Storage Allocation for List Multiprocessing

by

Steven D. Johnson
Department of Computer Science
Indiana University
Bloomington, IN 47405

TECHNICAL REPORT NO. 168

STORAGE ALLOCATION FOR LIST MULTIPROCESSING

by

Steven D. Johnson

March, 1985

This material is based on work supported by the National Science Foundation under Grant No. DCR 84-05241.

STORAGE ALLOCATION FOR LIST MULTIPROCESSING

Steven D. Johnson

ABSTRACT

Communication in a hypothetical general-purpose list multiprocessor is considered. The gross architecture is a collection of processors sharing access to a collection of memories through a multi-stage, buffered routing network. The design goal is to decentralize process coordination through refinements to the network structure and the design of routing elements.

This issue considered here is storage allocation. Routing elements anticipate and buffer "new" requests, making the connection network a reservoir for anticipated processor demands.

The result is an implicit dissipation of resource demand among memories and an implicit distribution of resource supply among processors. The "new-sink" solves coordination problems for a fundamental operation on the data space.

Although motivated by a demand-driven computational model for purely applicative programming, the technique applies to any multiprocessing architecture with distributed resources, and suggests a general approach to other coordination problems.

1. Introduction

This paper sketches elements of a hypothetical multiprocessor—I will call it *LiMP*—with a typical gross organization: it is a collection of processing elements (PEs) connected to a collection of shared storage elements (SEs) by a multi-stage, buffered routing network. LiMP's design is distinctive in that it is intended to do parallel, general-purpose *list* processing. Hence, among other things it must provide for automatic storage management in the presence of parallelism. Storage allocation is an example of an implicit task that must be decentralized if parallel performance is to be obtained.

Decentralization of storage allocation leads to pervasive refinements in all of LiMP's major components. "Available space" is distributed by requiring that each SE maintain a local pool of new cells for dissemination to the PEs. However, not much is gained if the PEs are forced to contend directly for access to these pools. Substantial choreography would be needed to synchronize the locating of, and transactions with, individual SEs with cells available. The solution proposed here is similar to the *UID network* mentioned by Ackerman for data flow computer memories [1, Section 5.0.5]. A buffered routing network is used to disseminate resources. Each routing element retains the addresses of a few new cells. The network becomes a reservoir of available space that both arbitrates and dissipates processors' resource demands.

The goal of parallel list processing may require some justification. Briefly, LiMP is motivated by a operational model of computation based on a *suspending constructor*, or "lazy CONS" [4]. As a basis for functional programming (no overt mention of state; no implicit reference to time), a suspending constructor offers a means to introduce parallelism as a byproduct of apparently ordinary data manipulation [6, 8]. Process creation is transparent, processor assignment is dynamic, and processor interaction is mediated by the memory. That is, parallelism is subsumed as a property of the data space; it may or may not be addressed at a language level.

In his review of proposed architectures for functional programming [14], Vegdahl correctly states that treatments of parallelism based on a suspending constructor* are concerned with a model for parallelism but do not specify an architecture for implementation of the model. Although some concrete proposals have resulted from this research (*e.g.* [7, 15, 16]), they are generalized and presented outside the context of the model. This paper also deals with an isolated problem, but its presentation addresses broader issues of performance and programming in LiMP.

Section 3 presents one possible communication architecture for LiMP, a rectangular network. This network was used for a simulation of communication, discussed in Section 4. Although its details are of marginal relevance to the storage allocation problem, this example shows how related problems, such as dynamic scheduling, are diminished by the appropriate choice of network structure and design of network elements.

The behavior of the new-sink sheds light on other communication issues in LiMP. The conclusion in Section 5 discusses some of these issues in the context of the suspending-construction model.

2. Computation in LiMP

LiMP's memory is logically organized as a binary-list storage unit. An individual PE performs transactions which are analogous to those of an ordinary list processor. These transactions fall into three categories:

- An *RSVP* is a transaction which requires a response, like a "fetch" or, in Lisp terminology, a CAR or a CDR.

* I refer to Vegdahl's discussion of "Friedman and Wise's Reduction Machine," in which he also mentions that storage management is unspecified.

- A *STING* is like a "store"; it does not require a response.
- A *NEW* requests the address of an unreferenced list cell.

The SEs are not passive memories. For example, they participate in storage allocation by maintaining local available-space pools for NEW-requests. The PEs are not pure mills. Each is executing a locally stored (micro) program. A good example would be a language interpreter, but assume also that input/output devices are specialized PEs. Data is introduced to and extracted from the system through the list-space.

A switching network interleaves RSVP, STING, and NEW transactions with the shared SEs. It is convenient to use the terms "communication" and "message" in discussing this interleaving, but the sense of these terms is restricted to the elementary transactions just mentioned. It is easier to think of the routing network as having separate layers for PE-to-SE communication and for SE-to-PE communication.

Since messages are small, it is appropriate to consider using buffered switches and a multi-stage routing network. This trades "turnaround"—the time it takes to complete a transaction—for "throughput"—the number of transactions that can be in progress. Each layer of the network is composed of individual routing elements which interact through message ports. Each routing element contains a cross-bar switch through which it transfers a permutation of its input-messages to its output ports. Transfer to an occupied port is inhibited. Figure 1 is a high-level block diagram for one layer of a 2×2 routing element. It resembles a general-purpose router proposed for data flow prototyping [2].

The "New-sink".

STINGs and RSVPs suffer turnaround proportional to the distance between source and destination in the routing network. STINGs are about half as expensive as RSVPs because there is no response. The turnaround for NEWs can be reduced to about unit time if the network participates in storage allocation.

In a third communication layer, message ports store (the address of) an available cell. When a processor-side port is empty, the routing element transfers a new cell from its memory-side port. Cells migrate toward the PEs, making the routing network into a reservoir of available space. Once the reservoir is filled, sporadic NEW transactions are served immediately by the routing elements; holes in the reservoir are filled as empty buffers ripple back toward the SEs.

A burst of NEW-requests from an individual PE will dissipate in the new-sink; each successive stage makes more cells available to supply the burst. If the supply of cells is limited to a few SEs, they will be distributed according to PE demand.

3. An Example Network for LiMP

Figure 2 depicts an instance of rectangular routing network with the structure of a *banyan network* [13]. The banyan network is attractive for LiMP because it yields a uniform and decentralized addressing mechanism:

- The leading bits of a message's destination address define a path through the network to a unique SE, independent of the source of the message. Take '1' to mean "route left" and '0' to mean "route right". Figure 2 shows the path to destination '101' from all sources.

- The source-address of a message can be dynamically reconstructed—in this or any other network—by recording the switch states along its path. Let '×' denote a left-right, right-left transfer, and let '=' denote a left-left, right-right transfer. With destination '101', the *path record* '×=×' ('=×=') is recorded for a message originating at PE_0 (PE_6).

- The control of the routing elements is uniform and extensible. The path-control bit is consumed by shifting the destination field, and this leaves room for inserting the path-record bit.

If a banyan network—or a network with similar characteristics—is used, a process can be placed in any PE without address translation. The target address for a STING or an RSVP is independent of its source. Responses to RSVPs are determined dynamically by the path record. Storage allocation is also relative; the home SE of a new cell is computed by the new-sink.

3. Some Observations about LiMP's Performance

A preliminary simulation* of communication in LiMP reveals some finer points about message traffic. PEs are modeled as stochastic processes with states IDLE, NEW, STING, and RSVP. The communication network model is as described in Section 3. PEs wait for STINGs to be absorbed by the routing network, for NEWs to be satisfied by the new-sink, and for the *responses* to RSVPs. Of these behavioral assumptions the last is perhaps simplistic: one might expect PEs to multiplex several processes and thereby carry out concurrent RSVPs. However, some amount of waiting is beneficial: the resulting "holes" in the routing network reduce message blockage.

In the new-sink, performance thresholds occur when the supply of new cells approximates the demand. Figure 3(a) shows the typical relationship between PE utilization and SE *responsiveness*: the rate at which an SE it can respond to NEW transactions. Utilization reaches its maximum as responsiveness approaches the rate at which processors issue NEW requests.

To conserve locality, the new-sink is *biased* by defining a preferred switch setting for transfers. For example, if each routing element attempts, when possible, to use

* Details are given in [11]

the '=' switch position, then each PE tends to get its resources from the SE directly across from it. If the switch-preference is uniform, then each PE is associated with one SE.

In the banyan network all communication paths have the same length. Nevertheless, if the association between an PE and its preferred SE can be maintained, traffic in the network is better organized and less subject to message blockage. Figure 3(b) shows the typical relationship between locality and responsiveness, assuming a fixed rate for NEW requests. Locality is worst when supply matches demand.

Bias for the other communication layers may be harmful. In moderate traffic, utilization diminishes slightly as locality improves. Messages that go "against the grain" are blocked for longer periods. This phenomenon disappears quickly if there are a few holes in the network because of waiting.

6. Conclusions and Remarks

Communications in LiMP include bi-directional RSVP transactions and uni-directional STINGs. From the PE's point of view, and subject to message blockage, NEWs and STINGs are served in unit time, so RSVPs are the dominant expense in computation. If the routing network participates in storage allocation, then the fundamental transaction of obtaining a new cell is uni-directional and has minimal delay.

There is little point in distinguishing uni-directional and bi-directional traffic if messages are sufficiently mixed. However, there are significant examples of purely uni-directional communication. One of these is input, where a device introduces new data to the list space through a succession of NEWs and STINGs. Similarly, a context-swap would involve construction of a process activation record using several new cells and requiring no fetches.

The remainder of this section reviews related results about programming and performance in LiMP. A general-purpose list-multiprocessor could could serve as a vehicle for demand-driven computation based on the suspending-construction model. However, further refinement of its components would be needed.

A *suspending constructor* initializes newly allocated records with *suspensions,* which are formally described as valuations [4]. In operational terms, a suspension is a process which computes a *manifest* result, that is, another record. In this context, the terms "suspension" and "process" are synonymous. The continuation of every suspension is to *converge,* that is, to store a result in its home location and thus to de-reference itself. Two rules define the basic process-coordination mechanism in the model:

1. A process attempting a fetch from memory must wait for a manifest value.
2. The sole obligation of a process is to converge.

Suspensions are transparent and ubiquitous. They are created by the list-constructor primitive. A process is blocked when it encounters a suspension in the course of manipulating manifest data. Any transaction with memory may reveal—to the system—a needed suspension and hence an inter-process dependency.

Suspending construction was originally proposed as a basis for applicative programming. Its effects include call-by-need semantics for a standard language interpreter, the facility to manipulate non-finite data structures, automatic co-routine behavior, and a direct implementation of data recursion. Experience tends to support the thesis that a practical programming discipline can evolve in a purely applicative setting, provided that certain issues are resolved in hardware. The best example is input-output, where the suppression of notions about state and time must be extended to peripheral devices [3, Chapter 5].

If side-effects are prohibited, then suspensions are autonomous. They may be activated arbitrarily should idle processors be available. The worst possibility is that an extraneous activation might consume otherwise useful resources. However, suspensions often take up more space than their results; so their activation may also free resources.

A uni-directional STING raises questions about process synchronization. A weak-but-adequate arbitration primitive has been proposed [7], in which "stores" to memory are inhibited once an "inoculating bit" has been set. This *conditional-store* is sufficient to coordinate concurrent computation in the suspending-construction model [10, 8, 12].

The addressing transparency in the banyan routing network clears the way for dynamic processor allocation. A process is another example of a consumable resource. Thus, the routing network might participate in process-allocation—scheduling—by providing a "need-sink" (or perhaps "source") for disseminating (the addresses of) revealed processes. A dynamic scheduling strategy has been proposed in which need is measured by proximity to an output device [6]. Put another way, output devices are a source of absolute, resolute need, which is communicated *via* RSVPs to the data-space [5]. In the proposed scheme, RSVPs in LiMP would generate responses if the target is manifest and need-messages if the target is suspended.

Parallel storage reclamation is apparently harder than parallel storage allocation. Maintenance of available space entails substantial processing in the SEs. Reference counting has advantages in a distributed scheme, but propagation of counts raises still another communication problem and can recover only some circular structure [9]. Occasional garbage collection is necessary; a parallel collector for LiMP seems possible but has not been designed. Wise has designed a binary-heap memory, which maintains a local available-space pool, does internal reference-counting, supports a conditional-store operation, and can be collected when necessary [16].

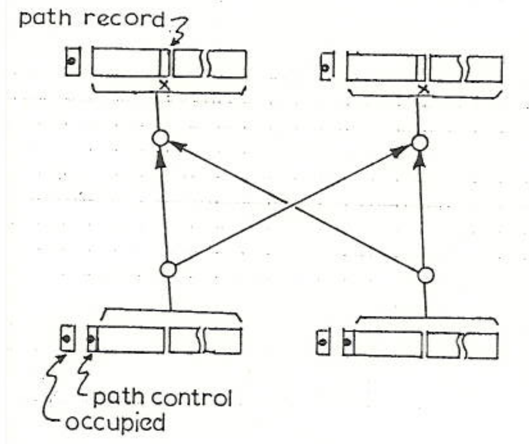

Figure 1. One Layer of a Routing Element.

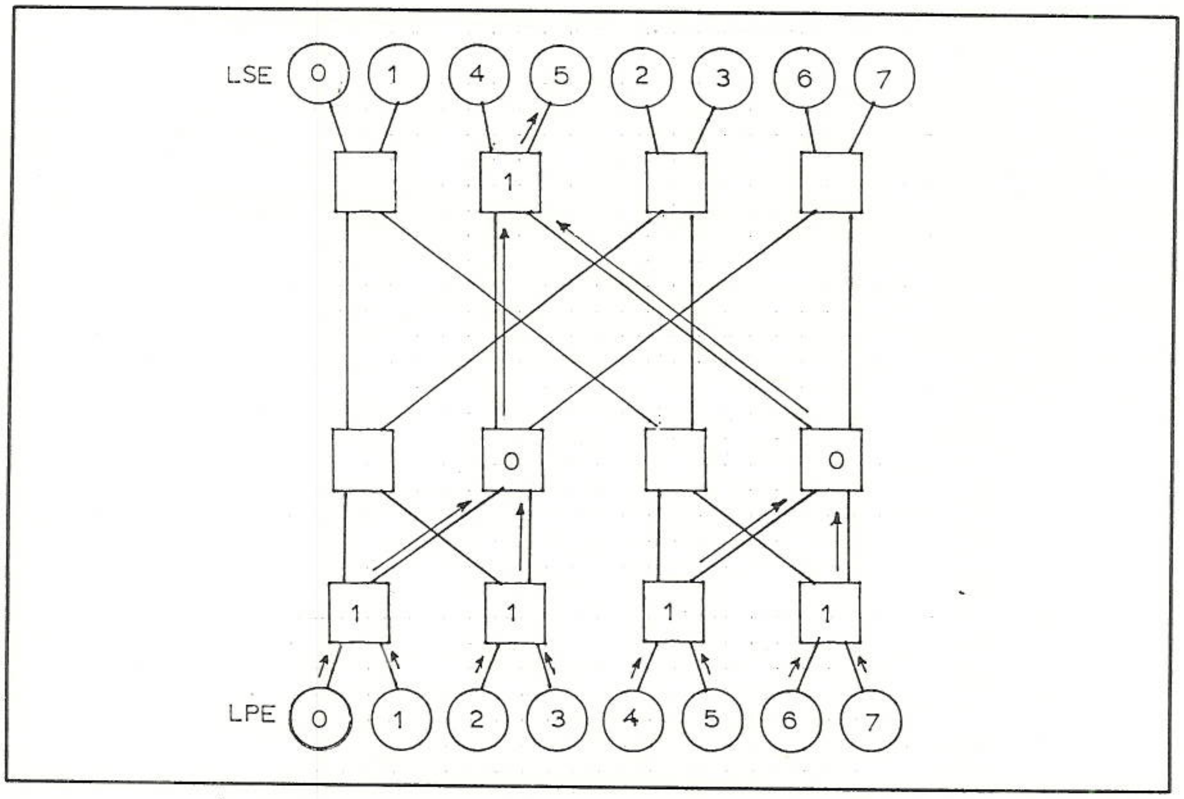

FIGURE 2. A Possible Network for LiMP.

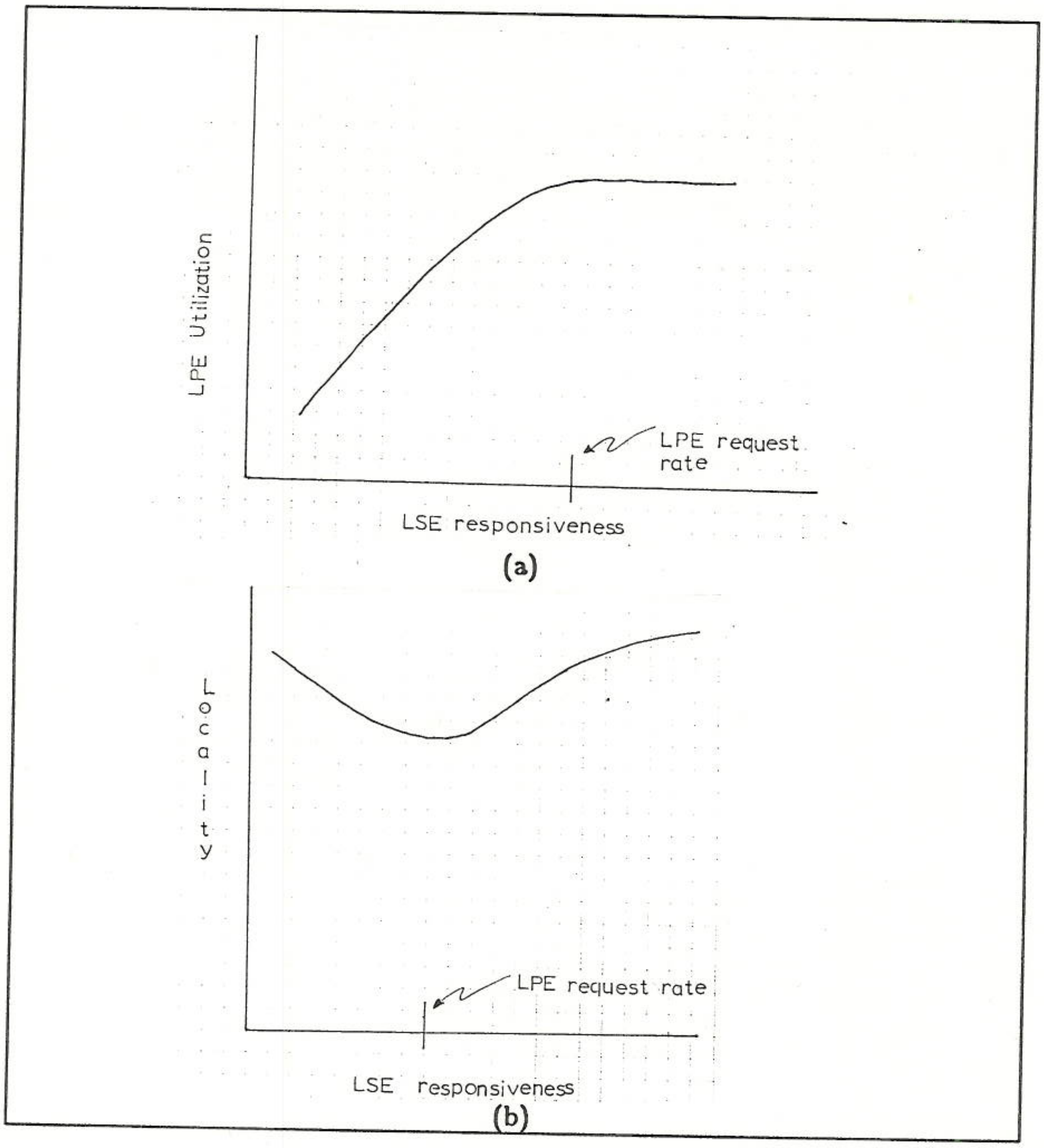

FIGURE 3. Effect of the New-sink on Performance.

References

[1] Ackerman, W. B., *A Structure Memory for Data Flow Computers,* M. S. Thesis, Dept. of Electrical Engineering and Computer Science, MIT, 1977. Published as MIT/LCS/TR-186, MIT Laboratory for Computer Science, 1977.

[2] Dennis, J. B., G. A. Boughton, and C. K. C. Leung, "Building blocks for data flow prototypes," *Proc. ACM-SIGARCH Seventh Annual Symp. on Computer Architecture,* (1980) pp. 1–8.

[3] Filman, R., and Friedman, D. P., *Coordinated Computing: Tools and Techniques for Distributed Software,* McGraw-Hill, New York, 1984.

[4] Friedman, D. P. and D. S. Wise, "CONS should not evaluate its arguments," *Automata, Languages and Programming,* (eds.) S. Michaelson and R. Milner, Edinburgh University Press, 1976, pp. 257–284.

[5] Friedman, D. P. and D. S. Wise, "Output driven interpretation of recursive programs, or writing creates and destroys data structures," *Information Processing Letters* **5**, No. 6 (1976), pp. 155–160.

[6] Friedman, D. P. and D. S. Wise, "Aspects of applicative programming for parallel processing," *IEEE Transactions on Computing* **C–27**, No. 4 (1978), pp. 289–296.

[7] Friedman, D. P. and D. S. Wise, "Sting-unless: a conditional, interlock-free store instruction," In M. B. Pursley and J. B. Cruz, Jr. (eds.), *Proc. 16th Annual Allerton Conf. on Communication, Control, and Computing,* University of Illinois (Urbana-Champaign, 1978), pp. 578–584.

[8] Friedman, D. P. and D. S. Wise, "An approach to fair applicative multiprogramming," *Proc. of Itl. Symp. on Semantics of Concurrent Computation* (eds.) G. Kahn and R. Milner, Springer, New York, 1979, pp. 203–225.

[9] Friedman, D. P. and D. S. Wise, "Reference counting can manage the circular environments of mutual recursion," *Inform. Proc. Ltrs.* 8 (1979), pp. 41–44.

[10] Friedman, D. P. and D. S. Wise, "Fancy ferns require little care," Technical Report No. 106, Computer Science Dept., Indiana University, 1981. Also in *Symposium on Functional Languages and Computer Architecture.* (eds.) S. Holmstrom, B. Nordstrom, and A. Wikstrom, Lab for Programming Methodology, Goteborg, Sweden, 1981.

[11] Johnson, S. D., "Connection Networks for Output-Driven Multiprocessing," Technical Report No. 114, Computer Science Dept., Indiana University, 1981.

[12] "A semaphore-free promotion strategy for `frons`," unpublished.

[13] Tripathy, A. R. and G. J. Lipovski, "Packet switching in banyan networks," *Proc. ACM-SIGARCH Sixth Annual Symp. on Computer Architecture* (1979) pp. 160–167.

[14] Vegdahl, S. R., "A survey of proposed architectures for the execution of functional languages," *IEEE Transactions on Computing* C-33 (1984), pp. 1050–1071.

[15] Wise, D. S., "Compact layouts of banyan/FFT networks," in *VLSI Systems and Computations,* ed. H. T. Kung, B. Sproull, and G. Steele, Computer Science Press, Rockville Maryland, 1981.

[16] Wise, D. S., "Design for a multiprocessing heap with on-board reference counting," Indiana University Computer Science Dept. Technical Report No. 163 (1985), submitted for publication.