance is likely to improve substantially.

3.1. Advantages of Large-Grain FFPM

(1) There is no need to load function definitions dynamically. A basic set of primitives would be stored in the machine permanently, and others loaded as needed by the FFP program before execution begins. The facilities for dynamic loading are in place should the need arise, but would not be used routinely.

(2) Smaller S cells are shifted during global STMG, since there is no need to move unexecuted reduction routine code as in the small-grain FFPM.

(3) Local STMG is often possible, because the capacity of LP is large. When local storage management is used, copying a subexpression from one place to another (e.g., copying the function expression when executing Apply-to-All) does not require requesting a sufficient amount of space prior to copying the expression, doing global storage management, and then copying the expression. The expression is simply distributed to all participating LPs where instances of it are inserted it into the linked-lists of S cells in the appropriate places. In this way many FFP primitives can finish without waiting for global storage management.

(4) Thanks to incremental partitioning, RAs held by several LPs may start without delay. This means improved "scalar performance." For example, if application B follows A (A is contained in B), B can start soon after A has finished. Global operation is now characterized by the fact that there may be any number of partitionings between two global storage managements. In other words, global storage management is now divorced from partitioning.

(5) An RA that is internal to an LP may start executing any time (i.e., not tied to partitioning), it can work without transient representation, and executes faster than the same RA held by two or more LPs (not only because of smaller size, but also because no messages and no transient representation are used).

(6) Numerical applications present no special problems (e.g., floating point, elementary functions), because numerical co-processors for LPs are economically feasible.

3.2. Disadvantages of Large-Grain FFPM

(1) Since massively parallel processing cannot be relied on, associate processing suffers (i.e., when every incoming message is to be processed against every local S cell).

(2) Each reduction routine needs two variants, one for local, the other for global RAs.

(3) The LP kernel is more complex than the L-cell kernel of the FFPM.

4. Conclusions

While commenting on the FFP Machine in a recent paper, Berkling [BER87] speculates that string reduction (which is the mode of operation for the FFPM) presupposes massive parallelism, and that only graph reduction can be implemented on a collection of von Neumann computers. LGFFPM indicates that string reduction can be implemented without massive parallelism. The LGFFPM is also interesting in that it offers a comparison between two parallel processors that support exactly the same model of computation, yet are of greatly differing granularities.

References

[AMD88] AMDAHL, G.M. Limits of expectations. *International Journal of Supercomputer Applications*, 2(1):88-97, 1988.

[BAC78] BACKUS, J. Can programming be liberated from the von Neumann style? A functional style and its algebra of programs. *Communications of the ACM 21*, 8 (1978), 613-641.

[BER87] BERKLING, K. System architectures for functional programming languages: problems and solutions. CASE Center TR 8724, Syracuse University, December 1987.

[MAG79] MAGO, G. A. A network of microprocessors to execute reduction languages. Two parts. *International Journal of Computer and Information Sciences*, 8, 5 (1979), 349-385, 8, 6 (1979), 435-471.

[MAG80] MAGO, G. A. A cellular computer architecture for functional programming. Digest of Papers, *IEEE Computer Society COMPCON* (Spring 1980), pp. 179-187.

[MAG84] MAGO, G. A. and MIDDLETON, D. The FFP Machine---A Progress Report. *International Workshop on High-Level Computer Architecture 84* (Los Angeles, California, May 23-25, 1984). (Reprinted in *Dataflow and reduction architecture* by S.S. Thakkar, IEEE Computer Society Press, 1987, and in *Computer architecture* by D.D. Gajski, V.M. Milutinovic, H.J. Siegel, and B.P. Furht, IEEE Computer Society Press, 1986, pp. 456-468.)

[MAG85] MAGO, G. A. Making parallel computation simple: the FFP machine. Digest of Papers, *IEEE Computer Society COMPCON* (Spring 1985), pp. 424-428. (Reprinted in *Computers for artificial intelligence applications* by Benjamin Wah and G.J. Li, IEEE Computer Society Press, 1986, pp. 324-328.)

[MAG89] MAGO, G.A. and STANAT, D.F.: The FFP Machine. Chapter 12 in *High Level Language Computer Architecture* (Milutinovic, V., ed.). Computer Science Press, 1989.

[ULL84] ULLMAN, J.D. Some thoughts about supercomputer organization. Digest of Papers, *IEEE Computer Society COMPCON* (Spring 1984), pp. 424-432.