Computational Models and Resource Allocation for Supercomputers

JON MAUNEY, DHARMA P. AGRAWAL, FELLOW, IEEE, YOUNG K. CHOE, EDWIN A. HARCOURT, SUKIL KIM, AND WAYNE J. STAATS

Supercomputers are capable of providing tremendous computational power, but must be carefully programmed to take advantage of that power. There are several different architectures used in supercomputers, with differing computational models. These different models present a variety of resource allocation problems that must be solved.

The computational needs of a program must be cast in terms of the computational model supported by the supercomputer, and this must be done in a way that makes effective use of the machine's resources. This is the resource allocation problem. The computational models of available supercomputers and the associated resource allocation techniques are surveyed. It is shown that many problems and solutions appear repeatedly in very different computing environments.

I. Introduction

Effective use of supercomputers naturally requires that generated code be structured to take advantage of the capabilities of the target supercomputer. The first step in programming any computer is choosing an appropriate algorithm, and this step is especially important for supercomputers. For many problems there exist several different approaches to the solution, with differing degrees and kinds of parallel or vector operations. Once an algorithm is chosen, it must be implemented in such a way as to make efficient use of the parallel or vector capabilities of the machine. Successful implementation requires a thorough understanding of the computational model(s) embodied in the design of the machine and of the best techniques for exploiting said model(s).

Discussions of supercomputer programming quickly come around to the issue of programming language. On the one hand, parallelizing compilers for older, sequential languages would allow us to avoid retraining armies of programmers and recoding stacks of programs. On the other hand, new, inherently parallel languages would free us from the shackles of old-fashioned, sequential thought patterns. Currently available parallel computers are often supplied with neither—programs are written in a sequential language augmented with constructs for explicitly creating and manipulating parallelism.

Since discussions of parallel and vector computing are immediately factionalized by programming language style, an important similarity is often obscured: in parallel program implementation, eventually it must be decided how best to match the parallel components of the program with the parallel capabilities of the computer. In a vectorizing or parallelizing compiler for, say, Fortran, the compiler must analyze the program to discover which parts of it can be efficiently executed in parallel or with vector instructions [1], [2]. A translator for a modern "data flow" or other "parallel" language will find that parallelism is readily apparent in the program, but must still determine the best way to use that parallelism on a given computer. A programmer using explicit parallel constructs must, of course, do all of this work by hand.

We assert that the problem of implementing programs for parallel or vector execution can be logically viewed in two phases: first the program is analyzed to discover operations that *could* be done in parallel, then these operations are compared to the characteristics of the target machine to determine which ones *should* be done in parallel (or with vector instructions) [3]. A given language may make the first phase difficult or trivial, but discussions of ways of exploiting parallelism, whether by hand or by compiler, should be decoupled from arguments over the best way to write parallel programs.

In this paper we will be concerned with these "second phase" issues of exploiting parallelism. First we will review the common styles of supercomputer architecture and their computational models. Then we will discuss strategies for allocating supercomputer resources, both statically and dynamically. Finally, we present a few case studies, showing concrete computational models and the allocation strategies employed.

Manuscript received July 27, 1988; revised March 24, 1989. This work was supported in part by the US Army Research Office contract DAAG-29-85-K-0236 and DAAL-03-89-K-D142, and NASA contract NAG-2-449.

J. Mauney and E. A. Harcourt are with the Department of Computer Science, North Carolina State University, Raleigh, NC 27695, USA.

D. P. Agrawal is with the Department of Electrical and Computer Engineering, North Carolina State University, Raleigh, NC 27695, USA.

Y. K. Choe is with Tangram Systems Corp., Cary, NC 27511-6446, USA.

S. Kim is with the Agency for Defense Development, Daejon, Korea.

W. J. Staats is with IBM, Cary, NC 27511, USA.

IEEE Log Number 8933307.

II. Computational Models

A supercomputer is essentially any machine that is near the top end of performance, regardless of how it is constructed [4]. In fact, however, all supercomputers achieve their high performance by using high-speed components and by performing multiple operations simultaneously. It is this overlapped execution that throws a curve into supercomputer programming. The same architectural features and programming problems can also be found in less expensive systems that don't provide supercomputer performance. Since we are concerned with computational models and ways of allocating resources to get the best performance from a given machine, we will not try to distinguish between "true" supercomputers and "mini-supers" or just plain parallel computers.

Parallel computers are commonly divided according to Flynn's classifications [5] into SIMD (Single Instruction Multiple Data) and MIMD (Multiple Instruction Multiple Data). This division is useful but not ideal; there are supercomputers that are difficult to classify, and important distinctions that are not made. Vector processors can be grouped with SIMD machines for purposes of discussion, although in fact they depend on another form of parallelism, pipelining, for their high performance. Newer machines such as the Alliant and the Cray X-MP provide multiple processors, each with vector capability, which we can call MSIMD. A diagram of Flynn's taxonomy and representative computers is shown in Fig. 1. The subdivisions in this taxonomy are explained in Section II-B.

Flynn's classification does not distinguish other characteristics that are important to the computational model, such as the likely granularity of parallel computation or the method of synchronization. A chart showing the grouping of supercomputers according to these characteristics is shown in Fig. 2.

A. Program Dependencies

The basis of parallel programming is the understanding of dependencies within a program. A dependency exists between two operations in a program whenever the logic of the program dictates that the operations must be performed in a particular order. The most important example is data dependence: one operation computes a value that is subsequently used by another operation. Clearly, the second operation must wait for the first to finish, in order to obtain the needed data. Other forms of dependence involve ensuring that updated values of a variable are written in the correct order, and determining the results of a conditional test, to see whether a particular piece of code will be executed. If there is no dependence between two operations, then they can be performed in any order, including simultaneously.

Analysis of such dependencies is a standard part of compiler technology [6]. This analysis has been extended to array references within a loop to determine the possible vector or parallel execution of a loop [1], and to determine possible MIMD parallel execution of operations in or out of a loop [7]–[9].

The synchronization imposed by data dependencies is the basis of the dataflow model of parallel computation [10]. Other forms of dependency are usually viewed as being artificial impediments to parallel execution. Functional [11] and single-assignment [12], [13] programming languages attempt to improve parallelism by preventing unnecessary dependencies. Non-dataflow dependencies can also be automatically removed by a compiler by "renaming" [14]. Renaming is an allocation technique: the needs of the source program—variables in this case—are mapped to hardware resources—storage locations—in a way that reduces the number of dependencies, thus improving the performance.

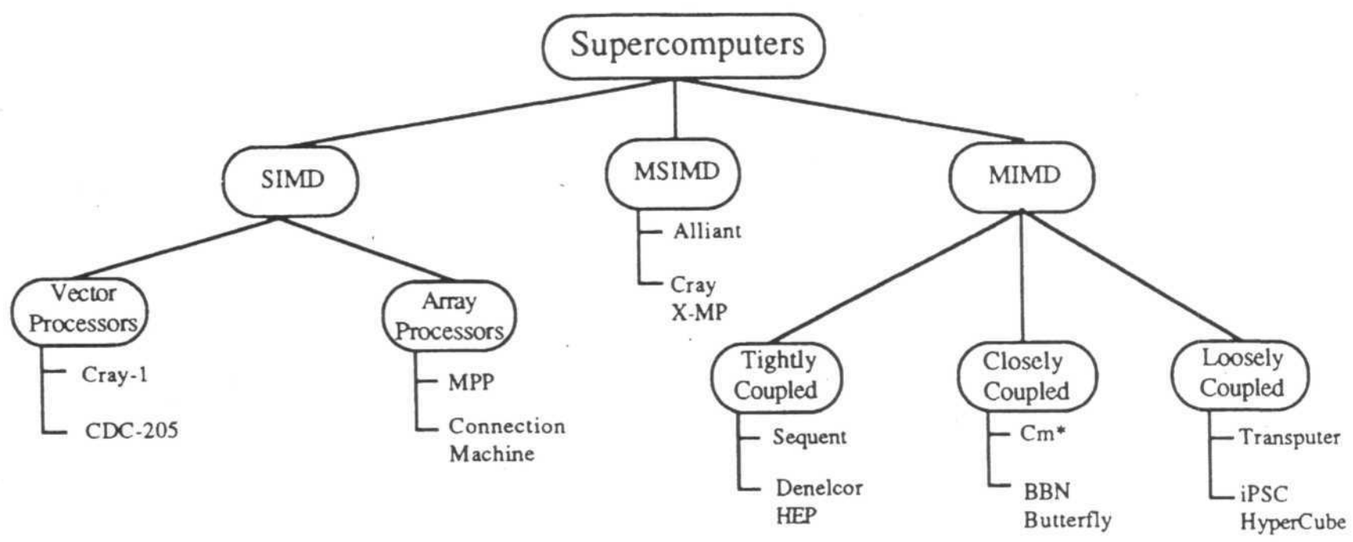


Fig. 1. Taxonomy of supercomputer architectures.

Grain size	Processor synchronization	Computational model	Flynn's Classification	Example machines
Fine granularity models	data flow	Data flow model	—	Sigma-1
	interleaving	Pipeline model	SIMD	Hitachi SX-200, Cray-1, CDC Cyber 205, Fujitsu VP-100/200, etc
	synchronous	Array processor model	SIMD	Burroughs BSP, ICP-DAP, Illiac IV, Goodyear MPP, Connection Machine
coarse/medium granularity models	asynchronous	Shared memory model	MIMD	Sequent Balance, Encore, Cray-X/MP, Alliant FX/8
	asynchronous	Message passing model	MIMD	Intel iPSC, Transputer

Fig. 2. Computational models.

1) Pipelining and Vector Processing: The basis of pipelining is to take an operation and divide it into several stages, one feeding into the next. Throughput is improved since we are able to perform many subtasks simultaneously, at different stages of the pipe. A factory assembly line is a familiar example, with each station on the line containing a product at some stage of completion, and all workers performing their tasks simultaneously. Pipelining is effective only when the pipe is kept nearly full, with new operands being fed in as fast as the pipe can accept them. If gaps appear, then fewer operations will be done in parallel, and overall performance suffers. Vector operations provide an excellent way to keep a computational pipeline full.

A vector instruction applies the same operation to all elements of a vector (or more likely, to corresponding elements of a pair of vectors). Some setup time may be required to configure the pipe to perform the particular operation, but then the operands can be fed into the pipe as fast as memory can deliver them. There is no need to pause to fetch a new instruction nor to ponder a conditional branch.

The primary model of computation on a vector machine, then, is a simple operation, or a combination of a few simple operations, being performed repeatedly over a block of data. Such operations are characterized in source code as small, tight loops.

Pipelining as a feature of computer architecture is in widespread use, not limited to vector processors, and is usually transparent to the computer programmer. Indeed, on vector machines the normal programmer's model is that of vector operations and loops as described above, not of pipelines. Pipelining, however, is also useful as a model of parallel computation at a higher level. A programmer may explicitly divide a task into several stages, ranging in size from a few instructions to many subroutines, feeding each other and executing concurrently. This is one of the typical models of computation on MIMD machines. Systolic arrays [15] and wavefront processing [16] can be thought of as two-dimensional pipelines.

2) SIMD Machines: The SIMD machine consists of many identical processing elements, each with its own data memory, but all executing exactly the same program. Obviously, such a machine can achieve very high performance by incorporating a large number of processors, but only if the task is such that all the processors do the same thing. The model of computation on an SIMD computer is very much like that of a vector processor: a single operation is performed over a large block of data.

Unlike the constrained pipeline operation of the vector processor, the array processor (an equivalent name for most SIMD machines) can be much more flexible. The processing elements are general programmable computers, so the task performed in parallel can be quite complex and can include conditionals. The usual manifestation of this computational model in source code is about the same as the vector operations: loops ranging over the elements of arrays, where the values produced in one iteration of the loop are not needed for use in another iteration.

Since the models of computation on vector and array processors are so similar, the two are frequently discussed as if they were equivalent. For many purposes, a vector processor can be treated as an SIMD machine. The true parallel nature of an array processor makes it more flexible and gives the promise of greater performance. On the other hand, the more sequential operation of a vector pipeline allows data results to be fed back into the computation. Thus, vector instructions can often be used to compute recurrences, which are more problematic on array processors.

3) MIMD Machines: The term multiprocessor covers most of the machines in the MIMD classification, and is frequently used as a synonym for MIMD, as array processor is for SIMD. In a multiprocessor system, each processing element (PE) executes its own program, relatively independently of the other elements. PEs must, of course, be connected in some way, and this leads to a subdivision of the MIMD classification. In a shared memory, or tightly-coupled, multiprocessor, there is a bank of data memory accessible to all PEs. A common bus or a communication network connects the processing elements to the shared memory. In contrast, a loosely-coupled or private memory machine divides all storage among the PEs, and each block of storage is directly accessible only to its associated processor. A communication network connects the PEs to each other.

The basic model of computation on an MIMD multiprocessor is of independent processes occasionally sharing data [17]. There is a great range of variation within this model. At one end of the spectrum there is distributed computing, in which a program is divided into fairly large parallel tasks, consisting of many subroutines. At the other end is the dataflow model, in which each operation in the program can be thought of as a separate process. The operation waits for its input data, the operands, to be sent to it by other processes. When all the incoming data is available, the operation is performed and the resulting value is sent to those processes that need it. Large- and medium-grain dataflow models [7], [18]–[20] take processes consisting of many operations and execute them in dataflow fashion.

4) Multiple SIMD Machines: Many newer supercomputers offer multiple processors, each of which possesses vector or SIMD parallel execution capabilties. We classify such machines MSIMD.

Compilers for MSIMD machines typically provide language-level constructs that allow the programmer to specify coarse-gain parallelism. Within each task, the compiler then automatically vectorizes suitable loops [21], [22]. MSIMD machines can be seen as a way of getting the best of both worlds: fine-grain vector operation for those parts of program that need it, and flexible MIMD operation for other parts of the program.

5) Same-Program Multiple-Data: There is another popular model of parallel computing that does not fit into the usual view of Flynn's classification scheme. In the same-program multiple data (SPMD) model, every processing element executes the same program on a different portion of the data. This model, also called data-parallel computation, is attractive because massive parallelism is most easily obtained by the partitioning of operations over a massive data set. Yet the SPMD mode is not as restrictive as the lock-step instruction synchronization of the array processor. In SPMD, the same high-level computation is performed on each piece of data, but the same low-level machine instructions may not be. Processing elements may take conditional branches independently of each other.

SPMD execution is, in fact, an asynchronous SIMD mode. Ironically, however, of the current hardware SPMD is most

appropriate for machines usually considered to be MIMD. Those machines can allow the necessary independent operation of the processing elements.

In theoretical computer science, the most popular model of parallel computing is the parallel random access machine (PRAM). In theory, a PRAM is a shared memory synchronous computer whose processor pool grows with the size of the input; interprocessor communication latencies are considered to be zero. These assumptions are not realized in hardware, of course. However, mapping a PRAM algorithm to a real shared memory machine is straightforward [23], [24]. If the real machine doesn't contain enough processors then a resource allocator must partition the data into larger grains to be processed by the processors. Ranade [25] has shown how PRAM algorithms can be transformed to run on a butterfly network with only an $O(\log n)$ slowdown. The mapped PRAM algorithm then executes in SPMD fashion on the MIMD machine.

C. Communication Models

One of the distinguishing features of a multiple-processor computer system is the communication network [26] that connects processors to other processors or to memory. The communication model for a multiprocessor system is so important that many performance measures and tuning factors are represented by the ratio of processing times to communication times of tasks [27]. There are two basic models for interprocessor communication: message passing and memory sharing. In a shared-memory multiprocessor, one processor writes to a particular memory location and another processor reads that memory location. To ensure data coherence and process synchronization, communication is often implemented by mutually exclusive accesses to mailboxes in shared memory.

In private-memory architectures, direct sharing is impossible. Instead, processors share data by sending messages over the interconnection network. The effectiveness of a communication scheme depends on communication protocols, underlying interconnection networks, and bandwidths of memory and communication links.

Often, and unwisely, in shared-memory and vector machines communication costs are overlooked, since communication problems are largely transparent to the programmer. Communication overhead does exist on these machines, in the form of bus, memory, and processor contention. As more processors are added to the system, more processes vie for the same data and bus, leading to saturation. The model of shared memory is very convenient for programming, and is sometimes provided as the high-level view of communication on a machine, even though the underlying system is in fact implemented with private memory and message passing. Cm* [28] is a well-known example. Obviously, the communication cost in such a machine is not zero, although it may not be evident to the average programmer. We have labelled such machines "medium-coupled" in Fig. 1, to emphasize that the machine encourages the programmer to use tightly-coupled as the high-level model, even though the low-level model is something else.

In circuit- and packet-switched networks, as communication requirements increase we must worry about network saturation. Here interprocessor communication ties up network resources: links, processors, message buffers. The amount of communication can be reduced by careful functional decomposition of the problem, and careful scheduling of the resulting functions.

III. Resource Allocation

Resource allocation involves assigning system resources to program components. The reverse view is equally valid: assigning program elements to system resources. For example, processor allocation and task allocation address the same issue from different viewpoints. Allocation strategies can be implemented by the compiler, operating system, or programmer. In literature dealing with these issues, the terms "scheduling" and "allocation" are often used interchangeably.

The system resources usually considered for allocation in supercomputing include pipelines, functional units, registers, processors, and primary memories. Communication devices and links, I/O devices, and various secondary data stores can be considered as auxiliary resources in allocation strategies for supercomputing. Program components considered include instructions, loops, data arrays, and tasks. Resource allocation techniques handle two aspects of concurrent execution of a program: correctness and performance. Data coherence, process synchronization, and deadlock avoidance are common issues of correct execution of concurrent or parallel programs. Any resource allocation technique should satisfy these basic constraints. Most research on resource allocation emphasizes pipeline scheduling, vector register utilization, memory allocation, loop scheduling, task partitioning, and task scheduling to get the best performance.

We will now examine the common allocation problems and their solutions for SIMD and MIMD machines. In SIMD machines, parallelism is found in looping constructs, and one of the primary problems is arranging the loop code so that it matches the capabilities of the hardware. A loop may be unsuitable if dependencies prevent parallel execution, if the pattern of memory references causes a bottleneck, or if conditional constructs require more flexibility than a synchronized SIMD machine can provide. Often the code can be transformed into a pattern that is better suited to SIMD execution.

Despite that great difference in target machine architecture, resource allocation for MIMD machines runs into many of the same problems. Loops are still a primary source of parallelism, and patterns of memory reference may be a bottleneck.

A. SIMD Allocation Strategies

1) Machine-Dependent Resource Utilization

a) Array processors: The most distinctive feature of the array processor is the synchronous operation of its resources, such as the processing elements, concurrent memory reference, and inter-PE communication network. All PEs operate in parallel in a lock-step fashion, synchronized by a global control unit (CU) that distributes instructions to all PEs. Most array processors are used as attached processors that are assigned to perform several compute-bound processes. Best performance, of course, is obtained when the program is structured to keep all processors busy. As the processors are highly synchronized, the idle pro-

cessors cannot be reassigned to some other task. Inter-PE communication is also performed synchronously under the control of the CU. Communication-bound processes are not suitable for these types of processors, because all processors must perform communication or be idle at the same time (even if some of them do not actually need communication) [29].

As PEs are synchronized, they all fetch data memory simultaneously. If processors attempt to access a shared memory unit, the resulting contention may reduce performance. The ideal case is realizable only if the individual data elements are appropriately distributed. That is, the data should reside in private memories attached to be corresponding PEs, or be distributed across modules of a shared memory bank.

If the distribution of data is not a good match for the memory access pattern of the code, then execution will be slowed by memory contention or message passing. Because memory access on a processor is much faster than inter-PE communication, finding a data-mapping scheme that enables concurrent memory reference is crucial for fast computation. For example, if a two-dimensional matrix is partitioned by rows and each row is allocated to one processor (or one memory module), then column-wise element references can be performed fully in parallel while row element references cannot. A matrix manipulation in which both rows and columns are frequently referenced cannot be executed efficiently.

Such a case is found in a matrix computation, $A \leftarrow B \times C$, where A, B, and C are two-dimensional arrays. Fig. 3 shows the distribution of parallel computations on n processing elements. If we partition the arrays B and C row- or column-wise, then to access the elements of B and C would require repeated, expensive inter-PE communication. This communication could be eliminated by storing the entire arrays B and C in all processors before computation.

"Skewing" an array is a similar technique to provide more chance of array accesses without communication during the computation. In a "skewed storage" scheme, columns or rows are cyclically shifted across the memory modules so that both column and row elements can be simultaneously referenced [30].

A fine-grain approach to SIMD computation is found in systolic array implementation. The idea behind the systolic array is to construct an application-oriented processor in which every processing element takes part in a few operations and passes the result to neighbors for successive operations. Systolic arrays, therefore, perform *multidimensional* pipelining operations, in which processing elements do not (always) perform the same operation. In some sense, resource allocation for systolic arrays is much closer to the allocation of vector processors that is described in the next section.

b) Vector processors: A pipeline requires an initial start-up time, or *latency*, which depends on the number of stages in the pipeline. After this lag, the functional units can turn out one result per clock period as long as a new pair of operands is supplied to the first stage every clock period. Thus, the steady-state performance is independent of the length of the pipeline, and depends only on the rate at which operands are fed into the pipeline. As a result, effectiveness of the pipeline is mostly determined by the number of identical operations and the number of operands to be fed, which are defined by the patterns of the user's applications.

Recent large-scale vector processors consist of a large number of arithmetic units, in which the individual units are highly pipelined [31], [32]. Multiple functional units can be cascaded to perform longer vector operations, forming

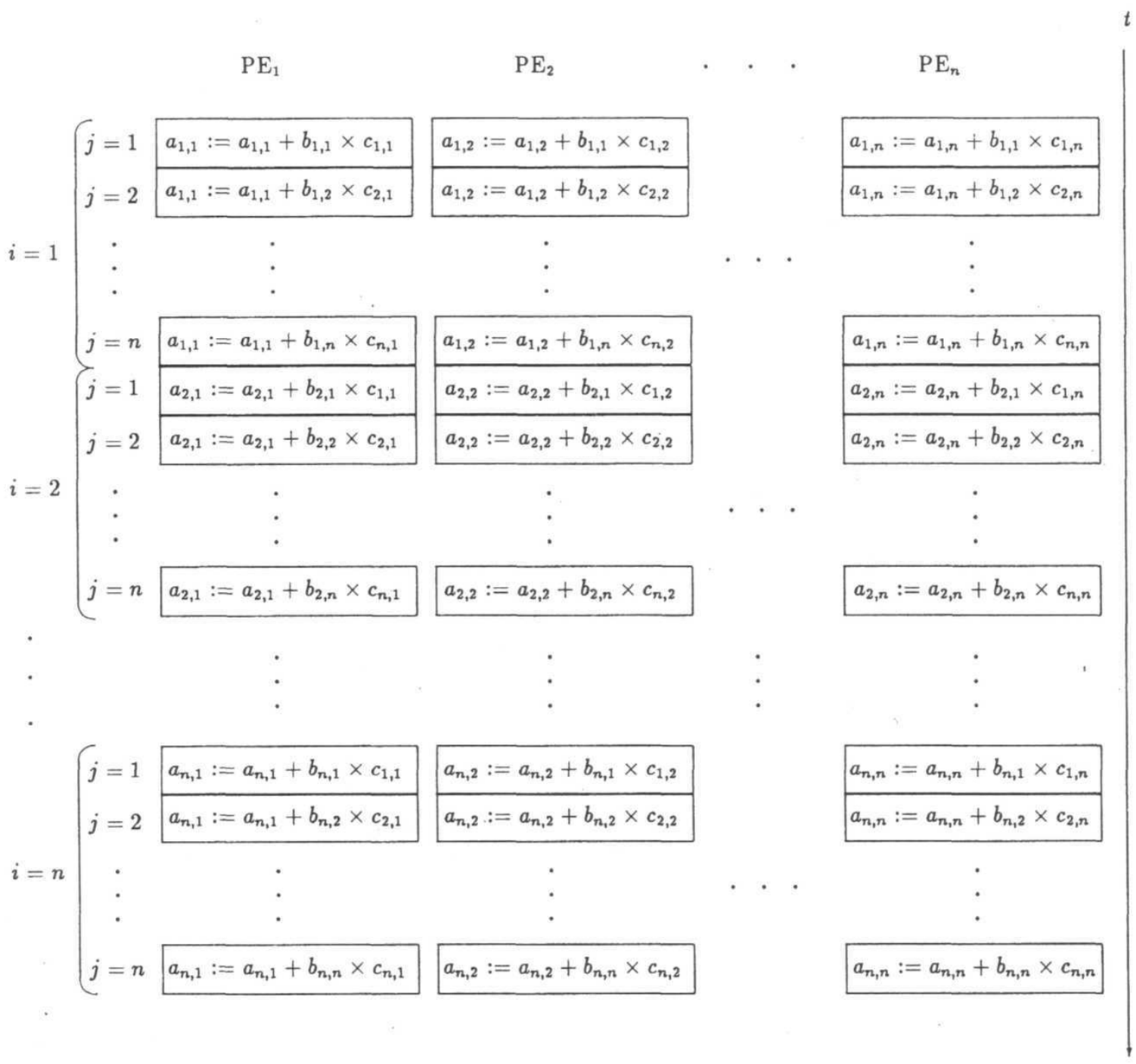


Fig. 3. Matrix multiplication on an n-processor SIMD computer.

a "supervector" operation. Simple supervector operations can be automatically identified in a program. An example of supervector operation is a vector operation

```
for i := 1 to 100
  a_i := a_i + s × b_i
```

where vector multiplication and vector addition are cascaded to obtain a higher computation rate than is produced by activating one pipeline following the other. Fig. 4 shows how chaining of two pipelines allows the second pipeline to begin computation before the first is completely finished.

We can compare the problem of combining vector operations into chains, or supervector operations, to the MIMD grain compaction techniques discussed below. In both cases, we must identify operations that interact closely, one producing results to be used by the other. Having identified such operations, we group them together to reduce overhead—the overhead of memory access in the case of a vector machine, the overhead of message passing in a private-memory MIMD machine.

Pipeline and array processors are somewhat similar to each other in terms of synchronized operation. Because vector operation can perform only simple combinations of operations, only the innermost loop of a set of nested loops can be vectorized. Such a restriction of pipelining operations allows a somewhat simpler environment for the pipeline processor applications than array processors.

Code transformations can improve the performance of a program on a vector processor. For example, one can select from a (multiply) nested loop the one loop that is most suitable for vector execution and *interchange* the loops so that the selected loop becomes the innermost loop [33]. Loop interchange can also change the *stride*, the addressing increment between successive accesses to a vector. In Fortran terms, the stride is the increment in the left-most subscript position. A reduction in the stride will mean that successive memory references go to more closely adjacent vector elements, which will usually improve memory performance. For instance, a loop

```
for i := 1 to 100
  for j := 1 to 100
    a_{i,j} := a_{i,j-5} + b_{i,j}
```

would be rewritten in a vector loop

```
for i := 1 to 100
  for j := 1 to 100 step 5
    for k := 0 to 4
      a_{i,j+k} := a_{i,j+k-5} + b_{i,j+k}
```

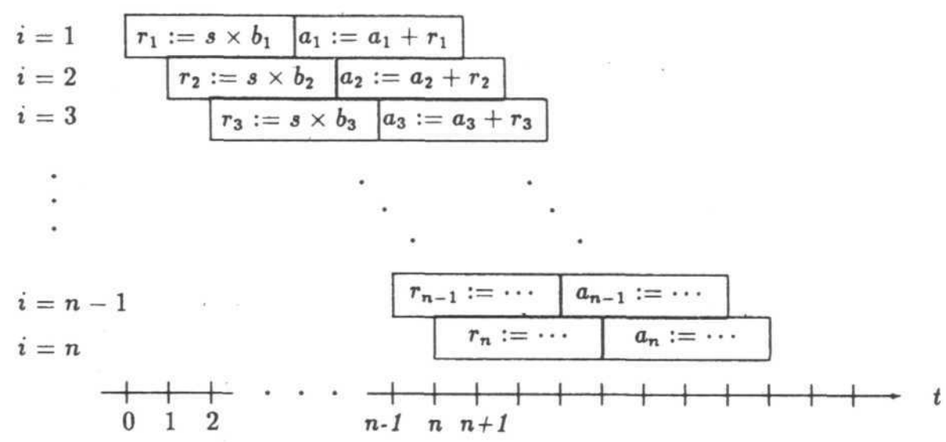
The inner loop is vectorizable, but the vector length is only 5 elements. A better result would be obtained by the following loop

```
for j := 1 to 100
  for i := 1 to 100
    a_{i,j} := a_{i,j-5} + b_{i,j}
```

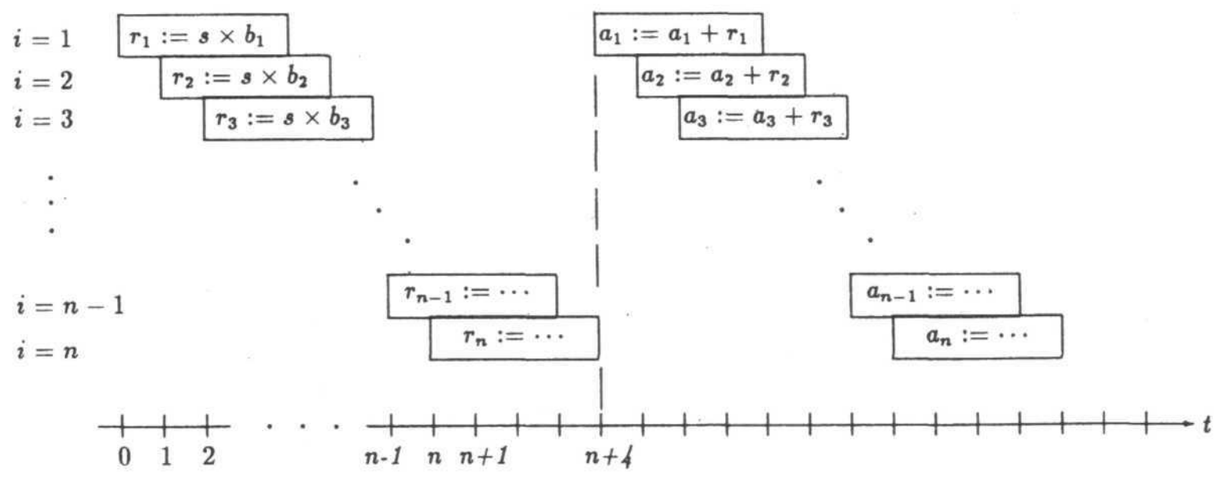
in which 100 successive memory references are possible.

The basic idea behind increasing the stride is that pipeline processors favor the use of long vectors. That is, the longer the vector fed into the pipeline, the less effect latency has on the overall performance. If interchanging loops does not produce an efficient memory reference pattern, then a restructuring of the data storage, as outlined above, might be necessary. To maintain a long vector for a better result, the following guidelines can be used:

- Use separate scalar variables instead of an array of very short dimension



(a)



(b)

Fig. 4. Pipelined computations a) by supervector operation and b) by two independent pipeline units.

- Use few long dimensions instead of using several short ones
- Copy short vectors into a longer temporary vector.

c) Vector registers: A vector processor may store its data solely in main memory, or may supplement memory with a set of vector registers. By storing vectors in main memory, a machine like the CDC Cyber-205 can allocate large vectors freely and provide virtually infinite working space in which to put data and temporary results. In this type of vector machine, however, fast memory access is crucial for high-performance operations. An interleaved memory scheme permits parallel memory references, and several requests are simultaneously serviced in a pipelined fashion.

The effectiveness of interleaved memory organization depends on the vector allocation strategies. If a vector resides on a single memory module, the memory subsystem can provide only one vector element per memory cycle, and the vector operand fetch rate will not satisfy the demand of the pipelines. On the other hand, if a vector is properly distributed among memory modules that can be (almost) simultaneously accessed, then memory will be able to keep up. Normally, interleaving is handled by the memory hardware design, and a logically contiguous vector is automatically spread across memory modules. As discussed above with array processors, however, if the program references both columns and rows frequently, skewed storage is preferred to prevent unexpected operand access delay due to memory contention, as in array processors [30].

Vector storage in vector registers raises some problems, as the number of registers is limited. A compiler must carefully allocate registers to minimize loading and storing of data from and to main memory. Long vectors must be split into segments that will fit into a vector register [34] simulating a doubly nested loop. The following loop

```
for i := 1 to n
  a_i := b_i + c_i
```

would be modified into a doubly nested loop

```
for ik := 1 to n step σ
  δ := min (n, ik + σ)
  for i := ik to δ
    a_i := b_i + c_i
```

where the innermost loop is (directly) vectorizable while outer loop is a sequential loop. The length of segment σ is given by the vector register size. Such a loop segmentation overcomes vector register size limitations, but it requires some overhead for the outer loop computation. Despite the outer loop overhead, the vector register approach promises to improve performance, as vector registers can be implemented with a faster memory technology and thus can deliver operands to the pipeline better than main storage can.

c) Resource Utilization by Programming: The basis for most vectorization in a program is the loop. To automate the process of loop vectorization, the compiler must do two things. First, it must locate the places where vectorization is possible. The compiler must then determine how best to exploit these vector operations on the target machine. If loop bounds for the loop are known at compile time, a compiler can perform analysis to determine which loop vectorization will execute fastest. It is possible that scalar execution might be faster than vector execution, if the loop is small. However, if the loop bound is not known at compile time, this analysis cannot be done, and the decision must be made by default, or with some additional information provided by the programmer. If a loop is not suitable for vectorization, there may be a way to transform it to an equivalent computation that *is* vectorizable. Several program transformations are available to the compiler that can enhance vectorization, including statement reordering, statement substitution, and loop interchange [1]. The idea behind these transformations is to reduce the number of dependencies between loop iterations; most of them are machine independent.

A simple case of vector operation is a loop in which there are no dependencies between iterations. Such a loop can be vectorized by partitioning the loop body into several simple loop bodies in which every loop body is a few pipeline operations that are allocated to a corresponding pipeline(s). Fig. 5(a) shows the case with two statements inside a loop where no dependencies exist. This loop can be transformed into two independent loops

```
for i := 1 to n
  s1;
for i := 1 to n
  s2;
```

and these two vectored loops can be targeted for multiplication and addition pipelines. Similar transformations are possible for several loops that have some dependencies inside loops. The original loops and data dependence relations are shown in Fig. 5. The structure of the vectored instructions for the next two loops would be identical to Fig. 5(a).

```
      for i := 1 to n                for i := 1 to n                for i := 1 to n − 1
s1      a_i := b_i × s       s1      a_i := b_i × s       s1      c_i := a_{i+1} × s
s2      c_i := b_i + d_i     s2      c_i := a_i + d_i     s2      a_i := b_i + d_i
        (a)                            (b)                            (c)
```

Fig. 5. Directly vectorizable loops a) s_1 and s_2 have no dependences; b) modified values of s_1 is used in s_2; c) s_1 uses a value before modified in s_2.

a) Resolution of conditionals: Performance of the pipeline is reduced by "irregular" operations, such as interrupts and branch instructions. An interrupt instruction prohibits the next immediate instruction from entering the pipeline until the pipeline is drained, and a branch instruction may be halfway down the pipeline before a branch decision is made. At this point, all prefetched instructions become useless. To alleviate this problem, pipelined hardware frequently incorporates techniques such as multiple pipelines or delayed branches. For extensive techniques for pipeline branch instruction manipulations, refer to [35].

The frequency of conditional statements within loops makes most large-scale vector processors equip several effective conditional branch processing approaches based on the mask vector register [31], [32]. A mask vector indicates the true–false values of conditional statements, and controls the arithmetic pipeline units. A computation result is discarded if the associated mask vector element is false, while the result is stored back to a vector if the mask vector element is true. This architectural feature enables vectorization for the loops with *conditional operations*. However,

a better result can be promised by reducing conditional operations inside loops as much as possible by rewriting them without conditional operations. The following loop

```
m := ···
for i := 1 to n
  if (m < n) then
    a_i := b_i × s
  else
    a_i := b_i/s
```

could be rewritten using separate loops for each comparison statement:

```
m := ···
for i := 1 to n
  if (m < n) then
    a_i := b_i × s
for i := 1 to n
  if (m ≥ n) then
    a_i := b_i/s
```

Now each loop is vectorizable and comparison statements control the mask vector registers. The example loop can again be rewritten to a loop that has no comparison statement:

```
m := ···
for i := 1 to m
  a_i := b_i × s
for i := m to n
  a_i := b_i/s
```

Unlike conditional operations, *conditional control* cannot be vectorized, because a branch causes "irregular" operations of the computation. One transformation technique for loops with conditional control is IF-conversion, which transforms loops with conditional control into loops with conditional operations [36].

b) Resolution of recurrence: A recurrence is an iterative computation in which the computation of the current value depends on a value from a previous iteration. This data dependency between iterations prevents execution in parallel, although simple recurrences can sometimes be handled by vector pipelines. Transformations of the loop may remove the recurrence by reordering the two statements, by splitting the loop into two or more loops, or by duplicating computation and substituting them into other statements within the loop body [1]. Consider a loop

```
for i := 1 to n
  a_i := b_i × s
  b_i := a_{i+1}
```

such that the used variable b_i is modified soon. The loop can be vectorized by using temporary vector t_i restructuring the loop (which is known as node splitting)

```
for i := 1 to n
  t_i := b_i × s
  b_i := a_{i+1}
  a_i := t_i
```

where three statements are recognized as vectored instructions. Another possible approach is to substitute statements producing a loop

```
for i := 1 to n
  a_i := b_i × s
  b_{i-1} := b_i × s
```

where two statements inside the loop are also vectorizable.

B. MIMD Allocation Strategies

The flexibility of MIMD machines leads to a variety of problems in allocation. As the programs executed by individual PEs can be practically anything, there are many possible ways to divide an application and distribute it among the processors. Highly independent processes may be arranged in a pipeline. Loops may be parallelized and all iterations performed simultaneously. Small pieces of a computation may be divided and executed dataflow-style. Choosing among these possibilities is a matter of balancing the characteristics of the program and the capabilities of the machine to achieve a lot of parallel execution and little communication overhead. The size of the parallel program components and the style of scheduling them are the two most commonly considered parameters.

1) Granularity and Communication Costs: One can view parallelism extraction from programs as partitioning a program into concurrent tasks called *grains* [37]. Grain size may vary anywhere from an elementary mathematical operation (fine grain) to an entire program (in this case we are back in sequential mode). There is a trade-off: With fine grains, there is likely to be a great deal of potential overlap of computations, but also a great deal of overhead in scheduling the grains for execution and in communicating data values between grains. For very large grains the situation is reversed: Overhead is reduced but so is parallelism. Overall performance can apparently be improved by finding a happy medium [7]. See Fig. 6.

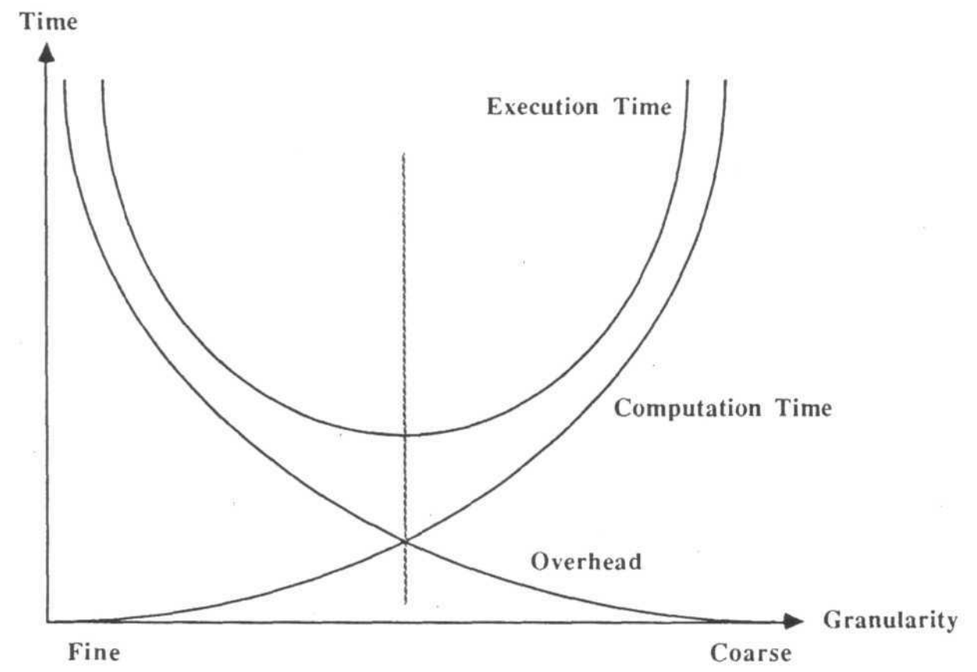


Fig. 6. Trade-off between parallelism and overhead.

Medium granularity coalesces several fine grains in such a way that decreases the intertask synchronizations without sacrificing the parallelism within the tasks. A typical medium granularity partioning is loop parallelization, in which individual loop iterations are considered as independent tasks. In this case the speedup depends on the loop bounds. Nested loops provide the opportunity to exploit more parallelism of the programs by interchanging the loops. Sim-

ulation [38] shows that the medium granularity approach based on loop parallelization achieves fairly good speedup.

More effective parallelization techniques try to exploit potential parallelism beyond the loop bodies [7], [9], [19]. They begin exploiting parallelism from the basic blocks that are a sequence of instructions having no branches except at the end of the block, and then expand detection of parallelism for the entire programs that are represented by global control flows among basic blocks. As each basic block contains only simple, statically determined data dependencies among its component operations, optimum basic block partitioning is easily established [39].

Kruatrachue and Lewis [19] proposed a grain-packing algorithm in which the grain size is automatically determined in a way that reduces the execution times of the basic blocks by balancing execution time and communication time. The basic technique of this method is to duplicate the small grains where necessary to reduce overall communication delays.

Another source of parallelism of the program is detected from control flow constructs such as loops and branch instructions. Since the number of iterations of a loop and the results of a conditional branch are usually determined at runtime, it is usually difficult to utilize the entire potential parallelism of global control flow. A global fine-grain compaction methodology has been proposed in [40], in which the control flows of the programs are heuristically guessed during compile time. With these guesses, global control operations are treated as ordinary basic blocks so that parallelism detection is easier. The key to performance is the accuracy of the guesses.

Because private-memory machines perform better with larger-grain parallelism, it is advantageous to find that the outer loop in a set of nested loops is parallel. The same loop-interchange techniques used to move parallel loops to the interior—where vectorization can be done—can also move a loop to the exterior, where large-grain parallelism is possible.

2) Scheduling: Given a program that has been divided into parallel tasks, the job of a scheduling strategy is to determine the order in which those tasks run, striving as always to improve performance. If the order is determined at compile time, it is called *static scheduling*. Static scheduling relies on predictions of execution times of partitioned tasks and communication costs (or amount of message traffic) between tasks. The success of static scheduling depends on the accuracy of the prediction. In many cases, such predictions can be quite reliable; van den Bout [41], for example, found that signal and image processing applications exhibit very regular behavior. In other cases, of course, a priori estimates of program behavior will be inaccurate and static scheduling will be inferior to dynamic scheduling. The attraction of static scheduling is that the scheduling cost for a given program is incurred only once, at compile time.

Dynamic scheduling is performed at run time. The goals are the same as in static scheduling, to arrange tasks onto processors so that processors are kept as busy as possible and to minimize wasted effort, such as time spent waiting for communication. Dynamic scheduling does not depend on predictions of program behavior, instead making decisions using information collected from the current execution. Thus it is more flexible and potentially more accurate, but at the cost of additional run time overhead.

Tanenbaum and van Renesse have pointed out [42] that task allocation techniques with the goals of maximizing throughput, mimimizing response time, and keeping the load uniform have somewhat opposing views depending on their assumptions about what is known and what is most important. Three classes of scheduling techniques show this relationship.

- *Coscheduling:* Due to the overhead of context-switching, processes that communicate frequently should run simultaneously on different processors [43]. By putting all the members of a process group that work together on different processors, one has the advantage of *N*-fold parallelism, with a guarantee that all the processors will be run in parallel to maximize communication throughput.
- *Clustering:* Interprocessor communication overhead causes delays. To cluster processes that communicate frequently on the same processor minimizes communication costs. This technique relies on the premise that communication costs can be statically determined, yielding a good static schedule. This is the technique used by [7], [9], [19] where a single task is being partitioned and scheduled.
- *Load balancing:* If we know nothing about the future communication patterns of a process, then it is best to distribute work loads evenly across the processors to preclude any process from becoming a bottleneck. To keep a uniform distribution of load (*balanced load*), each processor estimates its own load and processes are created and migrated. Load balancing is discussed in more detail in the next section.

Instruction scheduling is the process of scheduling instructions on a processor to maximize throughput. In the context of instruction pipelines [44], where several instructions overlap execution phases, a common performance metric is the number of instructions that are issued per clock cycle. Because the possibility exists that instructions may interfere with each other, it may not be possible to keep the pipeline full.

Another type of instruction scheduling is used for VLIW (very long instruction word) architecture. Here, many operations expressed in a high-level language are folded into one very long instruction word. The instruction word controls, in parallel, all of the functional units of the processor. The task is to pack as many operations into one word as possible. The more concurrency that can be found in the code the more packing that can take place.

Scientific programs are deterministic in the sense that control-paths through the programs tend to be predictable because they are generally *data-independent*. Knowing these control paths at compile-time allows us to pack instructions into an instruction word much more efficiently than if we did not know control-paths statically. For example, when a branch is taken that was not predicted when scheduling took place, part of the instruction word will be wasted, incurring a loss of parallelism.

Trace scheduling [40] is a technique that treats many adjacent basic blocks of program code as one basic block. This allows parallelism to transcend the artificial boundaries of

a basic block. The basic idea is to pick an execution path, or *trace*, from a program and compile it as straight-line code. We will have to make conjectures as to which way conditional branches will go at run time. Much research has been done on this topic; one technique is to always predict that backward branches will be taken. The reason is that, in all likelihood, it is a branch in a loop statement that will fall through at the end of processing an array of data [35].

Trace-scheduling can be viewed as static scheduling of a multiple pipeline. All instructions that can execute simultaneously are loaded into the multiple functional units of the pipeline. In order to mimimize pipeline delays in the presence of conditional branches, trace-scheduling executes several different branches simultaneously, ultimately discarding results from the branches not taken. This strategy is equivalent to multiple pipelines.

3) Load Balancing: Load balancing as an MIMD resource allocation strategy is based on a key performance enhancement issue that overall processor utilization can be achieved by distributing system load uniformly on all processors. Once we have decided that a problem is amenable to load-balancing, questions about static and dynamic techniques arise. Because of the run-time-dependent nature of load, static load balancing is sometimes called load-balanced task allocation [45]. Saltz *et al.* have worked on a comparative analysis of static and dynamic load-balancing strategies [46].

Dynamic scheduling methods [46]–[49] inherently resolve the load-balancing problem at runtime. That is, tasks are created on or migrated to less loaded neighbor processors to keep all processors as busy as possible. All processors are kept busy until a precedence relation between two or more processors prohibits their further execution.

Load balancing in static scheduling looks at the possible communications between tasks and tries to arrange tasks to minimize time spent waiting on communication. Graph-theoretic methods model this balancing problem as a minimum-cut/maximum-flow network problem. Although optimal static algorithms are available [50], heuristic algorithms are often used, which trade off optimality for speed [45], [46]. For example, list-scheduling techniques [51], [52] are a class of static scheduling methods in which tasks are assigned priorities and inserted in a priority queue. Whenever ready tasks contend for processors, the next process to be assigned to a free processor is the one at the head of the priority queue. This characteristic of list scheduling tends to evenly distribute tasks over all processors, that is, balance the load. It is generally known that list-scheduling strategies yield optimal load balance on an "ideal" parallel computing machine in which synchronization overhead is negligible [53]. These scheduling strategies are not adequate in real parallel computing environments because in practical parallel machines' synchronization overhead is not negligible.

4) Memory Allocation: In shared memory MIMD machines, the memory is typically partitioned into modules so that multiple memory references can be simultaneously served. Issues of memory interleaving and array distribution are similar to those for vector and array processors. Memory conflicts and race conditions can arise on any variable, if two or more processors attempt to simultaneously access or update the variable.

If a variable is frequently accessed by several different tasks, multiple copies can be distributed among the memory modules so that multiple accesses can be concurrently served. The duplication scheduling strategy of [19] might be considered an extreme example.

A race condition occurs if more than one processor tries to read and write a shared variable at the same time. Some sort of synchronization, such as a critical region or a semaphore, must be employed to assure proper results. In some cases, variable renaming [14] can solve the problem without a need for explicit synchronization. Different uses of a variable are split into separate variables, removing the conflict entirely. In private memory MIMDs, no variables are directly shared. Copies of a variable residing on separate processors can be thought of as being implicitly renamed. The message-passing event with which a value is shared with another processor also provides synchronization, albeit at a noticeable communication cost.

If multiple copies of global variables exist in MIMD machines, there may be a coherence problem, as distinct copies of the global variables must be kept consistent. Several approaches would be possible to solve this coherence problem for shared-memory MIMDs. One approach is a "static coherence check," which avoids multiple copies of shared variables and allows only one copy of global variables. A second approach is "dynamic coherence check," in which multiple copies of global variables are allowed, but they are discarded when the global variables are modified [54]. A more systematic approach is proposed in [55], in which the compiler detects the variables that can cause a coherence problem and carefully allows multiple copies of them. As Lee *et al.* [55] point out, this approach is quite efficient, although a compiler must do more work.

In a private-memory MIMD, more severe coherence problems may arise. The primary objective of resolving the coherence problem here is to ensure that correct values are used whenever a global variable is referenced. This problem can be resolved by inserting a message-passing call for each global variable as it is updated. The communication primitives can be inserted at algorithm development phase, either manually or by versatile compilers which analyze the program precedence relation, task partitioning, and scheduling at compile time.

5) Loop Scheduling: As loops in a program are the major source of parallelism and can easily consume a large portion of computing resources, allocating loops onto processors is an important issue for supercomputing. There has been a considerable growth of interest in detecting parallelism of loops, in scheduling loops, and in efficient implementation [1], [48], [49], [56]–[58]. The major difference of loop processing between MIMD and SIMD processors is that MIMD multiprocessors can schedule multiple nested loops to processors while SIMDs allow only a single loop parallelization.

a) Loop structure: Loop structures can be classified based on their execution behavior [58]. Loops in which every iteration can be evenly distributed onto the processors and can simultaneously start with no interaction between processors are called DOALL. Such loops can effectively utilize any kind of parallel computer, including SIMDs. Loops that contain dependencies between iterations are not directly parallelizable, and parallelization of such loops may even reduce performance due to interprocessor communication overhead. A loop in which the next iteration can begin only after the current iteration has finished is clearly a nonpar-

allelizable loop, and every iteration of the loop must be executed sequentially to ensure the correct result. Such loops are called serial loops (DOSER). For many loops with dependencies, a certain amount of parallelism can be exploited. Such a loop is one in which the next iteration can begin only after some delay T_d after the current iteration begins. If T_d is less than the execution time of the loop body, then distributing iterations to processors would clearly shorten the completion time of the loop execution. Suppose T_l is the execution time of the loop body. Then, the percentage of overlap that is possible within a loop is measured by T_d/T_l. If $T_l \leq T_d$ then the loop is strictly a DOSER loop; if $T_d = 0$ then it is a DOALL loop; and otherwise it is a class of parallelizable loop known as a do-across (or DOACR) loop. In the global sense, DOALL and DOSER loops are special cases of DOACR.

A more systematic representation of T_d is to express it by the distance of the dependency or the recurrence factor δ, that is, the number of iterations crossed by the precedence relation, and the communication overhead to reference the dependent variable(s) across δ-iterations. The following loop has a data dependence that crosses iterations:

```
        for i := 11 to n
S1        a_i := b_i + d_{i-10}
S2        d_i := e_i + a_i
```

This loop would be a non-DO ALL loop with $\delta = 10$. The loop is, however, partially parallelizable because the modified array elements of the vector d are used after 10 iterations have passed, producing the parallel loop

```
        for pe := 0 to 9
          for i := 11 + pe to n step 10
S1          a_i := b_i + d_{i-10}
S2          d_i := e_i + a_i
```

where the outermost loop distributes iterations to processors 0 to 9, utilizing 10 processors with no delay. The loop can utilize no more than 10 processors. This technique is known as cycle shrinking. Cycle shrinking is generally not used for vector processing unless δ is big enough to utilize a pipeline.

Parallelism is severely restricted when δ is small. Consider a non-DOALL loop such that $\delta = 1$:

```
        for i := 1 to n - 1
S1          :
            .
S2        a_i := b_i + d_i
S3          :
            .
S4        d_{i+1} := e_i + a_i
S5          :
            .
```

Statement s_2 references an element of array d that has been computed in the previous iteration. Suppose that iterations k and $k + 1$ are distributed to two processors p_k and p_{k+1}, respectively. Then p_{k+1} could execute s_2 only after d_{k+1} has been brought to the processor. If the communication overhead to pass d_{k+1} exceeds the speedup obtained by executing s_1 in parallel with the statements of the previous iteration, then distributing iterations to processors will not provide better performance than allocating all iterations to a single processor. Such a metric is useful to identify non-DOALL loops into DOACR and DOSER loops [7].

b) Static scheduling: Loop scheduling can be done statically or dynamically. Static scheduling assumes that the loop bounds are known at compile time, dependence relations within the loop are fixed throughout the execution, and execution time of an iteration is consistent. Then, all iterations can be evenly distributed to the processors by the compile-time scheduler [14], [58]. The main advantage of static scheduling is its low run-time overhead and its straightforward task execution mechanism that provides easy execution tracing and debugging.

Often, however, dependence relations and execution time of each iteration are unpredictable because of conditional branches inside the loops. And frequently the loop bounds are sometimes unknown at compile time. Despite the difficulties, static scheduling is widely applied to supercomputing, particularly for the message-passing MIMDs, to prevent excessive scheduling overhead owing to expensive communication overhead.

For simple loop structures, optimum results could be expected by careful implementation, applying a few of the loop parallelization techniques described. Notice that the optimal static scheduling algorithms usually involve polynomial or even exponential complexity.

c) Dynamic scheduling: Dynamic scheduling can allocate iterations at run time, in which an iteration is assigned whenever a processor becomes available [48], [49], [59]. The dynamic scheduling mechanism is a kind of "barrier synchronization" mechanism associated with each loop in the construct to provide loop index synchronization. The basic idea behind dynamic scheduling is to provide mutually exclusive variables that are shared but can be accessed strictly in sequential order as the loop behaves. Such variables include the loop indices, dependent variables, and recurrence factors, that are used mostly for the loop synchronization. Mutually exclusive variables are tested whenever a processor needs to reference the dependent variable so as to determine whether the dependent variable has been ready to be used by the process. Similarly, the processor that has completed producing a dependent variable flags the corresponding mutually exclusive variables to allow other processes to consume the dependent variable.

For a non-DOALL loop shown below,

```
        for i := 3 to n
S1        a_i := b_i + d_{i-2}
S2        d_i := e_i + a_i
```

would be scheduled at run time

```
        for i := 3 to n
I_in      while (not R_{i-2}) wait
S1        a_i := b_i + d_{i-2}
S2        d_i := e_i + a_i
I_out     set R_i
```

where $n - 2$ iterations are dynamically distributed, and test simultaneously whether the mutually exclusive variable R_{i-2} has been set. If R is not yet set, then iterations are blocked at I_{in} statement, and if R is set then the processor that performs the corresponding iteration would continue s_1 and s_2. Once a processor has finished s_2, it has the right to invade the mutually exclusive area and set R_i. This is done in I_{out} statement.

Here a barrier synchronization counter is R that bookkeeps the iterations that have already completed. If the

value of a barrier counter is equal to a loop bound, the next immediately active instance is activated when the barrier counter becomes zero. As shown in the example, the synchronization instructions ensure that data dependencies between iterations are preserved. With synchronization instructions inserted, non-DOALL loops are transformed into DOALL loops, and then they can be allocated as DOALL loops [57].

Tang and Yew [59] propose a self-scheduling technique in which one iteration is allocated to an idle processor by increasing the loop indices in such a way that promises a synchronized iteration as discussed in the previous example. The loop scheduling, therefore, involves n dispatch operations where n is the number of iterations. Because self-scheduling allocates one iteration at a time, it would promise the best performance in terms of load balancing and processor utilization. It is not always optimal, however, if n times of iteration assignment overhead is not negligible compared to the execution cost of the loop. For instance, n iterations on p processor system would incur n/p assignment where p iterations are assumed to be dispatched simultaneously.

Rather than issuing a system call to the operating system one iteration at the time, processors can schedule themselves by "fetch-and-adding" a shared variable to get loop indices of a chunk of iterations. The main self-scheduling code for each component consists of three parts: loop self-scheduling, the original loop body or instance, and bookkeeping of the completed iterations. Suppose m is the number of iterations for chunks. Then the number of loop assignments at run time would be reduced to $1/m$ of the self-scheduling. However, it is hardly possible to determine the optimum chunk size for the best result, and the balanced load distribution would be lost. Guided self-scheduling resolves the difficulties of loop partitioning by a simple strategy that decreases the chunk sizes by 1 for every chunk allocation [48]. At the beginning, the chunk size is equal to that of the extreme case of the chunk allocation (chunk size is n/p), and at the last stage, the chunk size holds one iteration (as self-scheduling does). Thus, guided self-scheduling compromises the two extremes of dynamic loop allocations, reducing the run-time scheduling overhead while improving the processor utilization [48].

Dynamic loop scheduling becomes more effectively parallelized by using the technique discussed in Section 3-A. Most parallelizing (vectorizing) techniques are applicable to improve the parallelism of loops. Particularly, loop interchange permutes a pair of nested loops so that the outer loop becomes inner loop and vice versa. The goal of loop interchange is that the longest possible set of parallel iterations corresponds to successive values of the outermost loop of the nested loops.

Loop coalescing transforms a series of multiple one-way nested DOALL loops into a singly nested DOALL loop [48]. A DOALL loop

```
for i := 1 to n
  for j := 1 to n
    a_{i,j} := b_{i,j} + ···
```

would be transformed into a single loop

```
for i := 1 to n^2
  ik := ⌈ i / n ⌉
  jk := i − n ⌈ (i − 1) / n ⌉
  a_{ik,jk} := b_{ik,jk} + ···
```

where the result requires n^2 iterations with one loop. When loop coalescing is applied, the overhead associated with the access of loop indices is reduced.

Loop distribution distributes a loop around each statement in its body, or around code modules inside the loop that form strongly connected components of the precedence relation. It is useful for transforming multiway nested loops to one-way nested loops

```
for i := 1 to n
  a_i := b_i + ···
  c_i := a_{i−1}
```

would be transformed into two DOALL loops

```
for i := 1 to n
  a_i := b_i + ···
for i := 1 to n
  c_i := a_{i−1}
```

The modified loops have no dependencies and do not require loop synchronization between two loops, but the number of total iterations to be scheduled is increased by $2n$. Thus, the decision to use distribution to synchronize a loop must be based on the trade-off between the start-up and scheduling overhead for the parallel loop.

IV. The Relationship Between Models and Allocation

Resource allocation is a problem common to all "real" supercomputers and, as we have seen, is a difficult problem to solve. Consequently, theoretical computer scientists involved with designing parallel algorithms have found it helpful to use idealized machine models that ignore hardware constraints, thereby freeing themselves to concentrate on the exploitation of parallelism in the given problem domain.

Programmers code in high-level computational models, which must then be translated by a compiler and resource allocator to fit the low-level machine model. Good resource allocation is difficult and optimality is in general NP-complete. A resource allocator requires that the programmer still program in an "intelligent" fashion—one suitable to a machine model. Well-known vectorizing compilers translate sequential models of Fortran to vector or array computation models. We will now examine two other environments and the ways that the high-level model is translated to a low-level one.

A. SISAL

SISAL is a single-assignment data-flow language that is a descendent of Val [12], [60]. SISAL provides many implicit and explicit language characteristics that exploit parallelism, which must be mapped to a machine. Loops in SISAL are parallel unless explicitly stated as sequential. The *single assignment rule* reduces data dependencies, thereby increasing exploitable parallelism. The *stream data*-type in SISAL abstracts a FIFO data structure associated with producer–consumer, *i.e.* pipelined, process relationships.

Data flow graphs represent fine-grained parallelism—generally a node for every operation, with arcs between

nodes representing communication requirements. SISAL is an attractive language to parallelize because it disallows many of the things that make it difficult to parallelize traditional imperative languages such as Fortran, Pascal, C [61]. For example, SISAL prohibits side effects.

1) The High-Level Computational Model: Can programs written in SISAL be targeted on medium to coarse-grain multiprocessors? Sarkar and Hennessy [9] have developed a static partitioning and scheduling scheme that takes SISAL program graphs as input and produces, at compile time, a task partition and a schedule using a parameterized communication model. Parameters are:

processor component: The duration for which a processor participates in its communication.

delay component: The fraction of the communication time during which the producer and consumer processes are free to execute other tasks.

These parameters in turn depend on the size of the communication, interprocessor distance, cost of reading inputs, cost of writing outputs, and communication delay.

2) Motivation: Standard dataflow analysis techniques can provide precise information on a program's communication patterns. The difficulty lies not within uncovering a program's communication requirements but in modeling a machine's ability to handle communication traffic. Bus and memory contention, synchronization, communication errors, routing strategies, interprocessor distances, and processor contention all tend to make communication modeling nondeterministic. Sarkar and Hennessy's parameterized communication model is robust enough to handle these obstructions, thereby answering our fundamental question of what can be done in parallel and what should be done in parallel.

The dataflow model presents an elegant computational abstraction. Here the actual target machine is not a dataflow one but an arbitrary multiprocessor. Task granularity is a function of the number of processors and interprocessor communication costs. A dataflow graph starts at a very fine grain and grains are continuously combined until a suitable task size is achieved. SISAL, as a language, is machine independent. The programmer assumes that the compiler or some resource allocator will exploit the parallelism (explicit and implicit) as best as it can, given a particular machine model.

Linear speedups on a variety of test cases have been reported. Because SISAL is designed as a language to be parallelized, almost linear speedups have been attained in nonnumerical algorithms. (On a 10-processor simulation, speedups of 10, 9, 8, 8 were attained for Towers of Hanoi, Merge-exchange sort, eight queens, and Multi-precision multiplication, respectively).

3) Two SISAL Implementations: Researchers at Colorado State have targeted SISAL to two different multiprocessors: A Denelcor HEP and a Sequent Balance—both are shared memory computers [62]–[64]. Implementing the dataflow model of computation on a non-dataflow architecture requires careful handling of resources in order to control the possibility of too much parallelism, which could cause resources to become saturated and communication overhead to become unacceptable [65]. SISAL requires significant run-time support because of the need for dynamic allocation of resources. Parallel loop bounds and recursion depth may not be known at compile time, and stream sizes may shrink or expand.

The Sequent implementation is a descendent of the HEP implementation. Many of the optimizations present in high-performance compilers were not implemented in early SISAL compilers. Consequently, the goal of that work has been to efficiently utilize the underlying hardware. This has proven successful; speedups similar to those of imperative programs with explicit parallelism have been achieved. This is significant, as parallelism in SISAL is implicit.

B. The Connection Machine

Connectionism is a computational model that attempts to model the brain's intelligence as millions (or billions) of neurons (processors) working together and in parallel. The Connection Machine's architecture was originally designed to implement these connectionist networks [66] but has since been used successfully in many other areas, which will be enumerated later.

The Connection Machine-2 (the second-generation Connection Machine) [67] is a SIMD computer with 65 536 (2^{16}) processors connected in a binary 12-cube. Each node of the 12-cube contains 16 bit-serial processors, each with 64K of memory (for a total half gigabyte of memory). The Connection Machine can be viewed as a coprocessor attached to a front-end computer (usually a Vax or a Symbolics), acting as an extended "intelligent" memory.

From this terse description we can readily see that the Connection Machine poses several problems in managing three resources:

- The processors—The processors are viewed as a dynamic resource. When programming the Connection Machine (usually in C or Lisp) processors are allocated as they are needed and are viewed as "virtual" processors.
- The memory—Efficient use of a half gigabyte of memory can be unwieldy. The message-passed structure of communication and the limited[1] number of processors makes each processor's local memory a valuable resource.
- The host computer—It is the job of the host computer to perform all other tasks that have not been designated to the Connection Machine. This will mainly involve I/O but also any computational task that may not map well to the Connection Machine. The host computer can potentially be a supercomputer, making it a valuable resource.

The hypercube network provides a flexible communication network. Many other networks are subsumed by or equivalent to the hypercube by altering processor addressing schemes [69]. On top of the hypercube network the Connection Machine also implements a grid network, which allows nearest neighbor communication. The combination of these two networks and special-purpose routing hardware allows "programmable connections" between the processors. This is the source of its power.

[1]In the context of connectionism, 64K processors is still a small amount. The Connection Machine can process 36×10^{11} bits of data/second, a factor of 20 million short of the estimated power of the brain [68].

1) Applications of the Connection Machine: The Connection Machine implements very fine-grained parallelism. Tasks that map well to it are applications that involve large amounts of data. This includes document retrieval [70], memory-based reasoning (semantic and neural networks) [66], graphics [71], fluid-dynamics [71], [72], and list processing [73].

An obvious question to address is how is the Connection Machine programmed? Versions of Lisp and C have been implemented for the Connection Machine. Suprisingly, one of the reasons for the success of the Connection Machine has been its ease of programmability. This ease is due to the designers' success in making the underlying topological structure of the architecture transparent to the programmer. The machine is simply a dynamic pool of processors to choose from—all seemingly accessible with the same amount of overhead. Hillis [66] refers to them as virtual processors. The connection machine is programmed in a data-parallel style. Data-level parallelism (as opposed to control level) performs the same operation(s) across multiple portions of the data in a manner similar to that of SIMD.

This "architectural transparency," data-parallel programming style, and the "illusion" of an unbounded processor pool and unit-time processor communication makes programming it very similar to programming a PRAM, as was discussed earlier. It is important to note that, in reality, the processor pool is not unbounded and communication overhead is never negligible. This idealized view of programming is very convenient, but convenience does not imply that a good algorithm is being used. It only aids in the design. The reader is referred to [23], [73] for a discussion of "efficient" parallel algorithms.

Example: A simple example is finding the end of a linear linked list [73]. At first glance this problem seems very sequential. That is, in order to get to the end of the list one must traverse every element before it. On the Connection Machine, though, data is mapped across the PEs, so what is really happening is that we have set up a linked list of processors. The idea is simple: On iteration i, each processor begins traversing its successor cell, $i + 2^0$, in parallel. On a successive iteration $i + 1$ each processor begins searching and will now examine the cell at $i + 2^1$ positions down the list (in parallel). On yet another iteration $i + 2$, each processor examines the cell $i + 2^2$, positions down the list—and so on. Several cells are examined more than once, but it is easy to see that each processor will reach the end of the linked list in no more than $\log_2 n$ iterations. Fig. 7 shows the pseudocode that performs this parallel search.

```
for all k in parallel do
    elem[k] = next(elem[k]);
    while elem[k] ≠ null and elem[elem[k]] ≠ null do
        elem[k] = elem[elem[k]];
    end do
end do
```

Fig. 7. A parallel search for the end of a linked list.

V. Conclusions

The various resource allocation techniques described above have a lot in common. This fact should not be overly surprising. In all cases, the goal is to improve performance by maximizing utilization of resources. There are two basic ways to achieve this goal: 1) rearrange the work to keep the processing resources busy, and 2) minimize the time lost to overhead, waiting for data or waiting for synchronization. We need to rearrange the work to keep pipelines full, to create vectorizable computations, to keep multiple processors busy. We need to minimize overhead to reduce pipeline interlocks, memory contention, and message passing.

The techniques used for resource allocation vary considerably in their specifics, but in their abstract ideas they are very much the same.

Programming a supercomputer effectively is a complex and challenging task. Future generations of supercomputers will presumably have more parallel capabilities and be even more complex to program. Fortunately, we can expect that future compilers will be able to optimize parallel decomposition and allocation automatically, just as current compilers can optimize sequential code. Vectorization techniques, from analysis to the many restructuring techniques, are by now well-established and available in commercial compilers.

Currently, the best-known commercial supercomputers are vector and array processors. Hypercube and other MIMD machines are the subject of considerable research and commercial development, and we can expect to see more such machines, with much more software support than is currently available.

One should realize that all of the automatic allocation techniques do not relieve the programmer of the task of finding a good algorithm. The best resource allocator cannot make up for an inefficient algorithm, nor can it transform an algorithm that is a poor match for the underlying machine model. A programmer should understand the problems that face a resource allocator, use this knowledge to choose a target machine that is suitable to his problem, and then structure his code in a suitable form.

References

[1] M. J. Wolfe, "Optimizing supercompilers for supercomputers," Ph.D. dissertation, University of Illinois at Urbana-Champaign, 1982.

[2] S. Kim, D. P. Agrawal, J.-S. Leu, and J. Mauney, "Modeling techniques in a parallelizing compiler for the B-HIVE multiprocessor system," *J. High Speed Comput.*, vol. 1, no. 1, pp. 143–164, May 1989.

[3] D. P. Agrawal, J. Mauney, and L. T. Simpson, "Structure of a parallelizing compiler for the B-HIVE multicomputer," in *Proc. EUROMMICRO 88*, pp. 79–84, Aug. 1988.

[4] D. P. Agrawal, Ed. *Advanced Computer Architecture Tutorial Text.* Los Alamitos, CA: IEEE Computer Society Press, 1986.

[5] M. J. Flynn, "Some computer organizations and their effectiveness," *IEEE Trans. Comput.*, vol. C-21, no. 9, pp. 948–960, Sept. 1972.

[6] A. V. Aho, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools.* Reading, MA: Addison-Wesley, 1986.

[7] J.-S. Leu, "Strategies for retargeting existing sequential programs for parallel processing," Ph.D. dissertation, North Carolina State University, 1987.

[8] J. R. Ellis, *Bulldog: A Compiler for VLIW Architecture.* Cambridge, MA: MIT Press, 1986.

[9] V. Sarkar and J. Hennessy, "Compile-time partitioning and scheduling of parallel programs," in *Proc. ACM/SIGPLAN 1986 Symp. Compiler Construction* (Palo Alto, CA, June 25–27) Association for Computing Machinery, ACM press, 1986, pp. 17–28.

[10] A. H. Veen, "Dataflow machine architectures," *ACM Computing Surveys*, vol. 18, no. 4, pp. 365–396, Dec. 1988.

[11] J. Backus, "Can programming be liberated from the von Neumann style?: A functional style and its algebra of programs," *Comm. ACM*, vol. 21, no. 8, pp. 613–640, Aug. 1977.

[12] J. McGraw, S. Skedzeilewski, S. Allan, R. Oldehoeft, J. Glauert, C. Kirkham, B. Noyce, and R. Thomas. *SISAL Language Reference Manual.* Lawrence Livermore National Laboratory, 1.2 edition, 1985.

[13] J. R. McGraw, "The VAL language: Description and analysis," *ACM Trans. Prog. Lang. Syst.*, vol. 4, no. 1, pp. 44–82, Jan. 1982.

[14] R. Cytron and J. Ferrante, "What's in a name? or The value of renaming for parallelism detection and storage allocation," IBM T. J. Watson Research Center, Technical Report RC 12785, 1987.

[15] H. T. Kung and C. E. Leiserson, *Introduction to VLSI Systems*, chapter "Systolic Arrays." Reading, MA: Addison-Wesley, 1980.

[16] S.-Y. Kung, K. S. Arun, R. Gal-Ezer, and D. V. Bhaskar, "Wavefront array processor: Language, architecture, and applications, *IEEE Trans. Comput.*, vol. C-31, no. 11, pp. 1054–1065, Nov. 1982.

[17] G. Pathak and D. P. Agrawal, "Task division and multicomputer systems," in *Proc. 5th Int. Conf. Distrib. Comput. Syst.*, pp. 273–280, May 1984.

[18] J. K. Peir and D. D. Gajski, "Data flow execution of Fortran loops," in *Proc. 1st Int. Conf. Supercomput. Syst.* (St. Petersburg, FL, Dec. 16–20, 1985), pp. 129–138.

[19] B. Kruatrachue and T. Lewis, "Grain-Size determination for parallel processing," *IEEE Software*, vol. 5, no. 1, pp. 23–33, Jan. 1988.

[20] I. Kaplan, "Programming the Loral of LDF 100 dataflow machine," *ACM Sigplan Notices*, vol. 22, no. 5, pp. 47–56, May 1987.

[21] *IBM Parallel FORTRAN, Language and Library Reference Manual.* IBM Corporation, Mar. 1988.

[22] A. H. Karp and R. G. Babb II, "A comparison of 12 parallel Fortran dialects," *IEEE Software*, vol. 5, no. 5, pp. 52–67, Sept. 1988.

[23] A. Gibbons and W. Rytter, *Efficient Parallel Algorithms.* Cambridge, UK: Cambridge University Press, 1988.

[24] Richard M. Karp and Vijaya Ramachandran, "A survey of parallel algorithms for shared memory machines," To be published in *Handbook of Theoretical Computer Science*, Amsterdam, The Netherlands: North-Holland Press, 1990.

[25] A. Ranade, "How to emulate shared memory," in *Proc. 28th Annu. IEEE Symp. Foundations of Computer Science* (Los Angeles, CA, Oct. 12–14, 1987), pp. 185–194.

[26] L. Bhuyan, Q. Yang, and D. P. Agrawal, "Performance of multiprocessor interconnection networks," *IEEE Comput.*, vol. 22, no. 2, pp. 25–37, Feb. 1989.

[27] D. P. Agrawal, V. K. Janakiram, and G. C. Pathak, "Evaluating performance of multicomputer configurations," *IEEE Comput.*, vol. 17, no. 3, pp. 41–53, Mar. 1985.

[28] E. F. Gehringer, D. P. Siewiorek, and Z. Z. Segall, *Parallel Processing: the Cm* Experience.* Bedford, MA: Digital Press, 1987.

[29] S. Kim, D. P. Agrawal, and R. J. Plemmons, "Least squares multiple updating algorithms on a hypercube," to appear in *J. Parallel Distrib. Comput.*, 1990.

[30] P. P. Budnik and D. J. Kuck, "The organization and use of parallel memories," *IEEE Trans. Comput.*, vol. C-20, no. 12, pp. 1566–1569, Dec. 1971.

[31] R. M. Russell, "The Cray-1 computer system," *Comm. ACM*, vol. 21, no. 1, pp. 63–72, Jan. 1978.

[32] A. Padegs, B. B. Moore, R. M. Smith, and W. Buchholz, "The IBM System/370 vector architecture: Design considerations," *IEEE Trans. Comput.*, vol. 37, no. 5, pp. 509–520, May 1988.

[33] J. R. Allen and K. Kennedy, "A parallel programming environment," *IEEE Software*, vol. 2, no. 4, pp. 21–29, July 1985.

[34] B. Liu and N. Strother, "Programming in VS Fortran on the IBM 3090 for maximum vector performance," *IEEE Comput.*, vol. 21, no. 6, pp. 65–76, June 1988.

[35] D. J. Lilja, "Reducing the branch penalty in pipelined processors," *IEEE Comput.*, vol. 21, no. 7, pp. 47–55, July 1988.

[36] IBM Corp., Designing and writing Fortran programs for vector and parallel processing. *Nov. 1986. File No. S370-25.*

[37] T. R. McKelvey and D. P. Agrawal, "Design of software for distributed/multiprocessor systems," in *Proc. of the Nat. Comput. Conf.* (Houston, TX, AFIPS, June 7–10, 1982), pp. 238–249.

[38] D. J. Kuck, A. H. Sameh, R. Cytron, A. V. Veidenbaum, C. D. Polychronopoulos, G. Lee, T. McDaniel, B. R. Leasure, C. Beckman, J. R. B. Davies, and C. P. Kruskal, "The effects of program restructuring, algorithm change, and architecture choice on program performance," in *Proc. 13th Int. Conf. Parallel Processing* (Bellaire, MI, Aug. 21–24, 1984), pp. 129–138.

[39] C. V. Ramamoorthy, K. M. Chandy, and M. J. Gonzales, Jr. "Optimal scheduling strategies in a multiprocessor system," *IEEE Trans. Comput.*, vol. C-21, vol. 21, no. 2, pp. 137–146, Feb. 1972.

[40] J. A. Fisher, J. R. Ellis, J. C. Ruttenberg, and A. Nicolau, "Parallel processing: A smart compiler and a dumb machine," in *Proc. ACM SIGPLAN 84 Conf. Compiler Construction* (Montreal, June 17–22), pp. 37–47, 1984.

[41] D. van den Bout, "A digital signal processor and programming system for parallel signal processing," Ph.D. thesis, North Carolina State University, 1987.

[42] A. S. Tanenbaum and R. van Renesse, "Distributed operating systems," *ACM Computing Surveys*, vol. 17, no. 4, pp. 419–470, Dec. 1985.

[43] J. K. Ousterhout, "Scheduling techniques for concurrent systems," in *Proc. 3rd Int. Conf. Distrib. Comput. Syst.*, pp. 22–30, IEEE, 1982.

[44] A. R. Pleszkun and G. S. Sohi, "The performance potential of multiple functional unit processors," in *Proc. 15th Annu. Int. Symp. Computer Architecture* (Honolulu, HI, May 30–June 2, 1988), pp. 37–44.

[45] H. Lu and M. J. Carey, "Load-balanced task allocation in locally distributed computer systems," in *Proc. Int. Conf. Parallel Processing* (Penn State U., Aug. 19–22, 1986), pp. 1037–1039.

[46] J. H. Saltz, M. A. Iqbal, and S. H. Bokhari, "A comparative analysis of static and dynamic load balancing strategies," in *Proc. Int. Conf. Parallel Processing* (Penn State U., Aug. 19–22, 1986), pp. 1040–1047.

[47] S. H. Bokhari, "Dual processor scheduling with dynamic reassignment," *IEEE Trans. Software Eng.*, vol. SE-5, no. 4, pp. 341–349, 1979.

[48] C. D. Polychronopoulos and D. J. Kuck, "Guided self-scheduling: A practical scheduling scheme for parallel supercomputers," *IEEE Trans. Comput.*, vol. C-36, no. 12, pp. 1425–1440, Dec. 1987.

[49] Z. Fang, P.-C. Yew, P. Tang, and C.-Q. Zhu, "Dynamic processor self-scheduling for general parallel nested loops," in *1987 Proc. 16th Int. Conf. on Parallel Processing* (St. Charles, IL, Aug. 17–21, 1987), pp. 1–10.

[50] S. H. Bokhari, "Partitioning problems in parallel, pipelined, and distributed computing," *IEEE Trans. Comput.*, vol. 37, no. 1, pp. 48–57, 1988.

[51] E. G. Coffman and R. L. Graham, "Optimal scheduling for two-processor systems," *Acta Informatica*, vol. 1, no. 3, pp. 200–213, 1972.

[52] H. Kasahara and S. Narita, "An approach to supercomputing using multiprocessor scheduling algorithms," in *Proc. 18th Int. Conf. Supercomput. Syst.* (St. Petersburg, FL, Dec. 16–20, 1985), pp. 139–148.

[53] T. L. Adam, K. M. Vhandy, and J. R. Dickson, "A comparison of list schedulers for parallel processing systems," *Comm. ACM*, vol. 17, no. 12, pp. 685–690, 1974.

[54] M. Dubois and F. A. Briggs, "Effects of cache coherency in multiprocessors," *IEEE Trans. Comput.*, vol. C-31, no. 11, pp. 1083–1099, Nov. 1982.

[55] K. Y. Lee, W. Abu-Sufah, and D. J. Kuck, "On modeling performance degradation due to data movement in vector machines," in *Proc. 13th Int. Conf. Parallel Processing* (Bellaire, MI, Aug. 21–24, 1984), pp. 269–277.

[56] M. Burke and R. Cytron, "Interprocedural dependence analysis and parallelization," Computer Science Department, IBM T. J. Watson Res. Center, 1986.

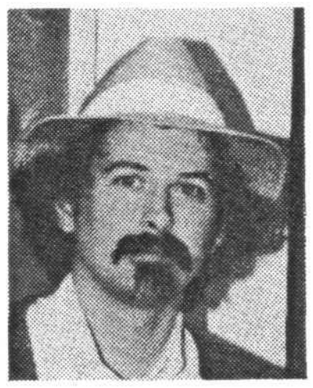
[57] S. P. Midkiff and D. A. Padua, "Compiler generated synchronization for do loops," in *Proc. 15th Int. Conf. Parallel Processing* (Penn State U., Aug. 19–22, 1986), pp. 544–551.

[58] C. D. Polychronopoulos, D. J. Kuck, and D. A. Padua, "Execution of parallel loops on parallel processor systems," in *Proc. 15th Int. Conf. Parallel Processing* (Penn State U., Aug. 19–22, 1986), pp. 519–527.

[59] P. Tang and P.-C. Yew, "Processor self-scheduling for multiple-nested parallel loops," in *Proc. 15th Int. Conf. Parallel Processing* (Penn State U., Aug. 19–22, 1986), pp. 528–535.

[60] S. Skedzielewski and J. Glauert. IF1: An Intermediate Form

for Applicative Languages. Lawrence Livermore National Laboratory, 1.0 edition, 1985.
[61] W. B. Ackerman, "Data flow languages," *IEEE Comput.*, vol. 15, no. 2, pp. 15–24, Feb. 1982.
[62] R. R. Oldehoeft and S. J. Allan, *Parallel MIMD Computation: HEP Supercomputer and Its Applications*, chapter Execution Support for HEP SISAL, MIT Press, 1985, pp. 151–180.
[63] R. R. Oldehoeft, D. C. Cann, and S. J. Allan, "SISAL: initial MIMD performance results," in *ConPar '86: Proc. Conf. Algorithms and Hardware for Parallel Processing*, New York, NY: Springer-Verlag, Sept. 1986, pp. 120–127.
[64] R. R. Oldehoeft and D. C. Cann, "Applicative parallelism on a shared-memory multiprocessor," *IEEE Software*, vol. 5, no. 1, pp. 62–70, Jan. 1988.
[65] D. E. Culler and Arvind, "Resource requirements for dataflow programs,"in *15th Annu. Int. Symp. Computer Architecture*, pp. 141–150, IEEE, 1988.
[66] W. D. Hillis, *The Connection Machine*. Cambridge, MA: MIT Press, 1985.
[67] L. W. Tucker and G. G. Robertson, "Architecture and applications of the connection machine," *IEEE Comput.*, vol. 21, no. 8, 26–38, Aug. 1988.
[68] D. L. Waltz, "The prospects for building truly intelligent machines," DÆDALUS, vol. 117, no. 1, pp. 191–212, Winter 1988. Journal of the American Academy of Arts and Sciences issue on Artificial Intelligence.
[69] S. Ranka, Y. Won, and S. Sahni, "Programming a hypercube multicomputer," *IEEE Software*, vol. 5, no. 5, pp. 69–77, Sept. 1988.
[70] D. L. Waltz, "Applications of the connection machine," *IEEE Comput.*, vol. 20, no. 1, pp. 85–97, Jan. 1987.
[71] W. D. Hillis, "The Connection Machine," *Scientific American*, vol. 256, no. 6, pp. 108–115, June 1987.
[72] S. Brand, "Implications of a truly new machine: a talk with Danny Hillis," *Whole Earth Review*, pp. 108–115, Spring 1987.
[73] W. D. Hillis and G. L. Steele, Jr., "Data parallel algorithms," *Comm. ACM*, vol. 29, no. 1, pp. 1170–1183, Dec. 1986.



Dr. Jon Mauney received the B.S. degree in mathematics from the University of North Carolina, Chapel Hill in 1977 and the Ph.D. in computer science from the University of Wisconsin, Madison in 1983.

He is Assistant Professor of Computer-Science at North Carolina State University. He helped to design and build the POE language-based editor while at Wisconsin. His current research interests include syntactic error-repair and compiling for parallel machines.

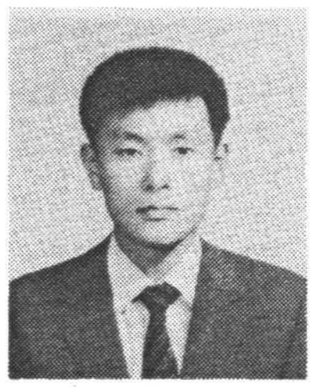
Dr. Mauney is a member of the IEEE Computer Society, ACM, and Sigma Xi.

Dr. Dharma P. Agrawal (Fellow, IEEE) received the B.E. degree in electrical engineering from the Ravishanker University, Raipur, India, in 1966, the M.E. (Hons.) degree in electronics & communication engineering from the University of Roorkee, Roorkee, India, in 1968, and the D.S.c. Tech degree in electrical engineering from the Swiss Federal Institute of Technology, Lausanne, Switzerland, in 1975.

He served the Southern Methodist University, Dallas, TX as a Post-doctoral Fellow and Instructor; the Federal Institute of Technology, Lausanne, Switzerland, as an Assistant; the University of Roorkee as a Lecturer, and M.N.R. Engineering College, Allahabad, India as a Lecturer. He has been a consultant to the General Dynamics Land Systems Division, the Battelle, Inc., and the U.S. Army. He has held visiting appointment at the AIRMICS, Atlanta, and the Tata Institute of Fundamental Research, Bombay.

He joined the faculty of the Wayne State University, Detroit, in 1977 and moved to the North Carolina State University, Raleigh, in 1982, where he is currently a Professor of Electrical and Computer Engineering.

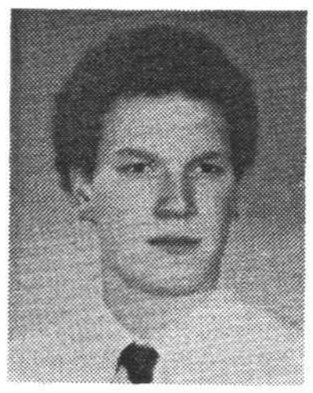
Dr. Agrawal has published a number of papers in the areas of parallel system architecture, interconnection networks, parallelism detection techniques, reliability of real-time distributed sytems, modeling of c-mos circuits, scheduling techniques, and computer arithmetic. He is an author of a tutorial text on *Advanced Computer Archtecture* (1986, IEEE Computer Society Press) and co-author of forthcoming tutorial texts on *Distributed Computing Network Reliability*, and *Advances in Distributed System Reliability*. He is a recipient of the Certificate of Appreciation from the IEEE Computer Society for his services to the IEEE-CS Publications Board. He is a member of the ACM and SIAM. He is an editor of the *IEEE Computer* and founding editor of *Journal of Parallel and Distributed Computing* (Academic Press) and the *International Journal of High-Speed Computing* (World Scientific Publishing Co.) He has been a co-guest editor for the *IEEE Transactions on Computers* and the *IEEE Transactions on Reliability*.



Young K. Choe received the B.S. degree in nuclear engineering from Seoul National University, Korea, and the M.S. degree in computer science from North Carolina State University. He is a doctoral candidate in the Department of Electrical and Computer Engineering at North Carolina State University.

He is a senior systems developer at Tangram Systems Corporation in Cary, NC. His research interests include the validation of distributed software and communication protocols, dynamic and incremental compilation, software engineering tools and standardization, and message mechanisms for distributed systems.

Mr. Choe is a member of the IEEE Computer Society, ACM, and Phi Kappa Phi.



Edwin A. Harcourt received the B.S. in computer science at the State University of New York, Plattsburgh, in 1986 and the M.S. in computer engineering at North Carolina State University in 1989. He is currently working on the Ph.D. in computer science at NCSU.

His research interests are in dataflow languages and their compilers. He is currently examining methods of decreasing compiler run time by performing attribute grammar evaluation in parallel.

Mr. Harcourt is a member of the ACM.

Sukil Kim received the B.S. degree in electrical engineering from Seoul National University, Seoul, Korea, in 1975, the M.E. degree in computer engineering from Yonsei University, Seoul, Korea in 1985, and the Ph.D. degree in electrical and computer engineering from North Carolina State University at Raleigh, NC in 1989.

He is a senior researcher at the Agency for Defence Development (ADD), Korea. His research interests include parallel algorithm design, parallel compiler, and applied parallel computing.

He is a member of the IEEE Computer Society.

Wayne J. Staats received the B.S. degree in mathematics and computer science from Westminster College, New Wilmington, PA, and the M.S. degree in computer engineering from North Carolina State University, Raleigh, NC.

He is currently working toward the Ph.D. degree in the graduate program in computer science at North Carolina State University. His research interests include parallelizing techniques, operating systems, and object oriented systems.