10-1-1984

Dynamic Systolization for Developing Multiprocessor Supercomputers

Kai Hwang
Purdue University

Zhiwei Xu
Purdue University

Dynamic Systolization for Developing Multiprocessor Supercomputers

Kai Hwang
Zhiwei Xu

TR-EE 84-42
October 1984

School of Electrical Engineering
Purdue University
West Lafayette, Indiana 47907

Dynamic Systolization for Developing Multiprocessor Supercomputers

Kai Hwang, Senior Member, IEEE, and Zhiwei Xu
Purdue University*

Abstract: A dynamic network approach is introduced for developing reconfigurable, systolic arrays or wavefront processors. This allows one to design very powerful and flexible processors to be used in a general-purpose, reconfigurable, and fault-tolerant, multiprocessor computer system. The concepts of macro-dataflow and multitasking can be integrated to handle variable-resolution granularities in computationally intensive algorithms. A multiprocessor architecture, Remps, is proposed based on these design methodologies. The Remps architecture is generalized from the Cedar, HEP, Cray X-MP, Trac, NYU ultracomputer, S-1, Pumps, Chip, and SAM projects. Our goal is to provide a multiprocessor research model for developing design methodologies, multiprocessing and multitasking supports, dynamic systolic/wavefront array processors, interconnection networks, reconfiguration techniques, and performance analysis tools. These system design and operational techniques should be useful to those who are developing or evaluating multiprocessor supercomputers.

Index Terms: *Systolic arrays, wavefront arrays, interconnection networks, macro dataflow, multitasking, reconfiguration techniques, supercomputer performances.*

1. Introduction

Multiprocessor supercomputers are playing a vital role in modern civilization. With fixed functional capability of the processors and fixed interconnection structures, most existing supermachines have very biased performances [28]. A machine may perform very well for certain classes of algorithms, but very poorly for other classes. It is very desirable to develop a reconfigurable multiprocessor system, that can be dynamically tuned to match with the special requirements of different application domains at different times. Such a general-purpose supercomputer may demand very costly

This research was supported in part, by research grant AFOSR-84-0385 and in part by Fujitsu Limited. From January 1985, the authors will be with Computer Research Institute, Dept. of Electrical Engineering, University of Southern California, Los Angeles, Calif. 90089. All rights are reserved by the authors, October 1984. A preliminary version of this paper was submitted for presentation at the *12th International Annual Symposium on Computer Architecture,* Boston, Mass., June 17-19, 1985.

hardware and software. However, its application flexibility and system performance will improve significantly. This paper proposes a dynamic network approach to developing very flexible processors for use in a multiprocessor system. Two design methodologies are developed based on *dynamic systolization* and *macro dataflowing.* These two design methodologies can be applied to many exploratory multiprocessor systems with prespecified performance levels. The Remps architecture is an integrated system consisting of multiple processors (each with multiple PEs), shared memory, and fast I/O facilities. The system has absorbed many of the attractive features from the Cray X-MP [1], the Denelcor HEP [2], the S-1 project [3], the Trac [4, 5], the NYU ultracomputer [6, 7], the blue Chip project [8], the Pringle [30], the SAM project [9], the Pumps [10], the Cedar [11], and several dataflow projects [12-16].

The Remps architecture is generalized from the aforementioned systems as a multiprocessor research model for developing design methodologies, multiprocessing and multitasking supports, application adaptability, and dynamic reconfiguration techniques. We are aimed at achieving high throughput with dynamic multiconfigurations, high availability by graceful degradation, and high performance with reduced development overhead.

The Remps exploits high-level, multitasking among communicating tasks with crude granularity based on the macro dataflow concept [17-19]; and the intratask parallelism using "dynamic" systolic/wavefront array processors, which are extended from their "static" counterparts [20-22]. The Remps differs from the Cedar in that hardware supports are provided for multitasking and intertask communications. The dynamic systolization concept is inspired by the Chip project [8], the PSC project [23], the multipipeline chaining in Cray X-MP [1], the restructurable computer [24], the expression processor [25], and the Pringle [30].

Static systolic arrays [20] or static wavefront processors [22] are dedicated for fixed algorithms. For special-purpose applications, such static hardware accelerators do perform very well, if the problems are compute-bound (rather I/O bound). Snyder's Chip project proposed to build dynamic systolic arrays using programmable switch lattices [8]. The reconfiguration capability of a switch lattice is limited by the capacity of the switch memory, the complexity of the lattice control, and the overhead associated with reconfiguration. The switch lattice must maintain local connectivity among *processing elements* (PEs) within short distance. We propose a network approach to implementing dynamic systolic arrays, which are not restricted by the local connectivities among PEs.

Such a *multi-PE systolization* approach demands the use of a dynamically reconfigurable, interconnection network among the PEs. A reconfigurable, packet switched network will be used for a dynamic wavefront processor. A dynamic systolic or wavefront processor should be able to execute different compound functions and algorithms with variable granularity [26]. It is of fundamental importance to provide dynamic hardware supports for macro-dataflow multitasking computations. The obvious advantage lies in significantly increased flexibility and adaptability for general-purpose, scientific applications. However, the gain may be overshadowed by the increased array reconfiguration overhead. We shall address these tradeoff issues and prove that the dynamic systolic approach is indeed plausible with state-of-the-art microelectronics technologies [55], [56], [59].

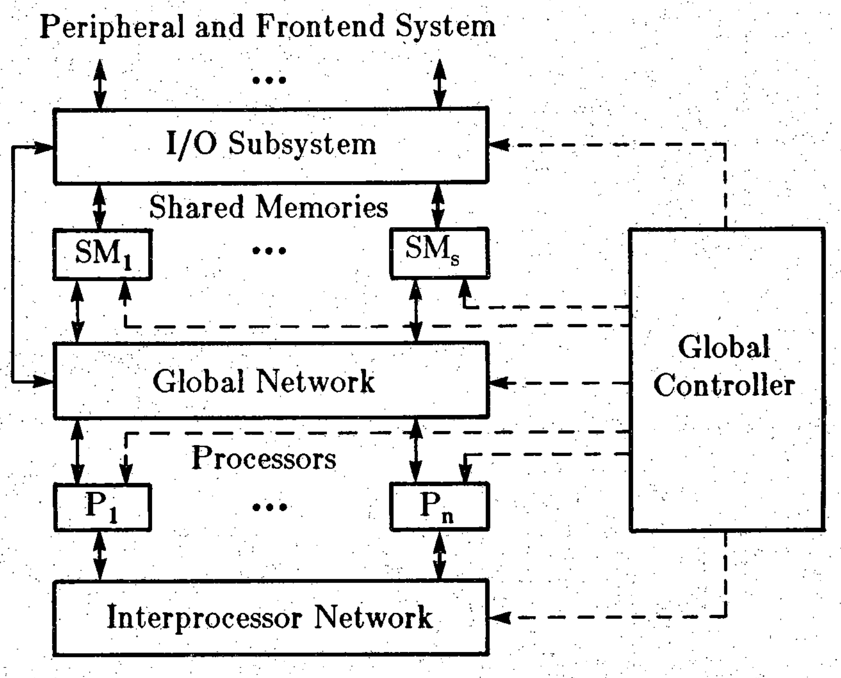
We present the Remps architecture and its reconfigurability in Section 2. The concept of dynamic systolic arrays and dynamic wavefront processors are introduced in Section 3. Then, we present in Section 4 the systolization methods and networking requirements of multiple PEs in each processor. In Section 5, multitasking among multiprocessors for macro-dataflow in Remps is described. Illustrative example algorithms and performance analysis are given in Section 6. Finally, we indicate the future research demand and the potential applications of Remps for numerical scientific computations and/or for symbolic AI-oriented applications.

Our studies intend to complement many of the on-going multiprocessor research projects. The proposed design methodologies, functional mechanisms, and communication networks, once completely developed, should be useful to computer designers, who are developing their own supercomputer systems or evaluating commercially acquired systems for specific applications.

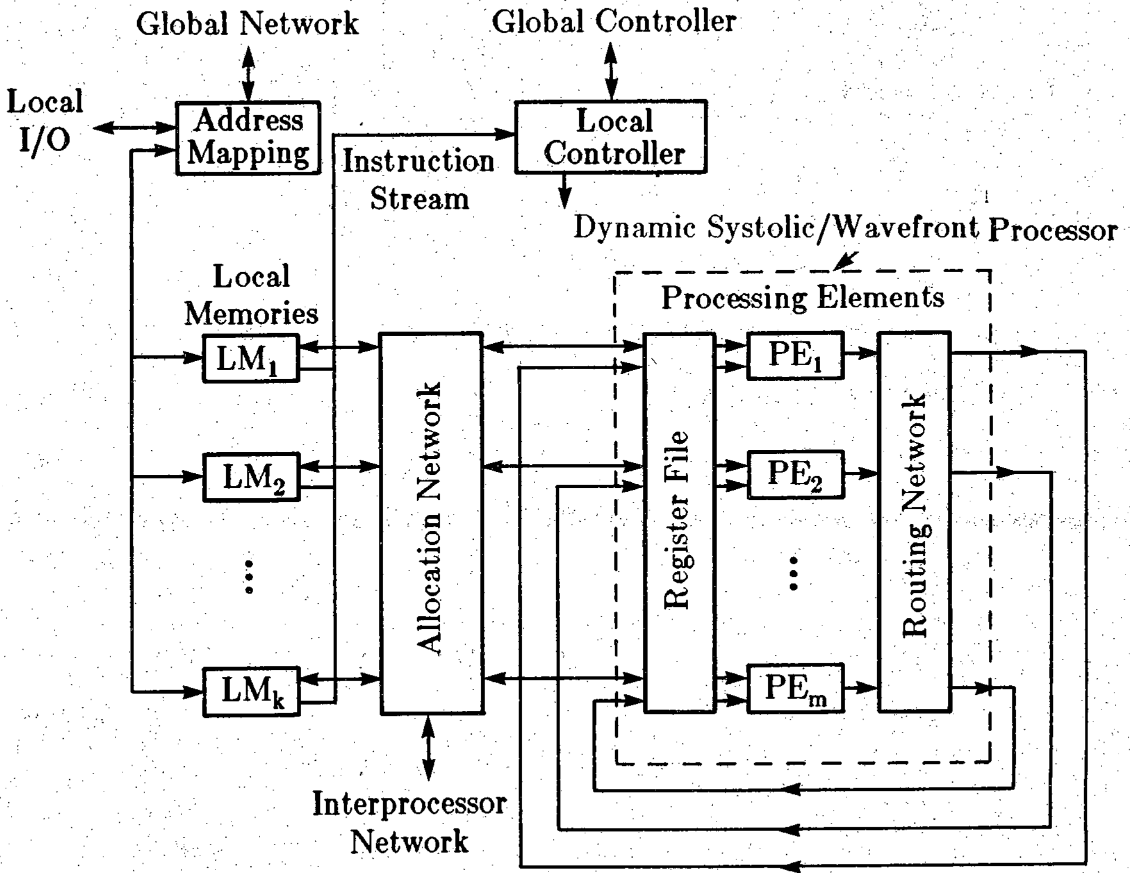
2. The Remps Architecture and Reconfigurability

The *Remps* is an MIMD computer with a three-level hierarchical structure, as depicted in Fig. 1. An $n \times m$ configuration of Remps has n identical processors, which are capable of exploiting high-level parallelisms among communicating tasks. Each processor has m pipelined PEs, for executing low-level parallelisms among individual instructions. At the global level, the machine is an event-driven, data-flow computer with token storage; while at the lower level, each processor is a multipipeline control-flow system. The system uses several interconnection networks to achieve flexibility in implementing a wide range of computation tasks.

System components of Remps are functionally described below. The shared memory consists of multiple modules, each module can be used to store a number of tasks for execution. A small portion of each module can implement an I-structure storage [13], where global data can be shared by several communicating tasks on different processors. A global network is used to interface the global memory with the processors. Tasks may be assigned to multiple processors for concurrent processing. Each processor has a local memory, some local I/O devices, a large register file, and multiple PEs that can operate in parallel. Note that the PEs are identical, each being



(a) The global structure of the Remps



(b) The detailed structure of each processor

Figure 1. The system architecture of the Remps

multifunctional, and they do not have to operate in a lock-step manner. Different functions can be performed at different PEs at the same time, similar to those parallel functional pipelines used in IBM 360/91, CDC 7600, or in Cray X-MP [27].

The I/O subsystem is connected to the shared memories for the input/output of large programs, data sets, and result sets. For small jobs, these I/O activities can be also directly handled by the local I/O facilities attached to each processor. The interprocessor network provides direct communication paths and buffers between processors. Shared data registers and special semaphore registers are contained in this network, so that interface buffers can be established to support multitasking in multiple processors. This kind of interprocessor network has been implemented in the Cray X-MP series for the same purposes. The global controller is responsible for task scheduling, memory management, multiprogramming, synchronization, and other functions at the global level.

Each processor is itself a high-speed, reconfigurable computer of the control-flow type. The interior structure of the processor is illustrated in Fig. 1b. This structure is generalized from Cray X-MP and NEC SX-2, among other multipipeline supercomputers [28]. Each PE is a functional pipeline which is capable of performing many arithmetic/logic operations of different data formats. The program codes and data sets are stored in the local memory. The register files can be dynamically allocated to the PEs. Multiple data streams are allowed to flow between the local memory and the PEs through an allocation network. The PEs can be interconnected as a systolic array processor or as a wavefront processor via the routing network. Each processor has a local controller, which must communicate with the global controller for multitask scheduling.

Static and unifunction pipelines are implemented as PEs in most of today's computers with multiple functional units. Such fixed function pipelines have two major shortcomings:

(a) Low pipeline utilization may exist due to poor matching of resource demands and low PE availability.

(b) Every pipeline becomes a critical resource due to no duplications. This renders the system to be prone to catastrophic failure.

By using homogeneous, multifunctional pipelines as PEs, the above problems can be greatly alleviated or eliminated. In the Remps, the PEs in each processor are identical and universal in their functional capability. At most m independent operations can be executed by m PEs in each processor simultaneously, resulting in a full utilization of the functional resources. Since a failing PE can be replaced by other PEs, graceful degradation is possible. By adding some handshaking mechanism into each PE, the *dynamic systolic array* (the dash-line box in Fig. 1b) can be modified to become a *dynamic wavefront processor,* extended from the static wavefront arrays suggested in Kung [22]. In this case, the routing network must be pipelined and packet switched as suggested in [29].

In static systolic arrays, only boundary PEs communicate with the memory as I/O interfaces. In our dynamic approach, different PEs may be used for I/O in differently connected topologies. The allocation network is responsible to interface between the local memory and the PE registers being assigned for I/O operations. Data can be also manipulated on-the-fly via the allocation network. The entire sequence of operations from the memory to allocation network, the register files, the input PEs, the routing network, the interior PEs, the output PEs, the register file, the allocation network, and back to the local memory form a *macropipeline*. Masked memory accesses and masked PE operations are also possible. Different systolic or wavefront arrays can be implemented in different processors when the system is used in MIMD multitasking mode or in multiple SISD mode.

The Remps is being designed to realize data-driven computations at the task level. A compiler is needed to partition a large job into a number of communicating tasks, to be processed by multiple processors. The task dependence graph, formed by crude granularity at compile time, is transmitted to the global controller for use in scheduling. An executable task queue is established for each job, which contains all the executable tasks. The global controller assigns an executable task to one or more processors whenever the resource becomes available.

Once a task is assigned to a processor, the processor assumes exclusive control of the execution. After a task is executed, the bulk of the results is stored back to the shared memory. Scalar results or semaphore messages can be passed directly among the processors through the interprocessor network. In a multiprogramming environment, the execution queues of several jobs can be combined into a single queue. Whenever there is an idle processor, the global controller will check this queue to initiate a new task.

Difficulties may arise for macro dataflow, when two or more tasks must communicate with each other before they finish. The Remps overcomes this problem by employing the I-structure storages, low-level context switches, and the interprocessor network. The shared data such as global variables can be stored in the I-structure residing in the shared memory. Status tags are associated with each shared data item in order to resolve conflicts. Special synchronization semaphores and interrupt signals are passed directly among the processors via the interprocessor network.

If the shared data are individual variables, each task will have a duplicated copy of the variables in the local memory. When the task requests to access a shared variable, it will check with the local copy. If the request is granted by the status tags, the tasks will continue. In case the request is to *write* a shared variable in the global memory which is locked by another task, the requesting task will be suspended until the status tags change. A processor level context switching is then performed and another ready-to-run task is initialized for execution. Shared data in large blocks (such as an array or a tree), are only stored in the global memory. Any task accessing these data blocks must have exclusive control of the access until completion.

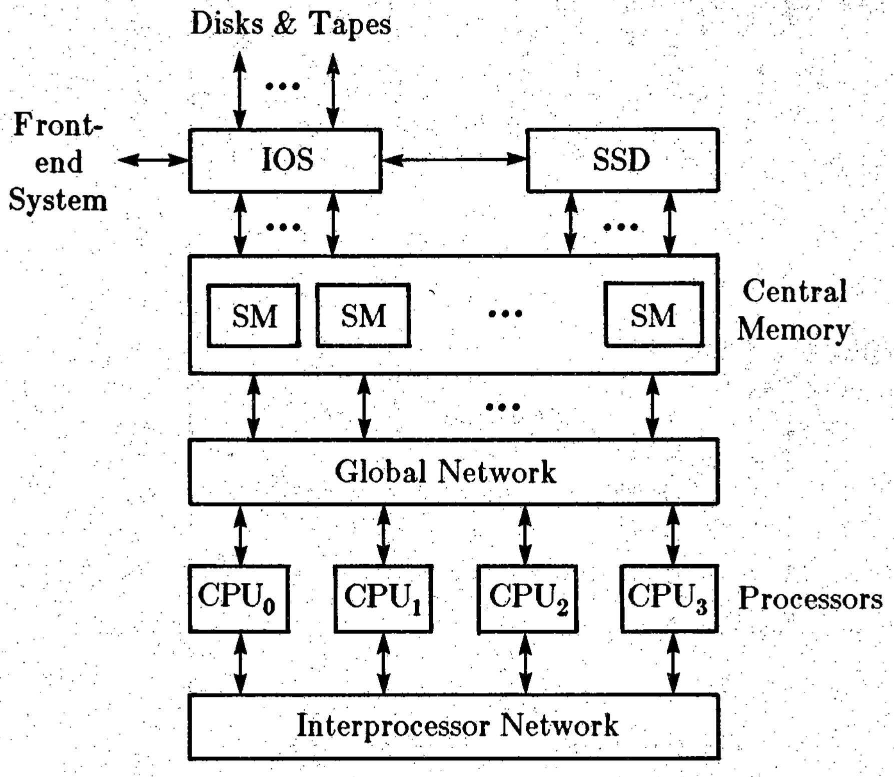
Reconfigurability is supported by the hierarchical control and the extensive use of pipelines and interconnection networks. This feature greatly enhances the system reliability, application flexibility, and availability of the system. The system can be reconfigured to meet different application demands and environmental changes. Two Remps reconfigurations are shown in Fig. 2 in order to emulate the Cray X-MP-4 and the Denelcor HEP.

In part (a), the shared memories become the central memory in X-MP [1, 28, 46, 56]. The Solid-state Storage Device (SSD) in X-MP is connected to both the central memory and the I/O subsystem. The interprocessor network now serves the same function as the inter-CPU communication and control unit in X-MP. Each CPU now has $m=13$ functional pipelines (PEs). In part (b), the global network is combined with the interprocessor network to form a single packet-switched network between the processors and the data memory modules in HEP [2, 28]. The I/O and peripheral devices are connected to this combined network.

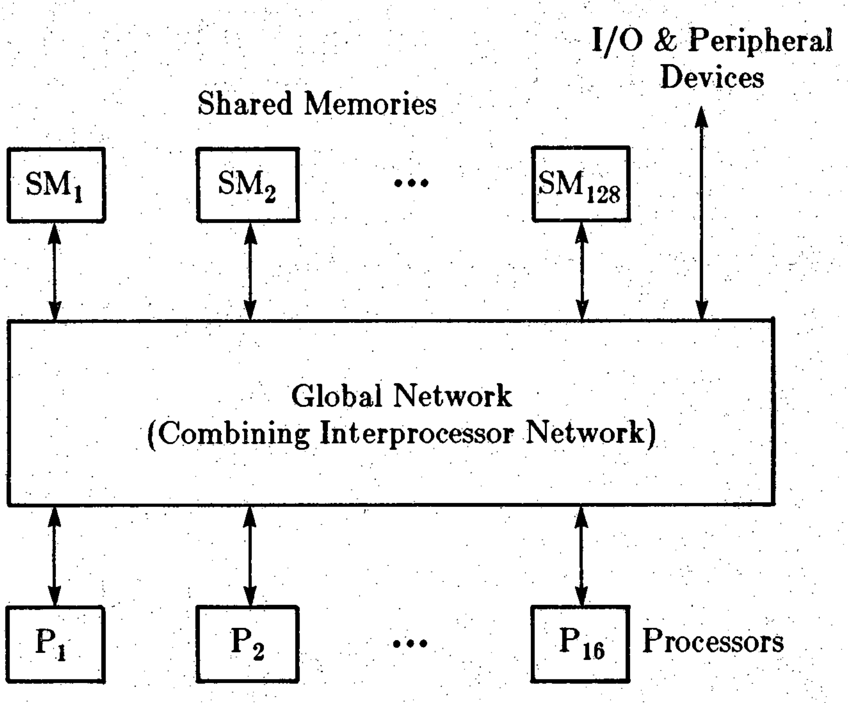
The Remps can be used to model a number of real supercomputers. Our intention is to provide a general model to help develop future supercomputers. The emulation modeling can help detect the system bottlenecks, determine the major shortcomings in existing machines, and predicate the performance of future machines. Such high-level emulation studies will significantly reduce the development overhead and, in many cases, totally eliminate unnecessary wrong undertakings. In the context of performance scalability, the Remps should increase its computing power linearly with the increase of the processor number (n) and the number of PEs (m) per processor.

3. Dynamic Systolic/Wavefront Array Processors

A network approach to performing dynamic, systolic or wavefront computations is proposed for fast execution of compound functions or algorithms with crude granularity. Different PEs are allowed to perform different functions at the same time. Multiple PEs in each processor can be interconnected to form any systolic arrays or wavefront arrays using a dynamically reconfigurable network as shown in Fig. 3a. This approach allows the system to reconfigure the pool of PEs and the network resource into different systolic arrays at different times. The connectivity among the PEs is governed by the interconnection provided by the routing network. Note that this approach is not restricted by local connectivity or uniform PE functions, as required in the static systolic arrays [20], [51], [52], or in the programmable switch-lattices [8], [30].



(a) Emulating Cray X-MP-4



(b) Emulating HEP using Remps

Figure 2. The reconfigurations of Remps for emulating X-MP-4 and HEP

In general, each PE has p inputs and q outputs as specified in Fig. 3a. The size of the network in Fig. 3b is $qm \times pm$, where m is the number of PEs in a processor. For illustrative purposes, we can assume $p = q = 3$ in Fig. 4, which shows how to construct a systolic tree using 7 PEs and a crossbar network. The solid lines correspond to top-down connections and dash lines for bottom-up connections. This allows one to traverse the tree machine in either direction.

In Fig. 5, we show a linearly-connected systolic ring [31] for solving triangular linear system of equations. Each PE in this case has 3 inputs and 2 outputs as specified in Fig. 5a. The 4 PEs can be interconnected using a data manipulator [32]. Another example is shown in Fig. 6. In this case, a hexagonal systolic array is implemented using a Benes network [33].

In order to implement any array topology, the crossbar network offers the highest connectivity. Due to the high cost of crossbar, one can also consider a number of other network classes, such as the Omega network, binary n-cube, data manipulators, flip network, regular S-W banyan, Benes networks, and baseline networks and their derivatives [34]. In order to systolize an array of PEs, the ***cut-set theorem*** [22] and the ***local correctness criterion*** [31] must be satisfied. Furthermore, the network must be designed to resolve conflicts and to allow pipelined operations, which must be synchronized with the PE operations.

What we proposed here is to separate the interconnections of PEs from the collection of PEs. This approach is very different from most of the existing systolic approaches. Various systolic/wavefront array processors are summarized in Table 1. The advantages of this dynamic-network-approach are listed below:

(a) Many of the state-of-the-art research results on interconnection networks can be applied. Instead of using multiple static systolic arrays, only a single dynamic routing network is needed.

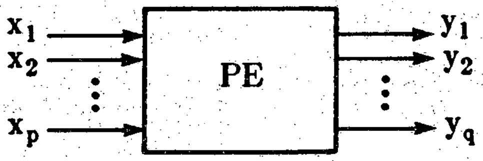
(b) Reconfigurable systolic arrays or wavefront processors can support wider applications and provide better match between the memory bandwidth and the throughput of the array processor.

(c) The separation of PEs from their interconnections offers some freedom in their VLSI/WSI implementations. Of course, the interconnections between the two parts should be minimized.

Of course, there are many other problems yet to be solved before we can claim that this network approach is more cost-effective for systolic/wavefront computing. The major problems include the network structures and control schemes, the reconfiguration overhead, the I/O port multiplexing, and the pipelined dataflow through the network. A thorough study is needed to determine the PE functionality and to select an efficient network, in order to cover a wide range of scientific/engineering applications.

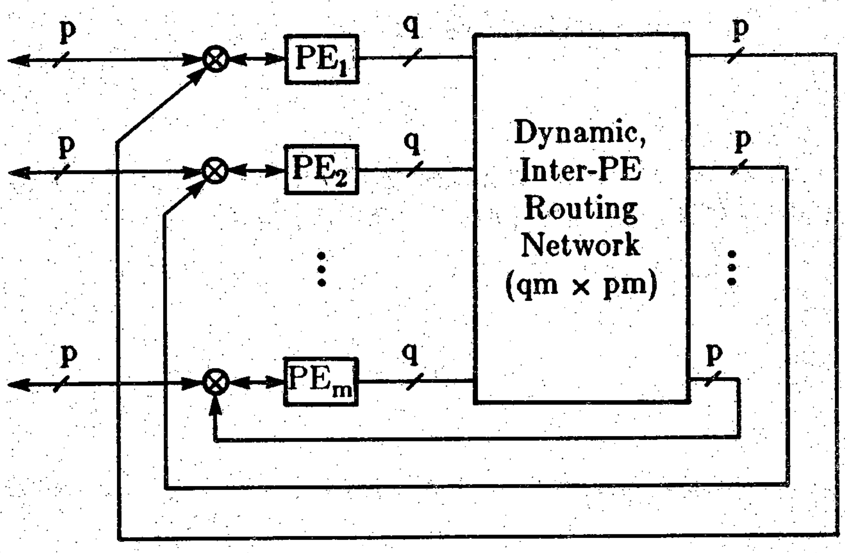
Table 1. Various Systolic/Wavefront Processors

Systolic Arrays (using synchronous PEs with a common clock control)	**Wavefront Arrays** (using handshaking PEs with data-driven operations)
Static systolic arrays: • Fixed interconnection among PEs with fixed function (Kung and Leiserson [51]) • Fixed array structure using programmable PEs (the PSC project at CMU [52]).	*Static wavefront arrays:* • Fixed array structure with wavefront propagation driven by data availability through handshaking among PEs (S. Y. Kung [22]).
Dynamic systolic arrays: • Reconfigurable switch lattice of programmable PEs (The Chip project by Snyder, et al. [8], [30]). • Programmable systolic array using multifunctional PEs which are dynamically interconnected by a circuit-switched network (Hwang and Xu [26]).	*Dynamic wavefront arrays:* • A wavefront array of multifunction PEs, which are dynamically interconnected by a packet-switched network (Hwang and Xu [26]). • Attractive for arbitrarily structured parallelism in user algorithms.



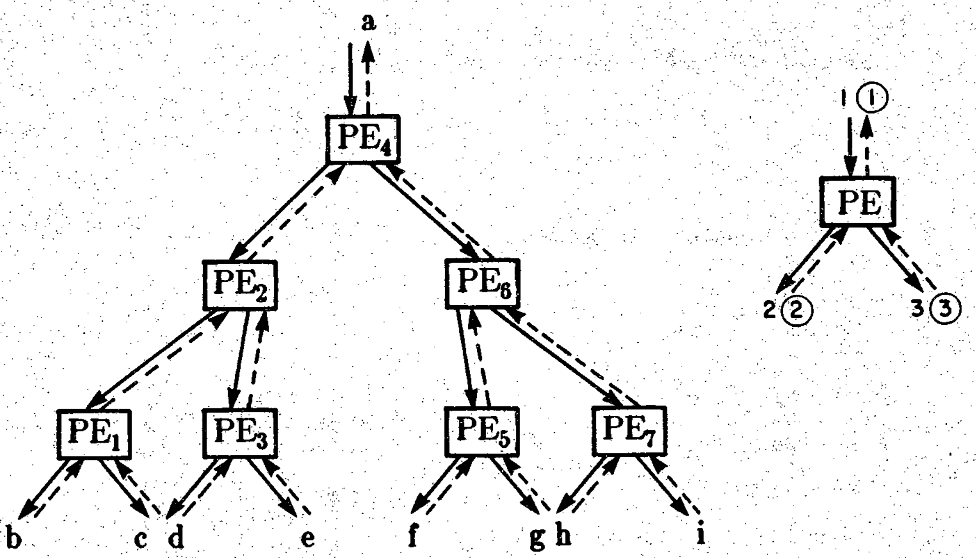
$$\begin{cases} y_1 = f_1(x_1, x_2, \cdots, x_p) \\ y_2 = f_2(x_1, x_2, \cdots, x_p) \\ \vdots \\ y_q = f_q(x_1, x_2, \cdots, x_p) \end{cases}$$

(a) A processing element (functional pipelines)

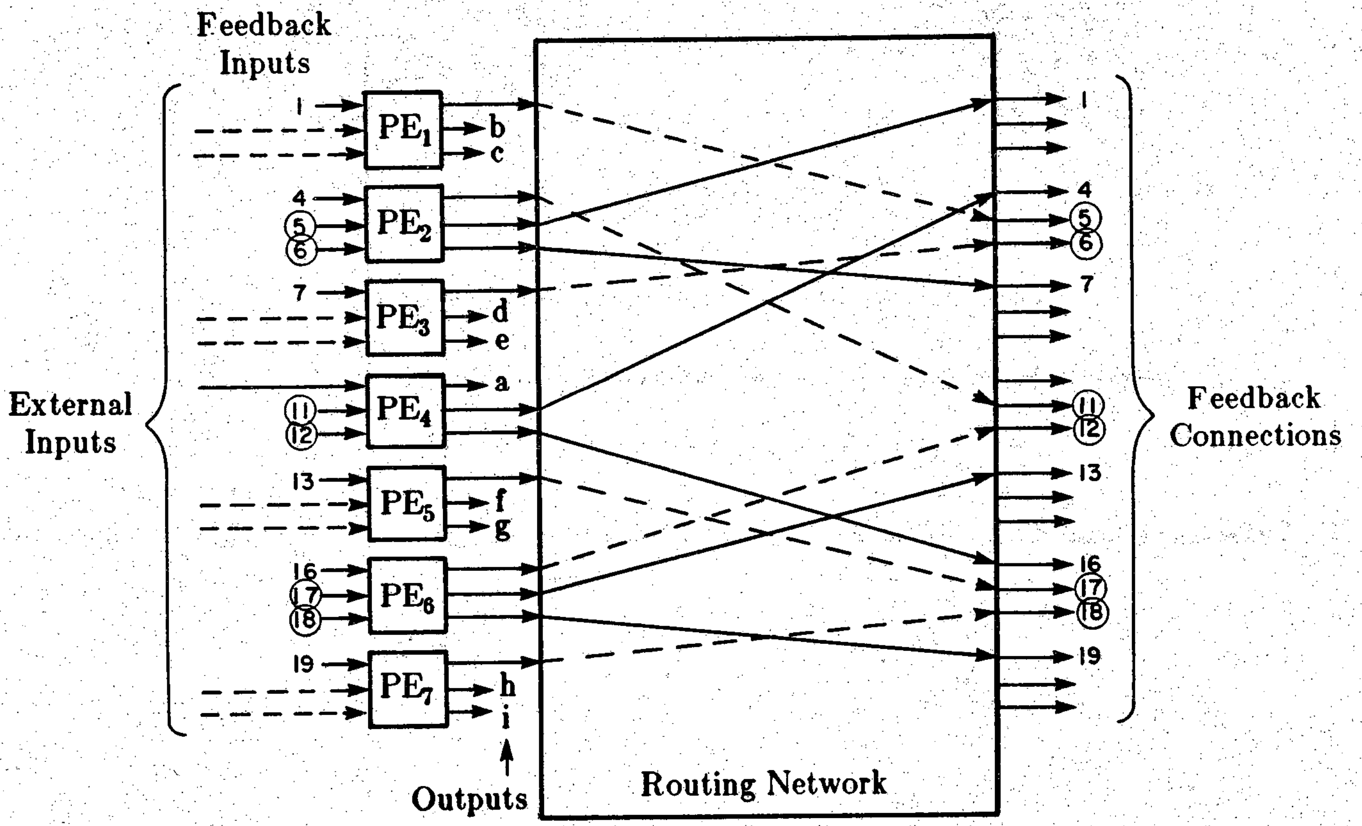


(b) Multi-PE networking

Figure 3. The concept of multi-PE networking for the construction of dynamic systolic arrays (p and q refer to the numbers of I/O ports per PE)

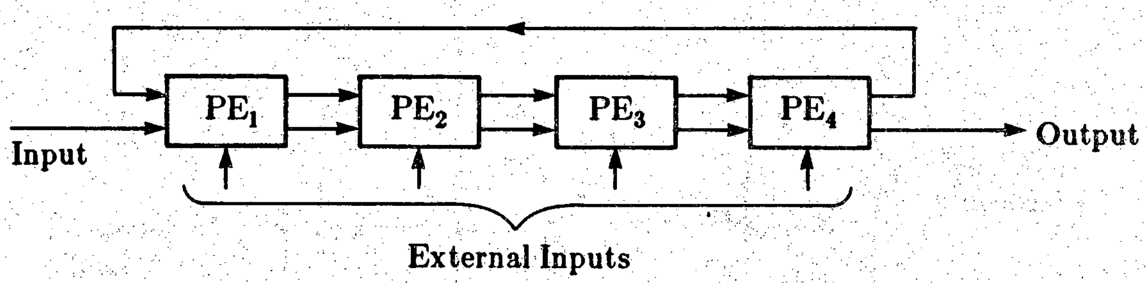


(a) A systolic tree machine for searching operations

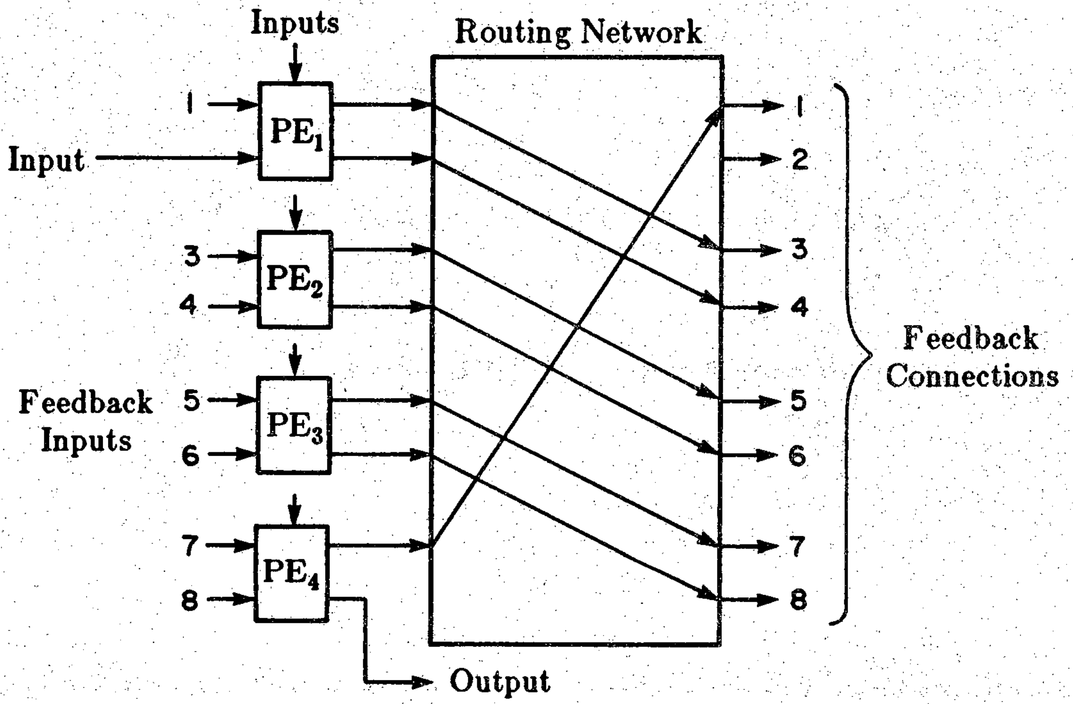


(b) The network interconnections for the systolic tree

Figure 4. A systolic search tree and the multi-PE network implementation

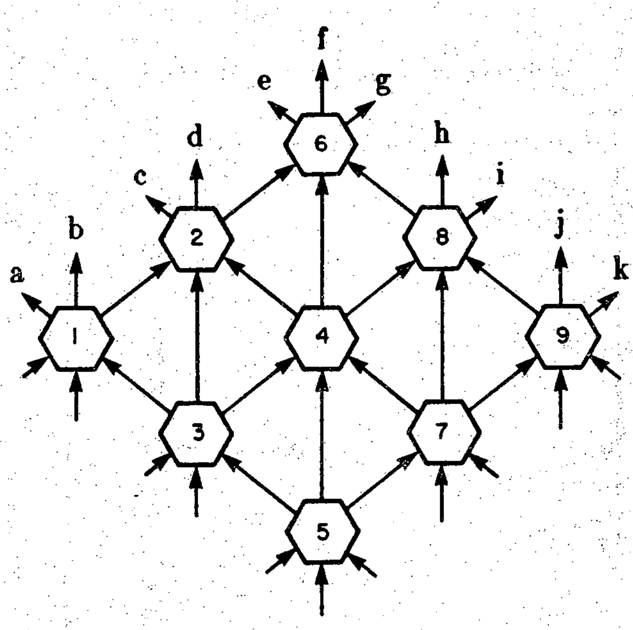


(a) A systolic ring of 4 PEs

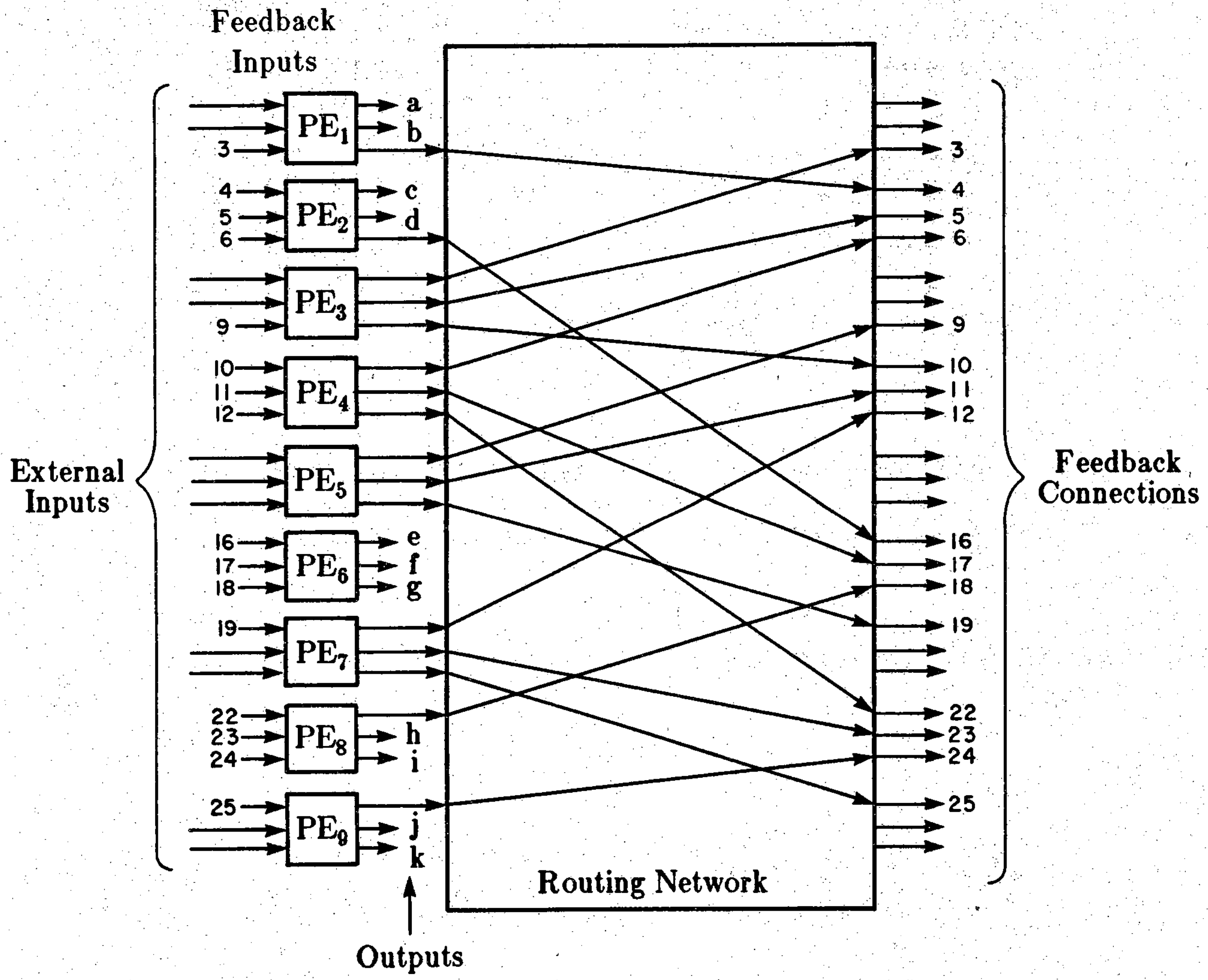


(b) The network implementation of the systolic ring

Figure 5. A systolic ring for solving triangular linear system



(a) A hexagonal systolic array



(b) The network implementation of the hexagonal array

Figure 6. A hexagonal systolic array for matrix multiplication

Two additional overheads are introduced with dynamic systolization: the *reconfiguration setup time* and the *network delays*. Whenever a systolic computation is to be performed, the routing network must be set up to create the desired systolic array. This setup time in most cases is dominated by the control complexity of the routing network. The problem of matching the problem size with the size of the systolic array is less severe in a dynamic processor. Multiple small arrays can coexist in one processor. In Remps, several systolic arrays in different processors can be interconnected via the shared registers to form a larger systolic computing structure. Our study is concentrated on multi-PE systolization within each processor. Only after we gain enough experience with intraprocessor systolization, then we shall challenge interprocessor systolization.

Design of the PE needs to cover the most frequently used scientific operators. Besides functional selection for the PEs, one must be concerned about the PE reconfiguration control and the I/O ports requirements. The construction of programmable PEs is desired [22], [23]. Design of packet-switched networks is needed for inter-PE connections in a dynamic wavefront array processor. Systolization experiments on the routing network demand the simulation of the connectivities and assessing the network delays. Other studies include the dynamic PE allocation, the linkage across processor boundaries, timing analysis, overhead and performance analysis on various systolic arrays.

4. Systolization and Networking Requirements

The design of a dynamic systolic/wavefront array processor involves three basic considerations: the structure of the PEs, the routing network, and the systolization procedures. These design considerations are discussed below. Because all PEs are identical, they must be multifunctional and can handle variable granularities. Register files should be allocated to each PE. Besides the primitive arithmetic/logic functions, each PE may have such functions as *dot product, compare and exchange,* and even *butterfly* operations used in FFT. Rescaling the clock rate is needed in order to match the PE processing rate with the network data transfer rate. Interface registers (or latches) must be used between the PEs and I/O ports of various networks involved. This is necessary to achieve pipelining with synchronous control in a systolic processor. For a wavefront processor, the PEs should also have some handshaking mechanism to support the data flow operations [12, 22].

Systolization procedures include: (i) the transformation of a dataflow graph or a signal flow graph to a systolic array; and (ii) the setup of a systolic array by specifying the function of each PE and establishing the desired connection pattern in the routing network. Systematically mapping parallel algorithms into systolic array structure is very desirable. Up to now, many algorithms have been proven systolizable [20]. Systematic synthesis procedures have been suggested in [35], [36]. However, the problem is still open for general systolic arrays. With our dynamic systolic approach, not only regularly-structured arrays (such as ring, tree, hexagonal, square, linear, triangular arrays) but also irregular arrays (such as any partially-ordered graphs) are implementable. Thus the array synthesis procedures should be more generalized and easier to be automated.

Although the dynamic systolization removes the ***spatial locality*** constraint, which is inherent with static systolic arrays, ***temporal locality*** is still required for dynamic systolic arrays. The local correctness property suggested in [31] is sufficient to guarantee the global delay correctness in static systolic arrays. This criterion also holds for dynamic systolic arrays. Furthermore, if the dynamic array is feedback-free, the systolization procedures (i.e., the cut-set rules) in [22] are directly applicable. Even if the dynamic array has feedback connections, the procedures can still be effective after only some minor modifications. Programmable noncompute delays should be associated with the routing network in order to add necessary delays among PEs, so that the ***local delay criterion*** is satisfied. The LINK chip reported in [21] is an interconnection chip, which can have up to 32 units of programmable delays between an input and an output, which covers most frequently-used computation structures.

With respect to the dynamic wavefront arrays, the above systolization problems can be greatly alleviated. This is due to the fact that both the spatial locality and the temporal locality are no longer needed. Theoretically, any dataflow graph or signal flow graph can be directly transformed into a wavefront array for execution, if enough handshaking registers are associated with the PEs [22]. Nevertheless, since the PEs cannot have infinite (or even large) number of handshaking registers, some workable procedure is still needed to transform the algorithm structure into a practical wavefront array.

The inter-PE routing network should offer nonblocking and a high degree of connectivity with a low hardware complexity, 0(m log m), a short network delay, 0(log m), and simple control complexity, 0(log m) or even 0(1), where m is the network size. Of course, when m is not excessively large, the crossbar network offers the most flexibility in interconnecting the PEs. Among all the candidate networks, our initial finding shows that the modified Benes networks may be the most promising type for a large scale system and the crossbar network is preferable for small systems.

Other networks issues worthy of further study include: the dynamic connectivity, the control strategies, the localization of broadcast operations, the reconfiguration topologies, and the modular construction of the networks. The purpose is to reduce the network delay and increase the bandwidth. Several inter-PE connection networks are examined below for systolization purposes. These networks are characterized by their *connectivity, blocking or nonblocking, hardware demand, network delay,* and *control complexity.* Various network properties are summarized in Table 2.

Crossbar Networks:

Crossbar switching networks are nonblocking and thus have full connectivity. The network delay may be either $0(m)$ or $0(\log m)$ depending on crossbar implementation schemes. The major disadvantage of a crossbar network is its high hardware demand $0(m^2)$, especially when the PE number m is very large. For small systems with less than 16 PEs per processor, crossbar switch is acceptable based on today's technology. In the LINK chip [21], an 8×8 crossbar network with a 4-bit wordlength was implemented. Sixty four such chips can be interconnected to form a 16×16 crossbar with a 64-bit wordlength.

When m is large, Franklin [37] has observed that a crossbar or a Banyan network can be implemented with a VLSI chip in an area of $0(m^2)$. For VLSI implementation, the gate complexity is not so severe an obstacle as compared with the pinout limitation. The Snyder's Chip machine [8] requires $0(m^2)$ switches for implementing all planar graphs with an $0(m)$ lattice delay. Thus crossbar should be at least as good as the switch lattices for general purpose applications, as far as hardware demand and network delay are concerned.

Benes Networks:

The Benes network is rearrangeable and thus has full connectivity. However, the network is blocking with a delay of $0(\log m)$ and a hardware demand of $0(m \log m)$. These are attractive features for systolization. Unfortunately, the control complexity for Benes network is $0(m \log m)$, which is not desirable. Nassimi and Sahni [38] have developed a self-routing scheme with control complexity 0(1), which sets up the network on-the-fly for a large number of permutations. It is still an open problem to determine whether Benes networks can provide all useful connection patterns in 0(1) time.

Other Blocking Networks:

This class of networks are not rearrangeable and thus have limited connectivity [34]. The modified data manipulator, flip networks, Omega networks, indirect binary n-cute, regular SW Banyan, and baseline networks are all belonging to this class. The network delay is $0(\log m)$ and the hardware demand is $0(m \log m)$. It is doubtful to utilize the limited connectivity in providing arbitrary connections. The major advantage of using this class is its low network control complexity 0(1).

Table 2. Various $m \times m$ Networks for the Construction of Dynamic Systolic/Wavefront Arrays

Network Class	Network Properties			
	Connectivity	Hardware Demand	Control Complexity	Network Delay
Crossbar Switch [21,37]	Nonblocking	$0(m^2)$	$0(1)$	$0(m)$ or $0(\log m)$
Benes [33], [38]	Rearrangeable	$0(m \log m)$	$0(m \log m)$ or $0(1)$	$0(\log m)$
Blocking Networks* [34]	Not rearrangeable	$0(m \log m)$	$0(1)$	$0(\log m)$
Nonblocking Networks [34], [39]	Nonblocking	$0(m \log^2 m)$	$0(\log m)$	$0(\log m)$
Packet Switching Networks [29,43]	Nonblocking	$0(m \log m)$	$0(1)$	$0(\log m)$

*(Omega, flip, n-cube, data manipulator, Banyan and Baseline).

Other Nonblocking Networks:

Besides crossbar, there are other nonblocking networks with a hardware demand of $0(m \log m)$ as assessed in [39], [40]. Unfortunately, there is no systematic method to construct such nonblocking networks, let alone the concern of the network delay and control complexity. Cantor [41] constructed a nonblocking network with hardware complexity of $0(m \log^2 m)$ and network delay of $0(\log m)$. The control complexity for such network was determined as $0(\log m)$ [42].

Packet-switched Networks:

This class includes many of the buffered multistage networks such as the buffered Delta networks [43] and multipath packet switching networks [29]. The 3-ported switches used in HEP network [2] are also an interesting design. The multipath, multistage networks require a hardware demand, $0(m \log m)$, a network delay of $0(\log m)$, and a control complexity of $0(1)$. When the traffic rate is moderate, such buffered networks become essentially nonblocking. This class of networks is necessary for the construction of dynamic wavefront array processors using the handshaking PEs.

Because of the increased overhead by network reconfiguration and transmission delays, a dynamic systolic array processor cannot be faster than a static counterpart. A tradeoff study between the increased application flexibility and the degraded array speed must be made. The performance degradation due to these extra delays is analyzed below. Let k be the number of pipeline stages in each PE, c be the length of the critical dataflow path in a systolic array, and N be the ***problem size*** which is equal to the number of operand blocks (or wavefronts) entering a systolic array. Then the total time required to process the N operand blocks in a static array is equal to:

$$T_s = k \cdot (c+1) + N-1 \tag{1}$$

where $k \cdot (c+1)$ clock periods are needed to process the very first block and the remaining $N-1$ data blocks each takes one clock period to come out of the pipeline.

Now consider the case of a dynamic systolic array with the same parameters k, c, and N. Let α be the network reconfiguration overhead, which is primarily determined by the network control complexity. The routing network has a size of $m \times m$ with $\beta = 0(\log_2 m)$ stages. If the routing network has a stage delay equal to one clock period of the PE pipeline, the network then has a transmission delay of β clock periods. This implies the total delay through the dynamic systolic array is equal to $c \cdot \beta$, because the critical path is the longest dataflow path in the array. This leads to the following total time required to process N operand blocks in a dynamic systolic array:

$$T_d = \alpha + (k+\beta) \cdot c + k + N-1 \tag{2}$$

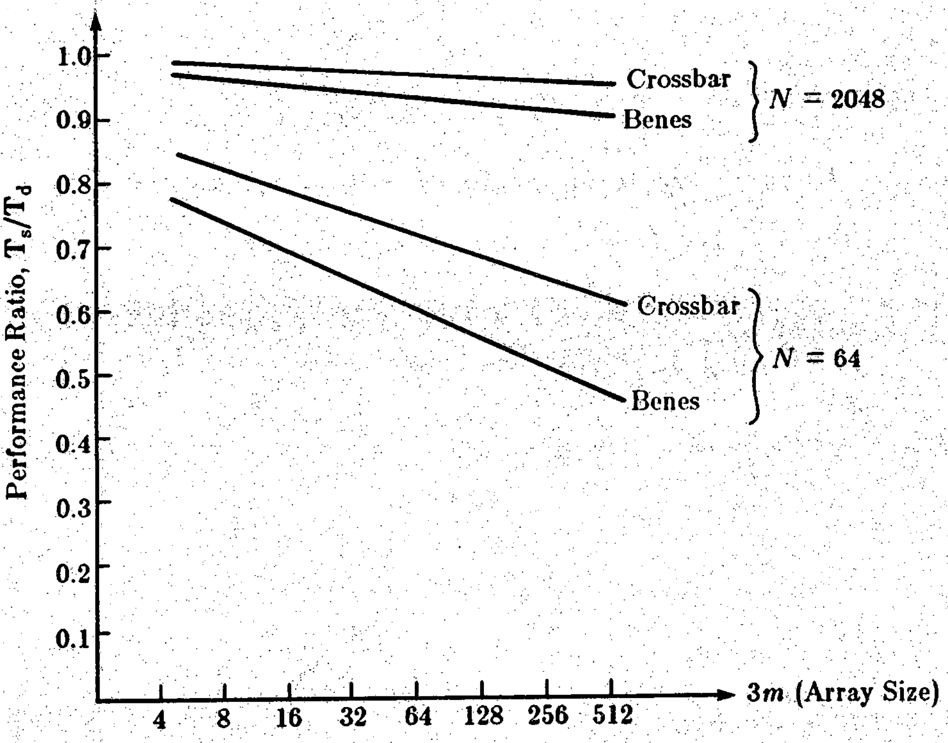
where $(k+\beta)c + k$ is the time for the first operand block to pass through the entire array and $N-1$ additional clock periods are needed to process the remaining $N-1$ blocks. For crossbar network, the overhead α equals a constant and $\beta = \log_2 m$, and for Benes network with self routing control, α is also a constant and $\beta = 2\log_2 m-1$.

The throughputs of the static and dynamic systolic arrays are $1/T_s$ and $1/T_d$ respectively. Thus $(1/T_s)/(1/T_d) = T_d/T_s$ represents the ***performance ratio*** of the dynamic versus static systolic arrays. By (1) and (2), we have $T_d/T_s \leq 1$. This performance ratio is plotted in Fig. 7 under the assumption $\alpha = 10$, $k = 8$, $c = 15$, and an array size $3m \times 3m$ (if each PE has 3 inputs and 3 outputs). The curves correspond to dynamic systolic arrays using either crossbar or Benes networks. Part (a) shows the performance ratio as a function of the array size $3m$ and Part (b) of the problem size N. When the problem size is small (say $N = 64$), both networks degrade with the increase of the array size. However, the performance ratios become fairly close to 1 when the problem size becomes very large (say $N = 2048$), regardless of the array size. With fixed network size (say $m = 1024$ as in part b), the performance ratios approach to 1 with the increase of the problem size. In both drawings, the crossbar-structured systolic array always has better performance than the array built with a Benes network.

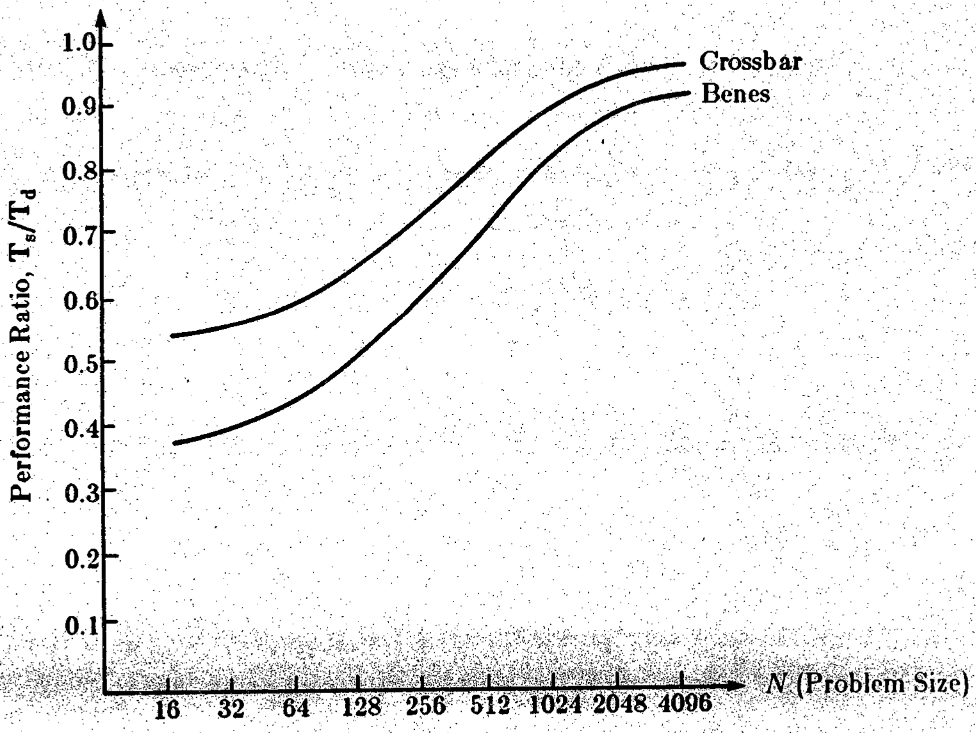
The dynamic systolic array can perform equally well as the static arrays, if the problem size is large. The smaller is the interconnection network, the better will be the performance. If the problem size is larger than the network size, then the dynamic systolic arrays can maintain 60~90% the performance of the static counterparts. For general-purpose applications, the flexibility in implementing any systolic algorithms is far more important than a minor speed degradation due to added network overhead. There is no doubt that static systolic arrays are still superior for implementing fixed algorithms. However, static arrays become useless when the application algorithms are changed.

5. Multitasking for Macro-Dataflowing

The potential problems of instruction-level, data-driven machines have been discussed in [44]. The major drawback is the excessive pipeline overhead per instruction. The task-level or macro dataflow will have much cruder granularities. It is fair to view the Remps as a ***macro dataflow*** machine with token storage, where a ***token*** represents a task and an ***operator*** corresponds to a processor. This architecture combines the advantages of both macro dataflowing at the global level and control-flowing at the processor level. Data-sharing among communicating tasks is implemented either with I-structure storage at the global memory or via the shared registers between processors.



(a) Performance vs. the size of systolic array



(b) Performance vs. problem size with a fixed array size $m = 1024$

Figure 7. Relative performance of static versus dynamic systolic arrays.

Macro dataflow can be implemented with macro pipelining at two different levels. Tasks with crude granularity are handled at the global level. Tasks with refined granularity are handled at the processor level or even at the PE level. Thus variable-resolution tasks can be executed at different clusters of PEs and/or processors at the same time. This macro dataflowing with variable task granularities must be supported by the hierarchical controls, the use of the multifunction PEs, and the interconnection capability provided by various networks in Remps.

Gaudiot and Ercegovac [18] have conducted an experimental performance evaluation of dataflow computers using variable-resolution actors. The machine model with which they have simulated is similar to the Arvind's dataflow machine at MIT [13]. They use a variable number of processing elements to handle various node granularities. Arvind's model does not have a hierarchical control as existent in the Remps. This shows the fundamental difference between our approach and that in [18]. The two approaches in fact complement each other in many aspects.

Example 1 (Multitasking of A DO Loop with IF statement)

Consider the dataflow graph in Fig. 8 containing an *IF* statement and the ***vector merge*** operation through masking. After vectorization, fine operations in the graph can be lumped together to yield the graph with crude granularity. If we use 3 PEs from Processor P_1 and 5 PEs from Processor P_2, two multipipeline systolic arrays are formed, respectively. The Boolean vector, X, generated by processor 1, can be transmitted to processor 2 through the shared registers in the interprocessor network.

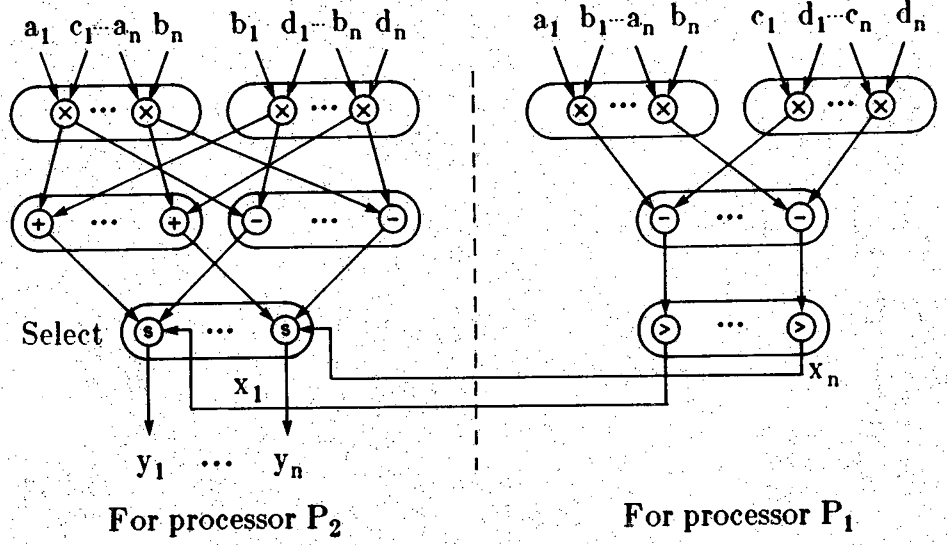
Note that pipeline chaining in Cray X-MP implements only linear data flow graphs. The proposed ***multitasking*** technique can implement any program graphs. Of course, the communication of systolic arrays from different processors must be supported by the interprocessor network. These networks should be designed to have high bandwidth, low transmission delays, and some data manipulation capabilities.

Advantages of the above approach to achieve macro dataflowing among multipipelines and multiprocessors are summarized below:

(i) The application flexibility will be greatly enhanced. Memory latency problem in handling intermediate results can be alleviated by using shared registers and packet switched networks.

(ii) The scheduling of vector instructions to pipelines and multiple tasks to processors is controlled at the local and the global levels, respectively. Thus higher resource utilization can be expected.

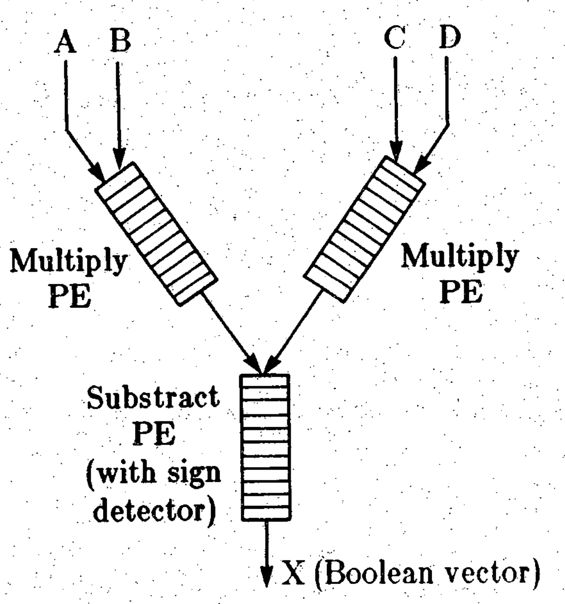
(iii) The system reconfiguration is supported by the flexible connectivities among hardware resources. Parallelism can be exploited at both task and instruction



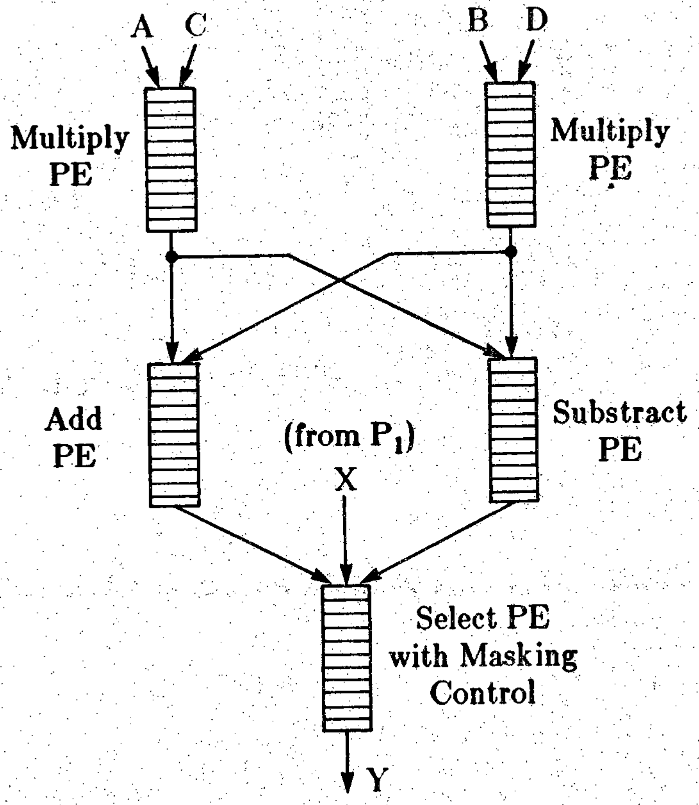
a_1 $c_1 \cdots a_n$ b_n b_1 $d_1 \cdots b_n$ d_n
Select
x_1
y_1 ⋯ y_n
For processor P_2
a_1 $b_1 \cdots a_n$ b_n c_1 $d_1 \cdots c_n$ d_n
x_n
For processor P_1

```
For i = 1 to n Do
    If $x_i = a_ic_i - b_id_i < 0$
    then $y_i = a_ib_i + c_id_i$,
    else $y_i + a_ib_i - c_id_i$
Repeat
```

(a) The dataflow graph and grouping of nodes for multitasking



(b) Systolic array implemented in P_1



(c) Systolic array implemented in P_2

Figure 8. Multitasking and systolization of multiple PEs in two processors *(Example 1)*

levels.

Multitasking for macro dataflow demands the partitioning of program graphs. This can be done by users at algorithm design time, using a language which can specify parallelism and indicate data communications between separated tasks. The partitioning can be also performed by compilers, such as the Parafrase [45]. Finally, the partitioning can be done at run time using sophisticated parallelism-detection schemes such as those for exploiting multiple functional units in IBM 360/91 and for multitasking in Cray X-MP [46]. Compile-time methods are needed for estimating the space and time complexities of computing events within a program. The purpose is to support automatic program partitioning subject only to constraints on data dependency and resource availability. These will include program restructuring, partitioning criteria, language extensions, compile algorithms, and heuristics for granularity complexity analysis of program sections.

Program analysis should be performed at compile-time to yield better resource utilization and higher system throughput and at run-time to reveal concurrent arithmetic/logic activities. In a multiprocessor system, the percentage of code that can be vectorized ranges from ten to ninety percent across a broad range of scientific applications. The nonvectorizable code (scalar operations) tends to become the bottleneck. Using a few very powerful processors, we emphasize crude granularity. Therefore, the partitioning should be conducted at the highest possible level. This must be linked to the resource availability, time and space tradeoffs, and the overhead problem associated with fine granularity at the lower levels. Some language features and procedure calls must be incorporated to support the automatic partitioning of programs.

Analysis of various program graph characteristics is needed to determine dependence properties, critical paths, maximum resource demanded, and tradeoffs between computation, memory and I/O activities for algorithms with fine and crude granularities in node primitives. Nested loop analysis and multitasking conditions need to be revealed for various program graph types. Vectorization will support expression evaluation. Crude-granularity partitioning must be performed to have low overhead.

Multipipeline and multiprocessor scheduling policies have been suggested for a control-flow supercomputer [47], [48]. Run-time methods are needed for dynamically scheduling partitioned computing modules to multiprocessor hardware resources. These will include special hardware and software extensions for detecting the executability of computing modules. These extended functional mechanisms must meet the performance requirement and achieve better fault tolerance.

We are considering several priority heuristic schemes [17] such as *most successor first* and *least processing time first,* etc., for scheduling multiple tasks to multiple processors. The choice of a particular scheme depends on workload distributions and application demands. We have found some heuristic algorithms, even though the underlying problem is probably of exponential complexity. Analytical and experimental results are needed to access the speed and quality of various scheduling algorithms including data flow and heuristic optimization.

Synchronization is needed to execute parallel tasks, because different tasks may require different amounts of time to finish. Either control level or data-level synchronization methods can be used. In the control level, we use directed synchronization graphs. In the data-level, synchronization is implemented through shared variables. The shared semaphore registers will facilitate interprocessor communications. The memory latency problem must be overcome in order to support multi-PE systolization. The interconnection networks must be designed to have short delays. We prefer to use the pipelined multistage networks. Program and data caches must be used in processors. In this case, the local register file forms the data caches. Prefetch scheme is used to shorten the effective instruction fetch delays.

6. Throughput Analysis of the Remps

The throughput analysis of any multiprocessor system is often problem-dependent. In this sense, any conclusion on general machine performance must be biased and unrealistic. The performance of Remps is assessed with benchmark algorithms. We choose to provide two benchmark evaluations of the Remps. The first benchmark shows how to use the multiple processors in Remps to form a large macropipeline for solving linear recurrence systems with vector unknowns. The second example shows multitasking among multiple processors for finding the inverse of a large, triangular matrix. In both examples, multi-PE systolization within each processor is practiced. To evaluate the performance of these dynamically formed systolic array processors, various network delays and overheads are quantified to show their effects on the overall performance.

Let α and γ be the *setup times* of the routing network in each processor and of the interprocessor network, respectively. Let β and θ be the ***delays*** of these two networks, respectively. For general applications, we assume crossbar networks in both cases. Thus the α and γ are small constants, say each equal to tens of the clock periods of the PE pipeline. The two network delays are $\beta = \log_2 m$ and $\theta = \log_2 n$, respectively. With the projected growth in multiprocessor and network sizes, we can assume $0(10) \leq n \leq 0(10^2)$ and $0(10^2) \leq m \leq 0(10^3)$. Thus, we can estimate $\theta \leq \beta \leq 0(10)$ clock periods. The number of pipeline stages, k, in each PE is usually also a small number, around 0(10). The above discussions lead to the assumption that all the five parameters, α, β, γ, θ, and k are small constants with an order of tens of the PE clock periods.

In the Remps with n processors and m PEs per processor, the ***maximum speedup*** is equal to $m \cdot n$, as compared with a uniprocessor system with a single PE ($n = 1$ and $m = 1$). We shall denote the total compute times for a given algorithm executed on the Remps as $T_{m,n}$ and that on an uniprocessor as $T_{1,1}$. Then the ***speedup*** is defined as $S_{m,n} = T_{1,1}/T_{m,n}$. Again, the ***problem size, N,*** indicates the number of ***operand blocks*** (or ***wavefronts***) to be processed in the multiple systolic array processors. Whenever two-dimensional array is involved, we use the shorthand notation $\sqrt{m} = r$ or $r^2 = m$.

Example 2 (Macropipelining for solving linear recurrence systems)

As often performed in solving a linear, block-tridiagonal system [49], a family of linear recurrence systems needs to be solved for a sequence of vector unknowns: $X_i = (x_{i1}, x_{i2}, ..., x_{im})^T$ for $i=1,2,...,N$, where each X_i is an m-dimensional column vector. The computations involved are matrix-vector multiplications defined by:

$$X_i = A_{i1} \cdot X_{i-1} + A_{i2} \cdot X_{i-1} + \cdots + A_{i,2n-1} X_{i-2n+1} \quad \text{for } i=1,2,...,N, \tag{3}$$

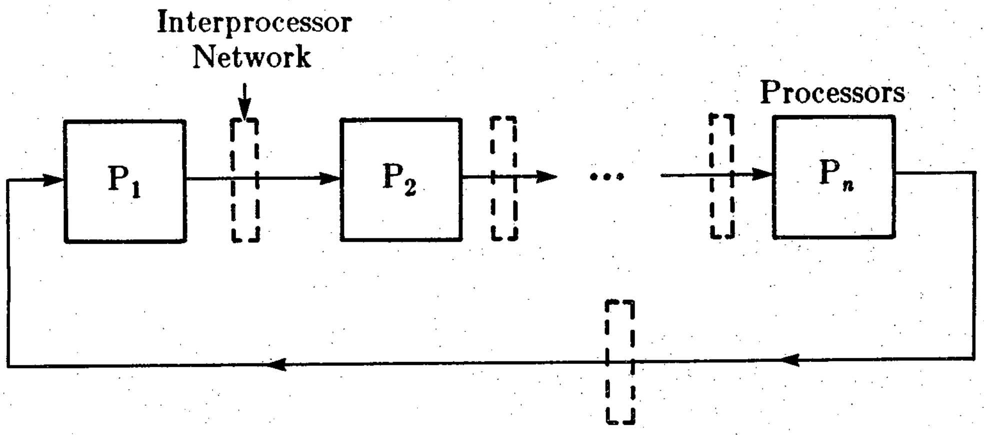
where each A_{ij} is an $m \times m$ matrix. The initial values of the system, $X_0, X_{-1}, ..., X_{-(2n-1)}$, should be given as ***inputs***. The ***outputs*** of the systems are X_i for $i=1,2,...,N$.

The n processors in Remps can be interconnected to form a ***macropipeline ring*** (Fig. 9a) for solving a recurrence system of order $2n-1$. Note that n processors are needed in the systolic ring for solving an order-$(2n-1)$ recurrence system as shown in [31]. The m PEs in each processor are configured into a ***linear systolic array*** (Fig. 9b) to carry out the component matrix-vector multiplications as characterized below: Denote $X_i^{(k)} = \sum_{j=1}^{k} A_{i,2n-j} \cdot X_{i-(2n-j)}$ for $k=1,2,...,2n-1$ and $X_i^{(2n-1)} = X_i$. Each linear systolic array processor performs:

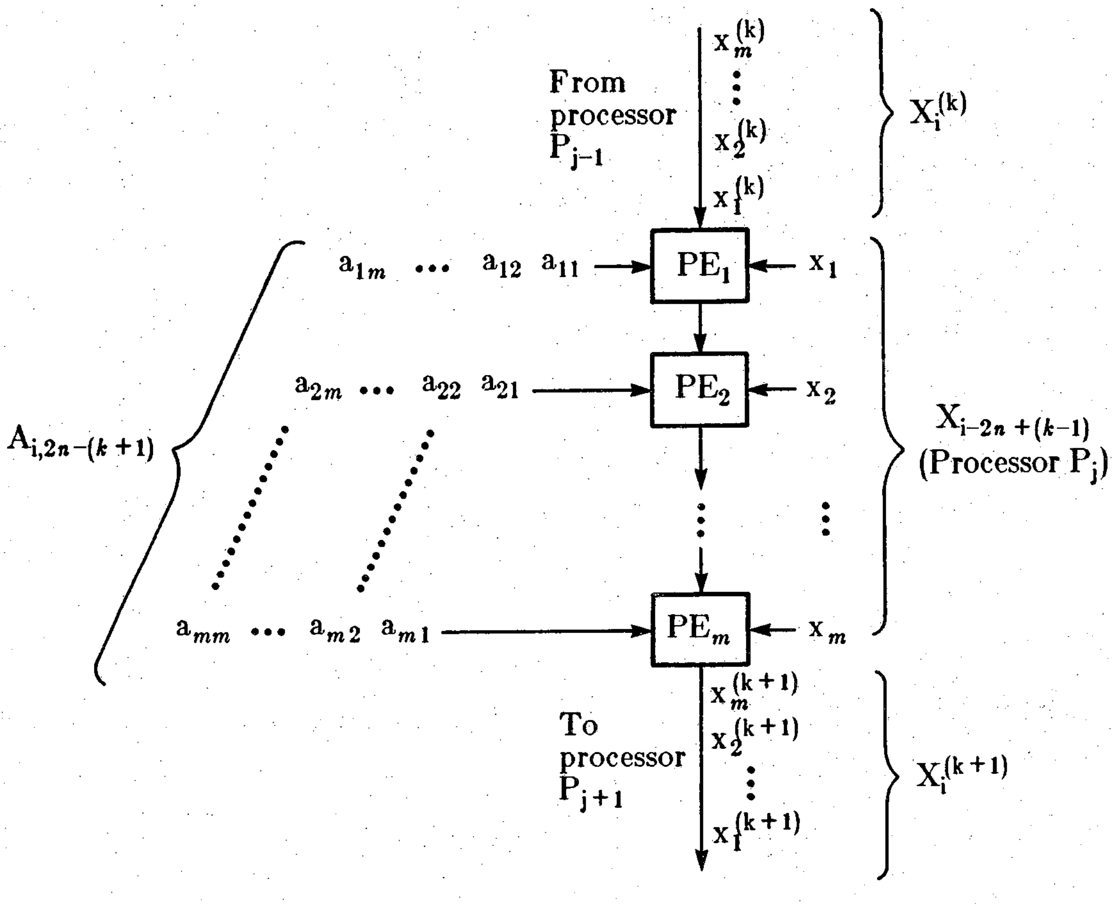
$$X_i^{(k+1)} = A_{i,2n-(k+1)} \cdot X_{i-(2n-k-1)} + X_i^{(k)} \quad \text{for } 1 \le k \le 2n-1 \tag{4}$$

The $X_i^{(k)} = (x_1^{(k)}, x_2^{(k)}, ..., x_m^{(k)})^T$ are from the outputs of the preceding processor and $X_i^{(k+1)}$ are the outputs to the next processor in the ring. The constant matrix $A_{i,2n-j}$ must be prestored in the local memory of the working processor.

The interprocessor network is used to establish the interfaces between adjacent processors in the ring. The n processors can communicate with each other in a dataflow fashion, that is, a processor initiates a task whenever all required data inputs become available. The global controller generates semaphores to be used for interprocessor communications. The interprocessor activities are mainly for block data transfers of the intermediate vectors, $X_i^{(k)}$, between adjacent processors. These intermediate vectors are passed directly between .processors without accessing the shared memory.



(a) Macropipeline ring of n processors



(b) The linear systolic array in each processor for computing Eq. 4

Figure 9. Multiprocessor macropipelining for solving linear recurrence system *(Example 2)*.

In each processor, it takes $mk+(m-1)\beta+(m-1) = (k+\beta+1)m-(\beta+1)$ clock periods to compute Eq. (4). The interprocessor network requires $\theta+m-1$ cycles to transfer the intermediate vector $X_i^{(k)}$. It takes t_1 time to compute the first output vector X_1, where $t_1 = \gamma+\alpha+(2n-1)[(k+\beta+1)m-(\beta+1)]+(2n-2)\theta$. The remaining vectors X_i for i=2,3,...,N, each can be produced in every $2m$ cycles. Thus, the total time needed to solve the recurrence system is:

$$T_{m,n} = t_1 + 2m(N-1) = 2mN + C_1mn - C_2m - C_3n + C_4 \tag{5}$$

where $C_1=2(k+\beta+1)$, $C_2=\beta+3$, $C_3=\beta-\theta+1$, and $C_4=\alpha+\beta+\gamma-2\theta-1$ are all small constants, compared with the much larger values of $N>m>n$. On the other hand, the same recurrence system requires $T_{1,1}$ time on a unprocessor, where

$$T_{1,1} = (2n-1)m^2N + k \tag{6}$$

where the PE in the uniprocessor is a pipelined scalar unit with k stages. However, the inner product operation $a \leftarrow a+b\times c$, takes 1 cycle to complete in this PE, once the pipeline is full. The speedup is thus obtained by dividing (6) by (5):

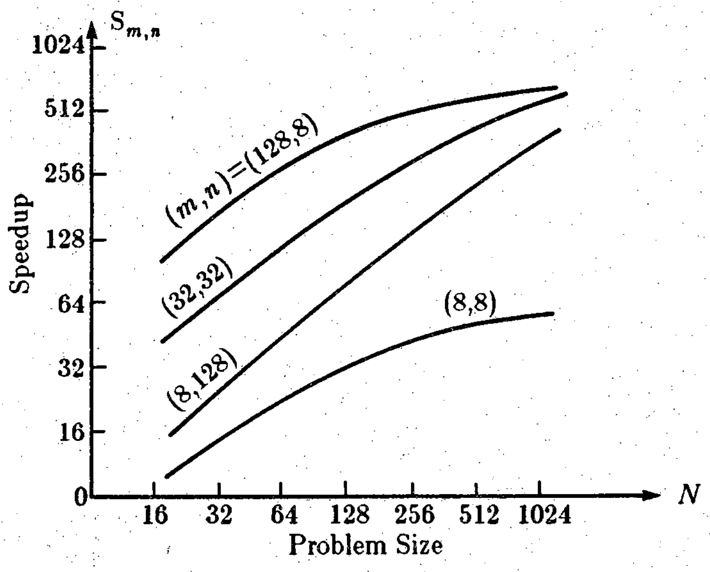
$$S_{m,n} = \frac{(2n-1)m^2N+k)}{2mN+C_1mn-C_2m-C_3n+C_4}$$

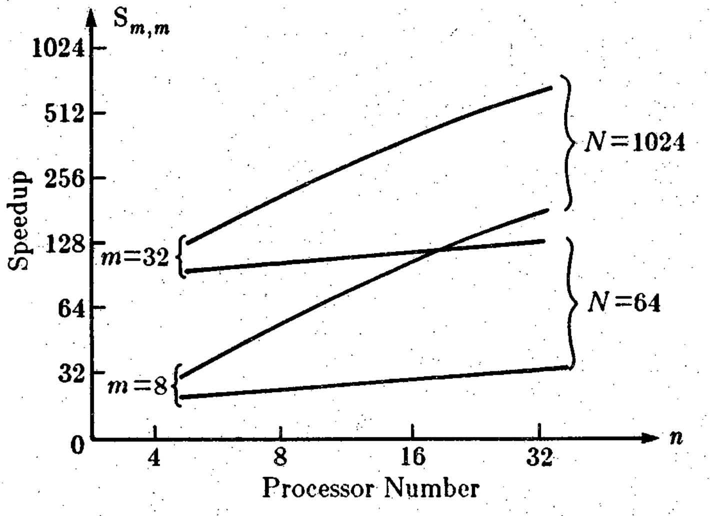
$$= mn, \qquad \text{as } N\rightarrow\infty \tag{7}$$

In Fig. 10, we plotted the speedup $S_{m,n}$ as a function of the problem size N, under various Remps sizes (m,n). This example demonstrates a worst-case study, because the critical path in the linear systolic array has the longest possible length, m. For ***systolic trees*** and ***hexagonal*** (or ***square) systolic arrays***, the critical paths would be respectively $\log_2 m$ and $\sqrt{m}$; much shorter than this worst-case of m. Therefore, the performance of the Remps would be even better for those systolic arrays with higher dimensions.

Example 3 (Multitasking for triangular matrix inversion)

The inversion of a nonsingular, triangular matrix can be done in a block-partitioning fashion as suggested in [50]. Given an upper, triangular matrix **U** of order rN. Suppose **U** is proven nonsingular. The purpose is to find the inverse matrix $\mathbf{U}^{-1} = \mathbf{V}$. The matrix **U** is being partitioned into $N(N+1)/2$ submatrices, each of which has a dimension $r\times r$. The output matrix **V** can be partitioned accordingly as shown below:





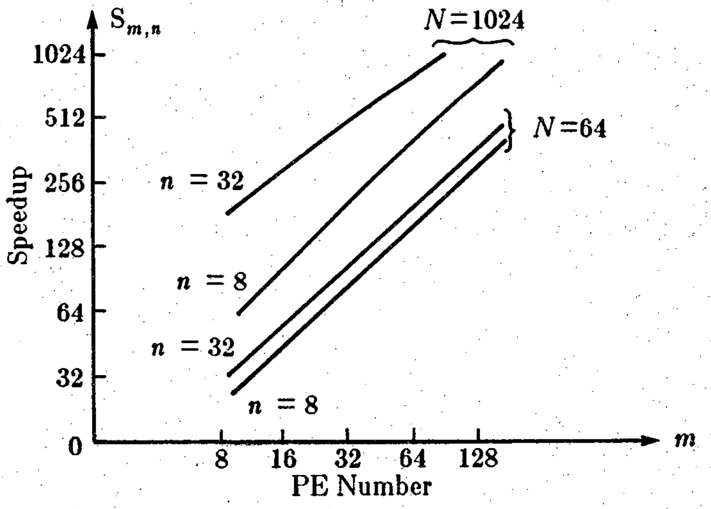


Figure 10. The speedup performance for *Example 2.*

$$\mathbf{U}^{-1} = \begin{bmatrix} U_{11} & U_{12} & \cdots & U_{1N} \\ & U_{22} & \cdots & U_{2N} \\ \bigcirc & & \ddots & \vdots \\ & & & U_{NN} \end{bmatrix}^{-1} = \begin{bmatrix} V_{11} & V_{12} & \cdots & V_{1N} \\ & V_{22} & \cdots & V_{2N} \\ \bigcirc & & \ddots & \vdots \\ & & & V_{NN} \end{bmatrix} = \mathbf{V} \tag{8}$$

where every U_{ij} and V_{ij} are $r \times r$ matrices for $1 \leq i \leq j \leq N$. The partitioned algorithm for systematically generating the output submatrices, V_{ij}, from input submatrices, U_{ij}, is specified below:

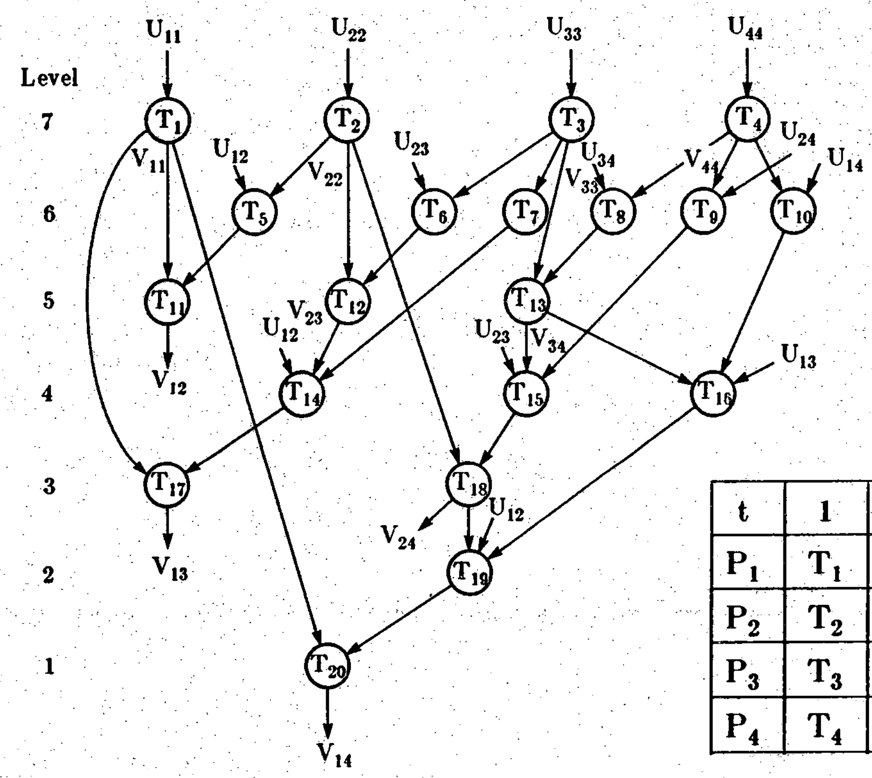
```
For i←1 to N step 1 do
        $V_{ii} = U_{ii}^{-1}$
Repeat
For i←1 to (N−1) step 1 do
        For j←1 to (N−1) step 1 do
                $W_{j,j+i} = \sum_{r=1}^{i} U_{j,j+r} \cdot V_{j+r,j+1}$                (10)
                $V_{j,j+i} = -V_{jj} \cdot W_{j,j+i}$
        Repeat
Repeat
```

The case of $N=4$ is shown in Fig. 11. The computing granularity of each node is specified in the submatrix level. All the submatrix computation tasks are listed in part (a). A precedence graph, showing the data dependence relations among these tasks, is given in part (b). From this graph, all the tasks belonging to the same level can be executed in parallel. There are $i(i+1)/2$ independent tasks parallelly executable at level $2i-1$ and i parallel tasks at level $2i$ for $1 \leq i \leq N-1$. In general, there are $2(N-1)+1=2N-1$ levels in the graph. A sample nonpreemptive schedule is shown in Fig. 11c for a Remps with $n=4$ processors.

The submatrix inversions (Eq. 9) are performed only with the diagonal submatrices. A triangular systolic array of $r(r+1)/2$ PEs has been suggested [50], [53] for computing the inverse of a triangular matrix of order r, as depicted in Fig. 12a. The elements of the input matrix U_{ii} are initially distributed in the PEs. The diagonal PEs perform scalar division operations. The remaining PEs perform scalar inner-product operations. It is assumed that each division or each inner-product operation takes one *time step,* which equals k clock periods in the PE pipeline. Such a triangular systolic array requires $2r-1$ time steps to complete the inversion process. Detailed PE operations in these steps can be found in [53].

Tasks	Submatrix Computations
$T_1 \sim T_4$	$V_{ii} = U_{II}^{-1}$ for i=1,2,3,4
$T_5 \sim T_{10}$	$W_{12} = U_{12} \cdot V_{22}$, $W_{23} = U_{23} \cdot V_{33}$, $W_{13} = U_{13} \cdot V_{33}$, $W_{34} = U_{34} \cdot V_{44}$, $W_{24} = U_{24} \cdot V_{44}$, $W_{44} = U_{14} \cdot V_{44}$.
$T_{11} \sim T_{13}$	$V_{12} = -V_{11} \cdot W_{12}$, $V_{23} = -V_{22} \cdot W_{23}$, $V_{34} = -V_{33} \cdot W_{34}$
$T_{14} \sim T_{16}$	$W_{13} = W_{13} + U_{12} \cdot V_{23}$, $W_{24} = W_{24} + U_{23} V_{34}$, $W_{14} = W_{14} + U_{13} \cdot V_{34}$
$T_{17} \sim T_{18}$	$V_{13} = -V_{11} \cdot W_{13}$, $V_{24} = -V_{22} \cdot W_{24}$
T_{19}	$W_{14} = W_{14} + U_{12} \cdot V_{24}$
T_{20}	$V_{14} = -V_{11} \cdot W_{14}$

(a) The partitioned computation tasks

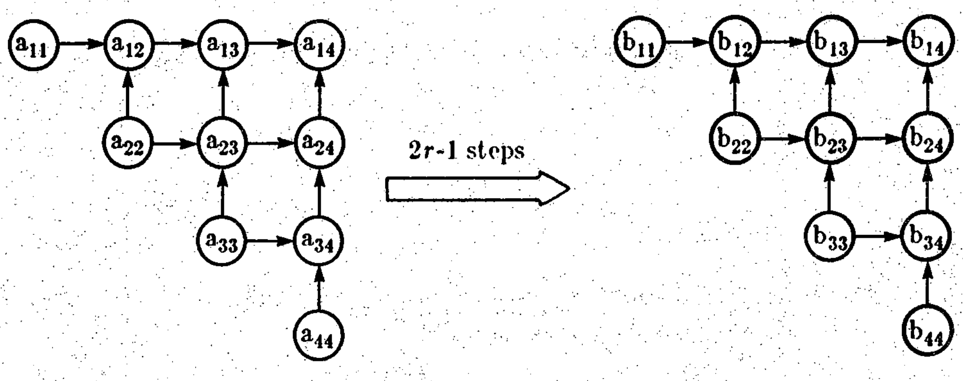


(b) Precedence graph

t	1	2	3	4	5	6	7
P_1	T_1	T_5	T_9	T_{11}	T_{17}		
P_2	T_2	T_6	T_{10}	T_{14}	T_{18}	T_{19}	T_{20}
P_3	T_3	T_7	T_{12}	T_{15}			
P_4	T_4	T_8	T_{13}	T_{16}			

(c) A sample schedule among 4 processors

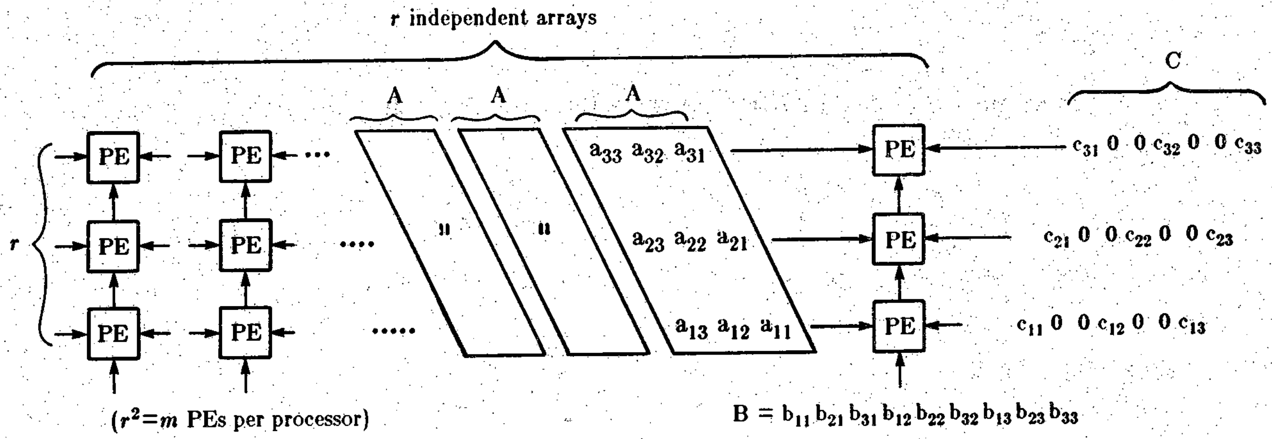
Figure 11. Partitioned matrix inversion for *Example 3* with $N = 4$



a_{11} a_{12} a_{13} a_{14}
a_{22} a_{23} a_{24}
a_{33} a_{34}
a_{44}
$2r$-1 steps
b_{11} b_{12} b_{13} b_{14}
b_{22} b_{23} b_{24}
b_{33} b_{34}
b_{44}

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ 0 & a_{22} & a_{23} & a_{24} \\ 0 & 0 & a_{33} & a_{34} \\ 0 & 0 & 0 & a_{44} \end{bmatrix}^{-1} = \begin{bmatrix} b_{11} & b_{12} & b_{13} & b_{14} \\ 0 & b_{22} & b_{23} & b_{24} \\ 0 & 0 & b_{33} & b_{34} \\ 0 & 0 & 0 & b_{44} \end{bmatrix}$$

(a) A triangular systolic array for matrix inversion (Preparata and Vuillemin 1980 [53])



r independent arrays
C
A
A
A
PE
r
a_{33} a_{32} a_{31}
a_{23} a_{22} a_{21}
a_{13} a_{12} a_{11}
c_{31} 0 0 c_{32} 0 0 c_{33}
c_{21} 0 0 c_{22} 0 0 c_{23}
c_{11} 0 0 c_{12} 0 0 c_{13}
($r^2 = m$ PEs per processor)
B = b_{11} b_{21} b_{31} b_{12} b_{22} b_{32} b_{13} b_{23} b_{33}

(b) Multiple linear systolic arrays for matrix multiplication ($C \rightarrow A \times B + C$) (Kulkarni and Yen 1982 [54])

Figure 12. Dynamically reconfigured systolic arrays used in *Example 3*.

Using a dynamic systolic implementation, the total time, including network overhead, required to invert a triangular submatrix, U_{ii}, equals $\alpha + t_2$, where $t_2 = (r-1)\beta + (2r-1)k = (\beta+2k)r - (\beta+k) \doteq (\beta+2k)r$ clock periods for large r. Note that the setup time α should be counted only once, if the array is to be used repeatedly for similar tasks. The N matrix inversions (Tasks $T_1 \sim T_4$ in Fig. 12a) can be performed by n processors in $\alpha + h \cdot t_2$ cycles, where $h = N/n$ is assumed an integer and only one triangular systolic array is implementable in each processor.

The remaining submatrix computations are for accumulative matrix-matrix multiplications (Eq. 10). Even there are square or hexagonal static arrays suggested for matrix multiplication in linear time [22], [29], [51], we choose to implement r independent, linear systolic arrays in each processor (Fig. 12b), to carry out r matrix-matrix multiplications in parallel. This decision is made due to the detrimental delay effects of using larger networks. There are at most nr linear systolic arrays in the entire system. Each linear systolic array multiplies two matrices, and, if needed, simultaneously adds the product $A \times B$ to a third input matrix C for accumulative operations. The detailed matrix-matrix multiplication steps in a linear array are described in [54]. The input matrix elements are flowing through the array, not the partial products. Every output element of the resulting matrix is computed (dot product operations) exclusively within each PE.

Once a linear array is set up, it requires $t_3 = (k+\beta)(r-1) + k + (r^2-1) = r^2 + (k+\beta)r - (\beta+1)$ cycles to multiply a pair of $r \times r$ matrices. For large value of r, we can approximate $t_3 \doteq r^2 + (k+\beta)r$. Assuming the desired operand data are continuously supplied to the systolic processors without a memory latency problem, we can estimate the total matrix multiplication time as follows:

The execution of all tasks at the same level on the precedence graph (Fig. 11b) must be completed before tasks at the successor level can be initiated. When N is very large, the number of matrix multiplications at each level may exceed the number, rn, of available systolic arrays. Consider $i = 1, 2, \ldots, N-1$. At the odd level $2i-1$, $p_{2i-1} = \left\lceil \frac{i(i+1)/2}{rn} \right\rceil$ iterations are needed to carry out the $i(i+1)/2$ pairs of multiplication. At the even level $2i$, $q_{2i} = \lceil i/rn \rceil$ iterations are needed to perform i matrix multiplications. In total, $M = \sum_{i+1}^{N-1} [p_{2i-1} + q_{2i}] \doteq (N^3/6 + 5N^2/4)/rn$ iterations are needed. Each iteration requires t_3 cycles to complete. Therefore, the total matrix multiplication time equals $\alpha + t_3 \cdot M$. The total time required to invert the entire matrix U is thus equal to:

$$T_{m,n} = (\alpha + h \cdot t_2) + (\alpha + t_3 \cdot M)$$

$$\doteq (N^3/6+5N^2/4)(r+k+\beta)/n + N(\beta+2k)r/n + 2\alpha$$

$$\doteq \frac{rN^3}{6n}, \quad \text{as the matrix size } rN\rightarrow\infty \tag{11}$$

On the other hand, a uniprocessor PE can compute $\mathbf{U}^{-1}$ in:

$$T_{1,1} = (k+rN-1)+[k+\frac{rN}{6}(r^2N^2+3rN-4)-1]$$

$$= \frac{r^3N^3}{6} + \frac{r^2N^2}{2} + \frac{rN}{3} + 2k-2 \tag{12}$$

The speedup performance is obtained by dividing (12) by (11):

$$S_{m,n} = \frac{T_{1,1}}{T_{m,n}} = \frac{r^3N^3/6}{rN^3/6n} = r^2n = mn, \quad \text{as } rN\rightarrow\infty. \tag{13}$$

The above analysis ignores the memory latency problem associated with fetching or storing the large number of matrix elements through the memory hierarchy (register files, local memory, and global memory). The memory latency is caused by memory access delays and various network delays. If one adds these delays, then we should modify the processing times t_2 and t_3 to $t_2' = t_2+Cr^2$ and $t_3' = t_3+Cr^3$, respectively, where C is the *average memory latency* per matrix element. Depending on the hierarchical memory structure and technology involved, the value of the latency, C, may range from a few tenths to tens of the processor clock period. The higher is the degree of memory interleaving and the faster is the memory technology, the smaller will be the value of C. The added terms Cr^2 and Cr^3 are the memory latencies associated with transferring r^2 and r^3 matrix elements in submatrix inversion and in r submatrix multiplications, respectively, in each processor. Substituting t_2 and t_3 by t_2' and t_3' respectively in Eq. 11, we obtain a degraded processing time:

$$T_{m,n}' = 2\alpha+ht' + Mt_3' = \frac{Cr^2N^3}{6n}, \quad \text{as } rN\rightarrow\infty \tag{14}$$

Using Eq. 12 in Eq. 14, we obtain a reduced speedup with $r=\sqrt{m}$: Note that the memory latency does not pose a problem for the scalar uniprocessors. Thus, the time $T_{1,1}$ does not change.

$$S_{m,n}' = \frac{T_{1,1}}{T_{m,n}'} = \frac{mn}{Cr} = \frac{n\sqrt{m}}{C}, \quad \text{as } rN\rightarrow\infty \tag{15}$$

The memory latency problem causes a degradation in processor performance, only when the processors are required to wait for the complete arrival of all the required data blocks for each task to be executed. This processor idling situation can be alleviated if *context switching* is allowed at the processor level. Whenever a dynamic systolic processor becomes free but the desired data blocks are not resident in the processor, the processor is allowed to be switched (and the systolic array be reconfigured) to execute other ready-to-run tasks. Only after all the required operands have been loaded into the register file, the processor can be switched back to execute an old task, which was waiting to be initiated. However, once a task is initiated, it must be executed until completion.

Context switching reduces processor idle time (waiting for operands) and increases the processor utilization. However, additional ***switching overhead***, Ω, is introduced every time the processor switches to another executable task. This overhead is attributed to the time needed to perform context switching of all register contents and processor states [27]. The value of the overhead, Ω, may vary from tens to hundreds of the processor clock periods, depending on the degree of multiprocessing being supported by the operating system. With context switching, the basic processing times t_2 and t_3 should be changed to $t_2'' = t_2+\Omega$ and $t_3'' = t_3+\Omega$, respectively. The total compute time in Eq. 11 should be modified as follows:

$$T''_{m,n} = (\alpha+t_2'')h + (\alpha+t_3'')\cdot M$$

$$= \frac{rN^3}{6n}\cdot(1+\frac{\Omega+\alpha}{r^2}+\frac{\beta+k}{r}), \quad \text{as } rN\to\infty \tag{16}$$

The speedup in Eq. 13 then becomes

$$S''_{m,n} = \frac{T_{1,1}}{T''_{m,n}} = \frac{mn}{1+\frac{\Omega+\alpha}{m}+\frac{\beta+k}{\sqrt{m}}}, \quad \text{as } rN\to\infty \tag{17}$$

The three speedups $S_{m,n}$, $S'_{m,n}$, and $S''_{m,n}$ are plotted in Fig. 13 under the assumption $\alpha=k=10$, $C=2$, $\Omega=20$, and $\beta=\log_2 3m$. When the memory latency does not pose a serious problem, the speedup function, $S_{m,n}$, increases rapidly to the maximum value of mn. The curve $S'_{m,n}$ shows a serious performance degradation caused by memory latency. The middle curve $S''_{m,n}$ shows that context switching indeed helps improve the performance significantly, even the memory latency is long when processors must go to the shared global memory for exchange of large data blocks. If $\Omega,\alpha=0(m)$ and $\beta,k=0(\sqrt{m})$, then the $S''_{m,n}$ declines only slightly, as the problem size N becomes large.

In Fig. 14, we show the detrimental effects on the speedup functions, $S'_{m,n}$ and $S''_{m,n}$, by increasing the values of the memory latency C and the context switching overhead Ω, respectively. We conclude from these plots that the memory latency should be minimized in any multiprocessor system. If an appreciable memory latency does exist, then context switching is highly recommended to alleviate the problem. To support multitasking among multiple processors with shared memory, context switching is necessary to minimize the processor idle time and thus increase the system throughput.

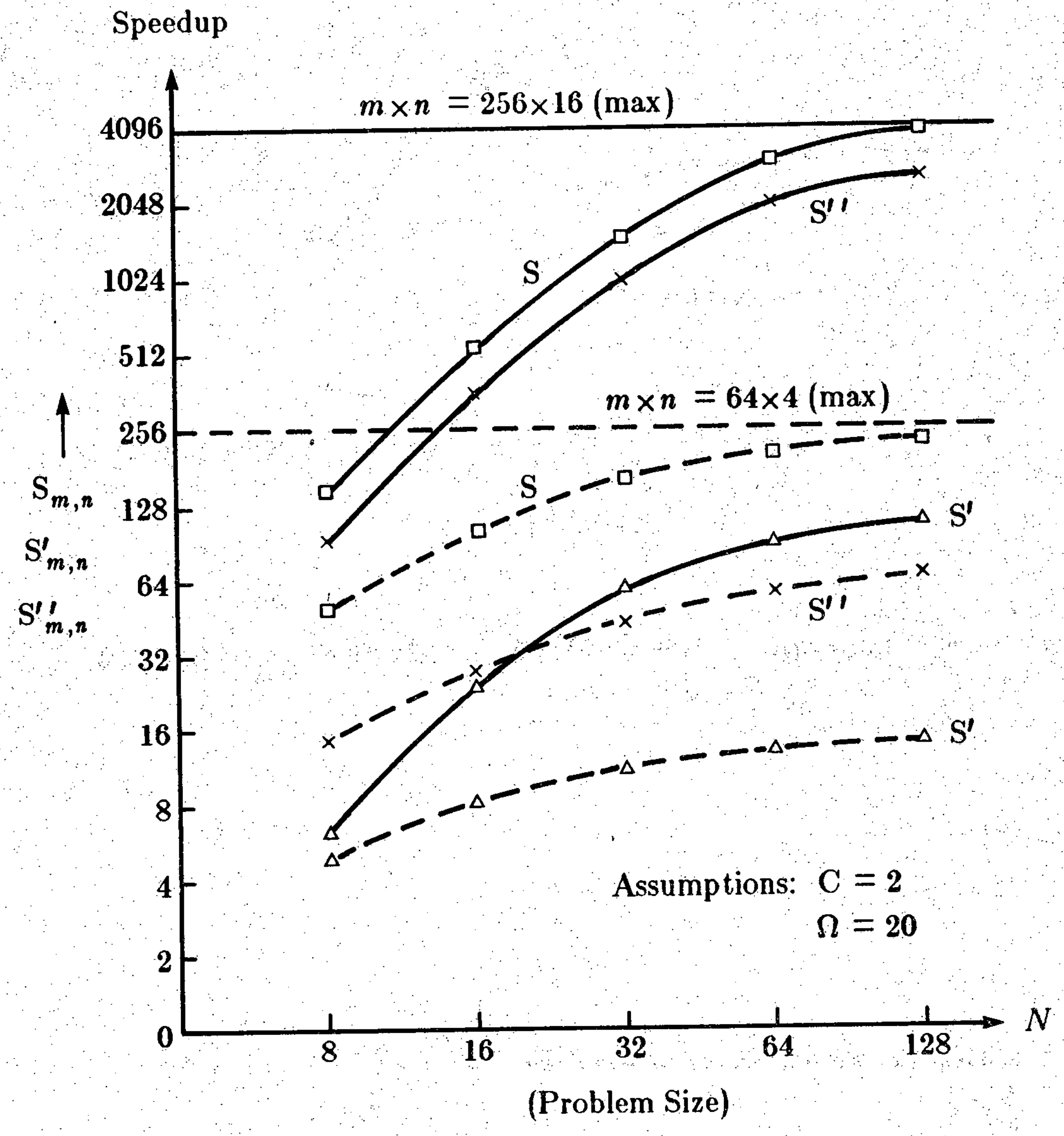
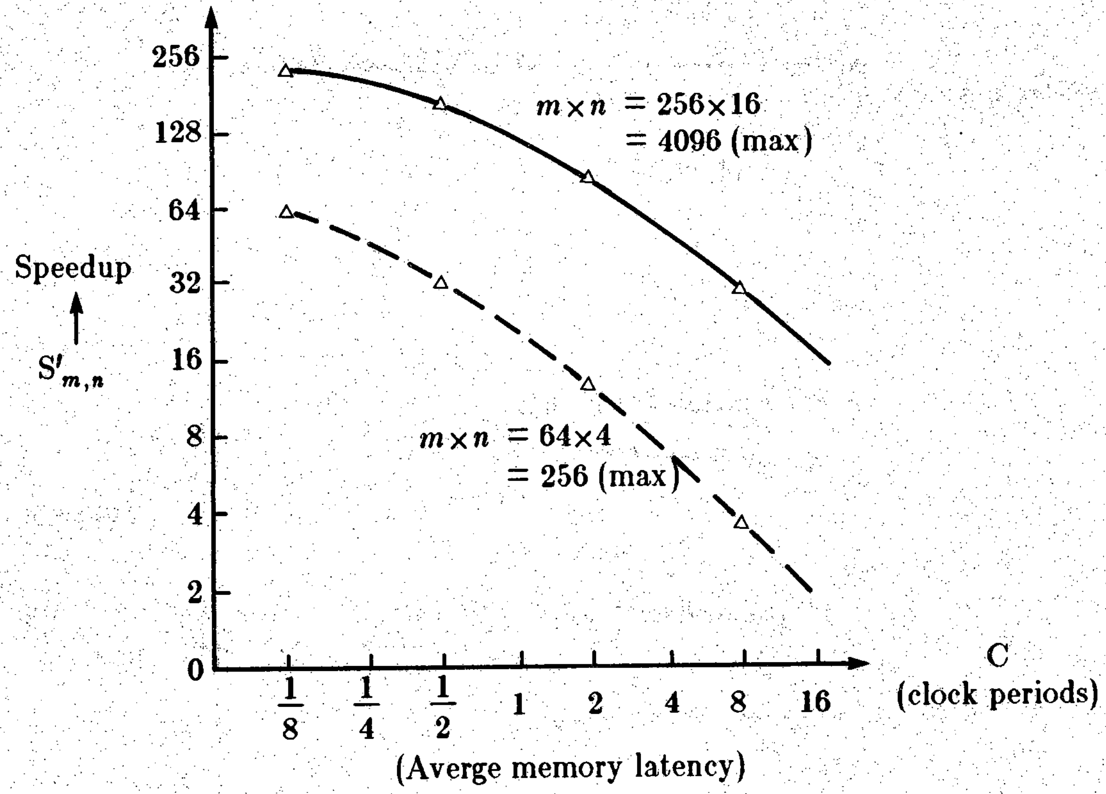


Figure 13. Speedup performance for *Example 3* with solid curves for (m,n) = (256,16) and dashed curves for $(m,n) = (64,4)$.



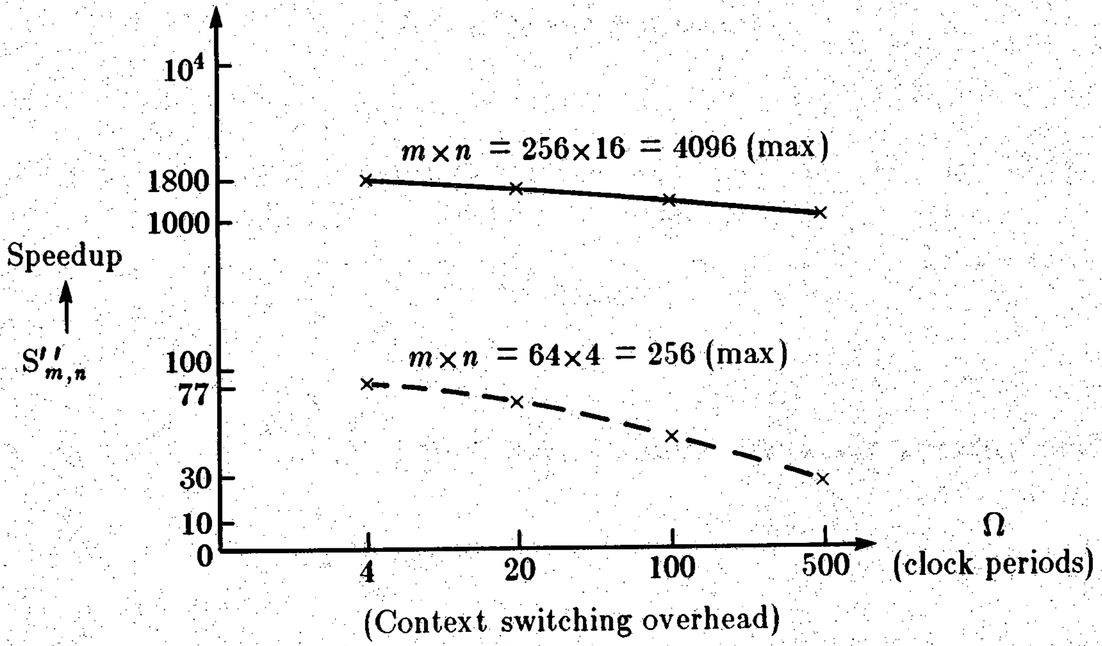


Figure 14. Performance degradation of the Remps of two sizes due to memory latency and context switching overhead.

However, the switching overhead should be maintained as low as possible. The above two examples demonstrate that the speed improvement can be achieved by the same set of hardware PEs to perform different algorithmic computations at different times. For large problem size, N, or large resource pool, (m,n), the speedups are rather impressive, plus the obvious gains in application flexibility.

7. Conclusions

The interprocessor network is used only to directly pass small and regular data sets between processors (as shown in Examples 1 and 2). The shared memory must be used, when large data sets are transferred among processors (as shown in Example 3). For reasonably large problems, the dynamic systolic arrays can perform as well as their static counterparts. The increased network overhead does not pose a serious problem when the problem size is sufficiently large and well partitioned for multitasking. When the problem size is small, frequent reconfiguration of the systolic processor may not be advantageous. Building dynamic, systolic/wavefront array processors demands extensive R/D efforts, before achieving cost-effectiveness in real machines.

The proposed reconfigurable multiprocessor system is designed at this stage primarily for supercomputing in scientific/engineering applications. Special design methodologies and operational features of the Remps are presented. The proposed MIMD system has a hierarchical structure which provides macro dataflowing at the interprocessor level and control flowing at the intraprocessor level. Reconfigurability of the system emphasizes application flexibility and fault tolerance [56].

Our continued efforts are being directed to performance analysis of Remps for large-scale, scientific/vector computations; such as for solving partial differential equations (PDE) problems [57], [58]. We are also extending the PE designs for implementing parallel-production systems in a multiprocessor environment [59], [60]. This explores the prospects of designing a supercomputer which is useful for both numerical analysis and AI-oriented applications [61], [62]. Our initial findings show that the dynamic systolic arrays may be more suitable for numeric processing, whereas the dynamic wavefront processors may be more attractive to symbolic manipulations.

References

1. Cray Research, Inc., "The Cray X-MP Series of Computer Systems," *Technical Brochure*, Minneapolis, Minnesota, Aug. 1984.

2. Smith, B. J., "Architecture and Application of the HEP Multiprocessor Computer System," *Real Time Signal Processing IV,* Vol. 298, August 1981.

3. Farmwald, P. M. "The S-1 Mark IIA Supercomputer," in *High-Speed Computations* (J. S. Kowalik ed.), Springer-Verlag, 1984.

4. Lipovski, G. J. and Tripathi, A., "A Reconfigurable Varistructured Array Processor," *Proc. 1977 Int. Conf. on Parallel Processing,* 1977, pp. 165-174.

5. Brown, J. C., "TRAC: An Environment for Parallel Computing," *COMPON,* pp. 294-298, Spring 1984.

6. Schwartz, J. T., "Ultracomputers," *ACM Trans. Programming Languages and Systems,* Vol. 2, No. 4, 1980, pp. 484-521.

7. Gottlieb, A., Grishman, R., Kruskal, C. P., McAaliffe, K. P., Randolph, L. and Snir, M., "The NYU Ultracomputer-Designing an MIMD Shared Memory Parallel Computer," *IEEE Trans. Computers,* February 1983, pp. 175-189.

8. Snyder, L., "Introduction to the Configurable Highly Parallel Computer," *IEEE Computer,* January 1982, pp. 47-64.

9. Kung, H. T., "Overview of Systolic Array Projects at CMU," *internal document,* Carnegie-Mellon University, September 1984.

10. Briggs, F. A., Fu, K. S., Hwang, K., and Wah, B. W., "Pumps Architecture for Pattern Analysis and Image Database Management," *IEEE Trans. Computers,* Vol. C-31, No. 10, October 1982, pp. 969-982.

11. Gajski, D., Kuck, D., Lawrie, D., and Sameh, A., "Construction of a Large Scale Multiprocessor," *Rept. No. UIUCDCS-R-83-1123,* Dept. of Computer Science, University of Illinois, Urbana, February 1983.

12. Dennis, J. B., "Data Flow Supercomputers," *IEEE Computer Magazine,* November 1980, pp. 48-56.

13. Arvind and Iannucci, R. A., "A Critique of Multiprocessing von Neumann Style," *Proc. of 10th Ann. Symp. Computer Architecture,* June 1983, pp. 426-436.

14. Carlson, W. W., and Hwang, K., "On Structural Data Accessing in Dataflow Computers," *Proc. 1st Int'l. Conf. Computers and Applications,* Beijing, China, June 1984.

15. Keller, R. M., Patil, S. S., and Lindstrom, G., "A Loosely Coupled Applicative Multiprocessing System," *Proc. Nat'l Computer Conf.,* AFIPS Press, 1979.

16. Watson, I. and Gurd, J., "A Practical Data Flow Computer," *IEEE Computer,* Vol. 15, No. 2, 1982, pp. 51-57.

17. Hwang, K. and Su, S. P., "Priority Scheduling in Event-Driven Dataflow Computers," *TR-EE 83-36,* School of E.E., Purdue University, 1983.

18. Gaudiot, J. L. and Ercegovac, M. D., "Performance Analysis of a Data-Flow Computer with Variable Resolution Actors," *4th Int'l. Conf. on Distributed Computing Systems,* San Francisco, California, May 1984.

19. Gajski, D. and Pier, J. K., "Parallel Processing: Problems and Solutions," *Tech. Report,* Univ. of Illinois, Urbana, Ill., 1984.

20. Kung, H. T., "Why Systolic Architectures?," *IEEE Computer,* January 1982, pp. 37-46.

21. Hsu, F. H., Kung, H. T., Nishizawa, T., and Sussman, A., "LINC: The Link and Interconnection Chip," CMU *internal document,* Dept. of Computer Science, Carnegie-Mellon Univ., May 1984.

22. Kung, S. Y., "On Supercomputing with Systolic/Wavefront Array Processors," *Proc. of IEEE,* Vol. 72, No. 7, July 1984.

23. Fisher, A. L., Kung, H. T., Monier, L. M., and Dohi, Y., "Architecture of the PSC: A Programmable Systolic Chip," *10th Ann. Int'l. Symp. on Computer Architecture,* June 1984, pp. 48-53.

24. Reddi, S. S., and Feustel, E. A., "A Restructurable Computer System," *IEEE Trans. Comp.,* Vol. C-27, Jan. 1978, pp. 1-20.

25. Vanaken, J. R., and Zick, G. L., "The Expression Processor: A Pipelined Multiple-Processor Architecture," *IEEE Trans. Comp.,* Vol. C-30, Aug. 1981, pp. 525-536.

26. Hwang, K. and Xu, Z. "Dynamic Systolization for Developing Multiprocessor Supercomputers," *TR-EE 84-42,* School of E. E., Purdue University, November 1984

27. Hwang, K. and Briggs, F. A., *Computer Architecture and Parallel Processing,* McGraw-Hill, New York, 1984.

28. Hwang, K. (Editor), *Supercomputers: Design and Applications,* IEEE Computer Society Press, August 1984.

29. Chin, C. Y. and Hwang, K., "Packet Switching Networks for Multiprocessors and Dataflow Computers," *IEEE Trans. on Computers,* special issue on Parallel Processing, November 1984.

30. Kapauan, A., Field, J. T., Gannon, D. B., and L. Snyder, "The Pringle Parallel Computer," *Proc. of the 11th Int'l Symp. on Computer Architecture,* June 1984, pp. 12-20.

31. Kung, H. T. and Lam, M., "Wafter-Scale Integration and Two-Level Pipelined Implementations of Systolic Arrays," *Journal of Parallel and Distributed Computing,* Vol. 1, No. 1, September 1984, pp. 32-63.

32. Feng, T. Y., "Data Manipulation Functions in Parallel Processors and Their Implementations," *IEEE Trans. Computers,* Vol. C-23, No. 3, March 1974, pp. 309-318.

33. Benes, V. E., *Mathematical Theory of Connection Networks and Telephone Traffic,* New York, Academic Press, Inc., 1965.

34. Wu, C.-L., and Feng, T.-Y. (Editors), *Interconnection Networks for parallel and Distributed Processing,* IEEE Computer Society Press, August 1984.

35. Moldovan, D. I., "On the Design of Algorithms for VLSI Systolic Arrays," *Proc. of IEEE,* Vol. 71, No. 1, 1983, pp. 113-120.

36. Quinton, P., "Automatic Synthesis of Systolic Arrays from Uniform Recurrent Equations," *11th Ann. Int'l. Symp. on Computer Architecture,* June 1984, pp. 208-214.

37. Franklin, M. A., "VLSI Performance Comparison of Banyan and Crossbar Communication Networks," *IEEE Trans. Computers,* April 1981, pp. 283-290.

38. Nassimi, D., and Sahni, S., "A Self-Routing Benes Network and Parallel Permutation Algorithms," *IEEE Trans. Computers,* May 1981, pp. 332-340.

39. Cantor, D. G., "On Non-Blocking Switching Networks," *Networks,* Winter 1971, pp. 367-377.

40. Bassalygo, L. A., and Pinsker, M. S., "On the Complexity of Optima Non-Blocking Switching Networks Without Rearrangement," in *Problems in Information Transmission,* Plenum Pub. Corp., New York, 1973, pp. 84-87.

41. Cantor, D. G., "On Construction of Nonblocking Switching Networks," *Proc. Symp. Computer-Communication Networks and Teletraffic,* Polytechnic Institute of Brooklyn, 1972.

42. Pippengen, N., "On Rearrangeable and Non-Blocking Switching Networks," *J. of Computer and System Science,* September 1978, pp. 145-162.

43. Kumar, M. and Jump, J. R., "Performance Enhancement in Buffered Delta Networks Using Crossbar Switches and Multiple Links," *Journal of Parallel and Distributed Computing,* Vol. 1, No. 1, August 1984, pp. 81-103.

44. Gajski, D. D., Panda, D. A., Kuck, D. J., and Kuhn, R. H., "A Second Opinion on Dataflow Machines and Languages," *IEEE Computer,* February 1982, pp. 58-70.

45. Kuck, D. J., Kuhn, R. H., Leasure, B., and Wolf, M. J., "The Structure of An Advanced Retargetable Vectorizer," in *Supercomputers: Design and Applications,* (Ed. Hwang) IEEE Computer Society Press, August 1984.

46. Larson, J., "Multitasking on the Cray X-MP/2 Multiprocessor," *IEEE Computer,* July 1984, pp. 62-69.

47. Su, S. P. and Hwang, K., "Multiple Pipeline Scheduling in Vector Supercomputers," *Proc. Int'l. Conf. on Parallel Processing,* August 1982, pp. 226-234.

48. Sahni, S., "Scheduling Multipipeline and Multiprocessor Computers," *IEEE Trans. Computers,* July 1984, pp. 637-645.

49. Sameh, A. H., "On Two Numerical Algorithms for Multiprocessors," *Proc. NATO Advanced Research Workshop on High Speed Computing,* Edited by Kowalik, W. Germany, June 1983.

50. Hwang, K. and Cheng, Y. H., "Partitioned Matrix Algorithms for VLSI Arithmetic Systems," *IEEE Trans. Computers,* Vol. C-31, No. 12, December 1982, pp. 1215-1224.

51. Kung, H. T., and Leiserson, C. E., "Systolic Arrays (for VLSI)," *Sparse Matrix Proc.,* (Duff, et al., eds.), *SIAM,* Philadelphia, Penn., 1978, pp. 245-282.

52. Fisher, A. L., Kung, H. T., Monier, L. M., and Dohi, Y., "Architecture of the PSC: A Programmable Systolic Chip," *Proc. of the 10th Annual Int'l Symp. on Computer Architecture,* June 1983, pp. 48-53.

53. Preparata, F. P. and Vuillemin, J., "Optimal Integrated-Circuit Implementation of Triangular Matrix Inversion," *Proc. Int'l. Conf. on Parallel Processing,* August 1980, pp. 211-216.

54. Kulkarni, A. V. and Yen, D. W., "Systolic Processing and an Implementation for Signal and Image Processing," *IEEE Trans. Computers,* Vol. C-31, No. 10, Oct. 1982, pp. 1000-1009.

55. Hwang, K., "VLSI Computer Arithmetic for Real-Time Image Processing," in *VLSI Electronics: Micro-structure Science,* Vol. 7, (Einspruch, Ed.), 1983, Academic Press, New York, pp. 303-331.

56. Hwang, K., "Multiprocessor Supercomputers for Scientific Applications," *IEEE Computer,* special issue on *Multiprocessing Technology and Systems,* July 1985 (to appear).

57. Rice, J. R., "Very Large Least Square Problems and Supercomputers," *Technical Report 464,* Dept. of Computer Science, Purdue University, December 1983.

58. Ni. L. M. and Hwang, K., "Vector Reduction Methods for Arithmetic Pipelines," *IEEE Trans. Computers,* accepted to appear early 1985.

59. McDonald, J. F., Rogers, E. H., Rose, K., and Steckl, A. J., "The Trials of Wafer Scale Integration," *IEEE Spectrum,* Oct. 1984, pp. 32-39.

60. Stolfo, S. J., Miranker, D. and Shaw, D. E. "Architecture and Applications of DADO: A Large-Scale Parallel Computer for Artificial Intelligence," *Proc. of Int'l Joint Conf. on Artificial Intelligence,* 1983.

61. Ullman, J. D., "Flux, Sorting and Supercomputer Organization for AI Applications," *Journal of Parallel and Distributed Computing,* Vol. 1., No. 2, November 1984.

62. Zadeh, L. A., "Making Computers Think Like People," *IEEE Spectrum,* August 1984, pp. 26-32.