MULTIPROCESSOR SCHEDULING TO ACCOUNT FOR INTERPROCESSOR COMMUNICATION

by

Gilbert Christopher Sih

Memorandum No. UCB/ERL M91/29

22 April 1991

MULTIPROCESSOR SCHEDULING TO ACCOUNT FOR INTERPROCESSOR COMMUNICATION

by

Gilbert Christopher Sih

Memorandum No. UCB/ERL M91/29

22 April 1991

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

MULTIPROCESSOR SCHEDULING TO ACCOUNT FOR INTERPROCESSOR COMMUNICATION

by

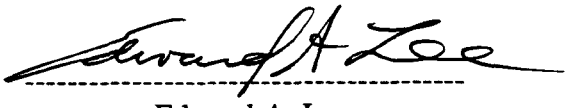
Gilbert Christopher Sih

ABSTRACT

Interprocessor communication (IPC) overheads have emerged as the major performance limitation in parallel processing systems, due to the transmission delays, synchronization overheads, and conflicts for shared communication resources created by data exchange. Accounting for these overheads is essential for attaining efficient hardware utilization. This thesis introduces two new compile-time heuristics for scheduling precedence graphs onto multiprocessor architectures, which account for interprocessor communication overheads and interconnection constraints in the architecture. These algorithms perform scheduling and routing simultaneously to account for irregular interprocessor interconnections, and schedule all communications as well as all computations to eliminate shared resource contention.

The first technique, called **dynamic-level scheduling**, modifies the classical HLFET list scheduling strategy to account for IPC and synchronization overheads. By using dynamically changing priorities to match nodes and processors at each step, this technique attains an equitable tradeoff between load balancing and interprocessor communication cost. This method is fast, flexible, widely targetable, and displays promising performance.

The second technique, called **declustering**, establishes a parallelism hierarchy upon the precedence graph using graph-analysis techniques which explicitly address the tradeoff between exploiting parallelism and incurring communication cost. By systematically decomposing this hierarchy, the declustering process exposes parallelism instances in order of importance, assuring efficient use of the available processing resources. In contrast with traditional clustering schemes, this technique can adjust the level of cluster granularity to suit the characteristics of the specified architecture, leading to a more effective solution.



Edward A. Lee
Thesis Committee Chairman

ACKNOWLEDGEMENTS

Many people contributed in bringing this thesis to fruition. Regarding the members of my thesis committee, I am especially indebted to my research advisor, Professor Edward Lee, for his generous support and encouragement, and for his patience while I struggled to find a thesis topic. I look forward to our continued correspondence as colleagues and friends. I thank Professor David Messerschmitt for providing valuable advice, and Professor Charles Stone, for making helpful suggestions. I also thank Professors Lee and Messerschmitt for their printpaper macros, which greatly eased the task of writing papers as well as this dissertation.

Several graduate students who arrived before me deserve special mention. Vijay Madisetti and Ho-Ping Tseng provided camaraderie as well as much-appreciated advice, while Ilovich Mordechay and Teresa Meng contributed valuable assistance.

Conversations with John Barry, Shuvra Bhattacharyya, Paul Haskell, and Phil Lapsley were enlightening, and interactions with John Baker, Joe Buck, Wen-Lung Chen, Soonhoi Ha, Wai Ho, Limin Hu, Horng-Dar Lin, Tom Parks, Maureen O'Reilly, Ravi Subramanian, Jane Sun, and Valerie Taylor have proven fruitful.

This work was supported by SRC grant 52055 and I am grateful to SRC for their financial backing.

On the personal side, I thank Julie Stoner for her love and for providing much-needed diversion from the rigors of graduate school. Finally, I dedicate this thesis to my parents, for their love, encouragement, and support.

CONTENTS

1

INTRODUCTION TO PARALLEL PROCESSING

Knowledge is the only instrument of production that is not subject to diminishing returns

— J.M. Clark

Computation speeds, which have increased geometrically since 1982, have still not kept pace with the insatiable demand for computing power. Applications such as weather prediction, video processing, medical imaging, and seismic processing require processing speeds exceeding the capabilities of current machines. At the uniprocessor level, computer architects have used features such as superpipelining, parallel instruction issue, and multiple functional units to increase performance. At a higher level, the use of cooperating multiple processors to simultaneously attack a single problem has become increasingly important, with major research and development

efforts underway in both academia and industry. As VLSI density approaches its physical limits, parallel processing offers the most promise for obtaining large computational power in a cost-effective manner.

The primary goal in utilizing multiple processors is to obtain a **speedup**, defined as the smallest number of time steps needed for sequential execution on a single processor $S(n)$, divided by the number of time steps required for parallel execution on multiple processors $P(n)$. The maximum speedup attainable with P processors is P. Existence of such a speedup can be shown using a problem which partitions perfectly into P equal, independent sections. The premise that a speedup greater than P is impossible is shown through contradiction. Given that the *fastest possible* sequential algorithm can solve the problem in $S(n)$ time steps, assume that a set of P processors can obtain a speedup greater than P and therefore solves the problem in time $P(n) < \frac{S(n)}{P}$. Simulation of the parallel execution steps using a single processor results in a sequential algorithm which executes in time $P \times P(n) < S(n)$, which gives the contradiction.

A time-honored argument, referred to as Amdahl's Law, contends that the obtainable speedup limit is constrained asymptotically by the reciprocal of the fraction of computation which must be performed serially [Amd67]. If F_s denotes the fraction of the computation which cannot be parallelized, and R denotes the speed ratio of the fast processing mode (parallel) to the slow processing mode (serial), then the total speedup can be represented as:

$$SPEEDUP = \frac{R}{1 - F_s + RF_s}. \tag{1.1}$$

Asymptotically, as R tends to infinity, the speedup tends toward $\frac{1}{F_s}$. This means that

if a speedup of 10 is desired, less than 10% of the computation can be performed serially; similarly, a speedup of 100 requires less than 1% of the computation to be unparallelizable. A graph illustrating this relationship is shown below in figure 1-1. Notice how quickly the speedup drops off as F_s increases. While critics have used this argument to disparage the idea of parallel processing, in actuality this law is slightly misleading, indicating a pessimistic performance bound. As R tends to infinity, the number of processors must also tend to infinity, and the problem will certainly be scaled upward with the number of processors. The real question then becomes: how does the unparallelizable fraction of the computation F_s change as the problem is scaled? It is easy to construct examples in which F_s tends to zero as the problem is scaled upward, so that the speedup tends to infinity as R tends to infinity.

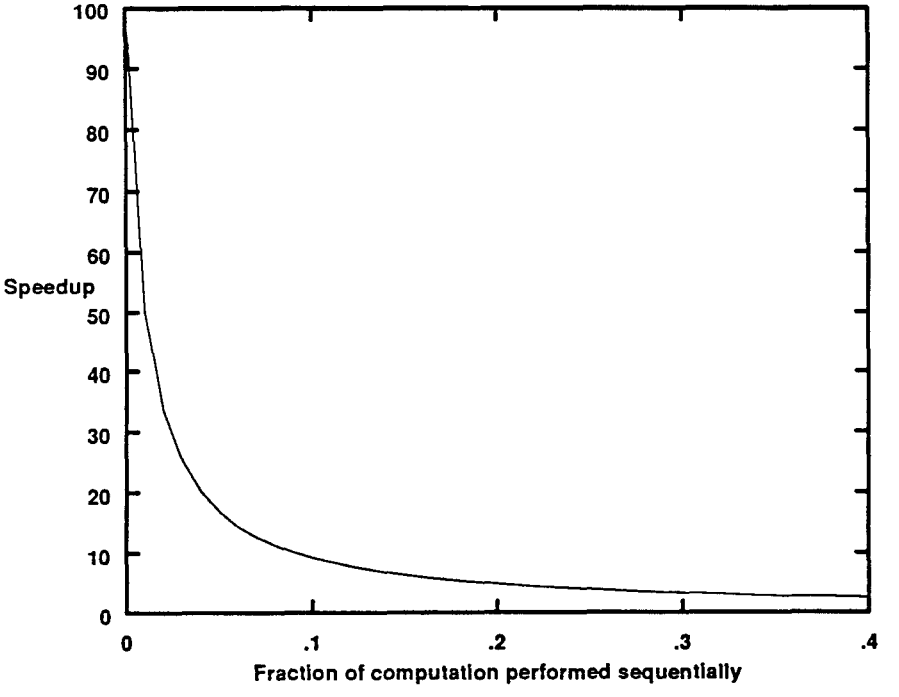


Figure 1-1. Amdahl's law

1.1. HARDWARE

A popular taxonomy for classification of processing systems, due to Flynn [Fly72], separates machines according to the number of instruction and data streams present.

1.1.1. Single-Instruction, Single-Data Machines

Single-Instruction, Single-Data (SISD) machines have a single stream of instructions from a control unit, which regulates the single stream of data between CPU and main memory. Virtually every uniprocessor-based machine belongs to this category. The traditional basis for such machines, the von Neumann computing model, traces most of its original precepts to an Institute for Advanced Study report written back in 1946 [Bur46]. This stored-program model, shown in figure 1-2, consists of a CPU connected to main memory through a narrow pipe, which also connects to an I/O port. If one views the function of a computer program as attempting to change the contents of memory via word-at-a-time transfers of addresses and data through this narrow pipe, the reason this connection between CPU and memory has been nicknamed the "von Neumann bottleneck" should be clear.

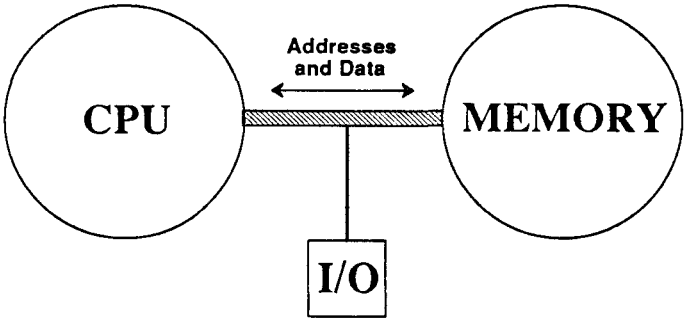


Figure 1-2. The von Neumann computing model

1.1.2. Single-Instruction, Multiple-Data Machines

Single-Instruction, Multiple-Data (SIMD) machines have a single stream of instructions controlling an array of multiple execution units. This design avoids a separate instruction fetch for each data value and permits multiple data units to be processed in parallel. By controlling all processors in lockstep with a single instruction stream, SIMD machines incur very little synchronization overhead, which encourages exploitation of fine-grained parallelism. This category includes the Connection Machine and MasPar MP-1 as well as most array and vector processors such as the Illiac IV [Bou72] and the Goodyear Massively Parallel Processor [Bat80].

1.1.3. Multiple-Instruction, Multiple-Data Machines

Multiple-Instruction, Multiple-Data (MIMD) machines can be classified as being **control-driven**, in which explicit control flow(s) cause the execution of instructions, **data-driven**, in which the availability of operands triggers instruction execution, or **demand-driven**, in which the need for a result provokes instruction execution [Alm89]. In this thesis, we restrict our attention to control-driven MIMD machines, in which each processor contains an autonomous control unit regulating its own data stream. These systems are more flexible than SIMD machines because processors can operate asynchronously. However, each processor requires memory to store its own program, and more complex control and communication strategies are necessary. As a result of the more extensive communication and synchronization overheads, parallelism is generally exploited at a coarser level. MIMD machines can be further classified as being multiprocessors, which communicate through shared memory, or multicomputers, which communicate through message-passing. The following sections discuss several popular multiprocessor and multicomputer topologies.

Multiprocessors

In a multiprocessor, individual processors access shared memory using a single global address space, where the memory is usually physically distributed among several banks to allow several processors to access variables simultaneously. As shown in figure 1-3, the processors (P_i, i = 1 ... P), which may have local memory (LM), are connected to these memory modules (M_j, j = 1 ... N) through some form of interconnection network. This processor-memory interconnection network is a major cost component in multiprocessor system design. A multitude of different interconnection schemes are commonly used, offering a wide range of cost/performance choices.

Shared Bus Multiprocessors

Single shared-bus multiprocessors consist of a small number (typically <20) of processors connected to a global shared memory through a high-speed bus. Regions of physical memory are mapped into the virtual address space of each processor, which

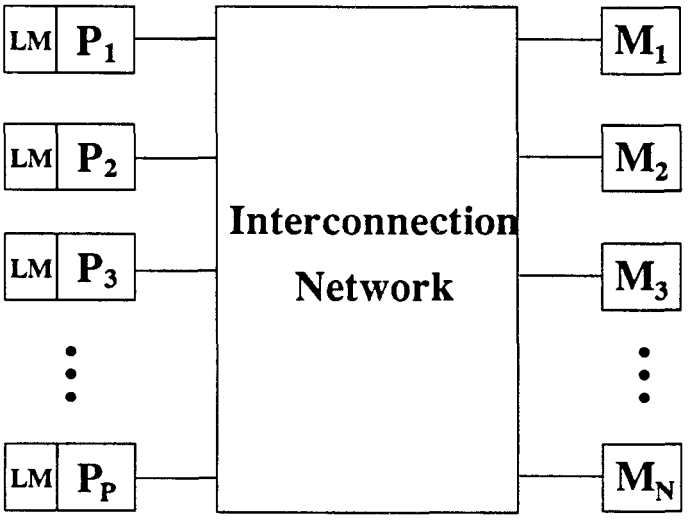


Figure 1-3. The general structure of a switching network multiprocessor

allows interprocessor communication through the use of shared variables. To reduce the amount of bus traffic and lessen the average memory access time, each processor generally has a small local memory, or cache, associated with it as shown in figure 1-4. While such designs are simple and inexpensive, system performance is sorely limited by the shared bus, which only allows a single processor to access the shared memory at any time. Bus arbitration is commonly handled using a hardware arbiter.

To eliminate the synchronization and contention resolution overheads normally associated with a shared bus, Bier et. al. propose an ordered memory architecture which is applicable to a limited class of applications [Bir90]. This architecture uses a central controller to grant the bus to processors in a pre-specified order.

To allow multiple data transfers in parallel, multiple bus architectures have been proposed [Mud87], in which processors are connected to a group of memory banks through several buses, as shown in figure 1-5. The memory banks are typically interleaved for higher memory bandwidth, and multiple arbiters are used to handle bus

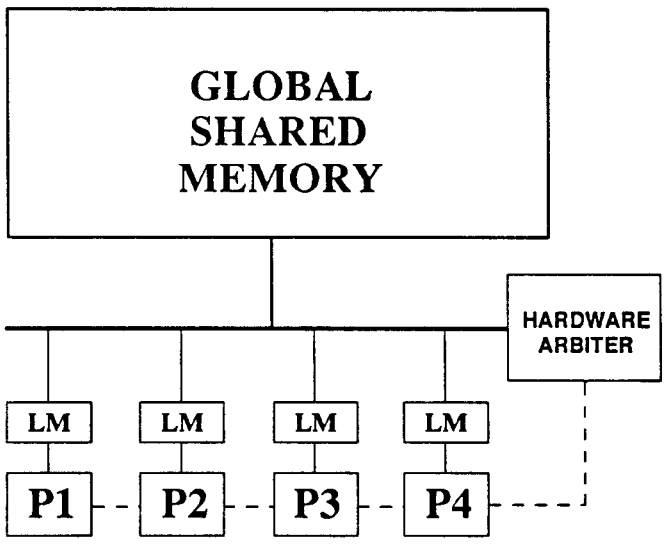


Figure 1-4. A single shared-bus multiprocessor

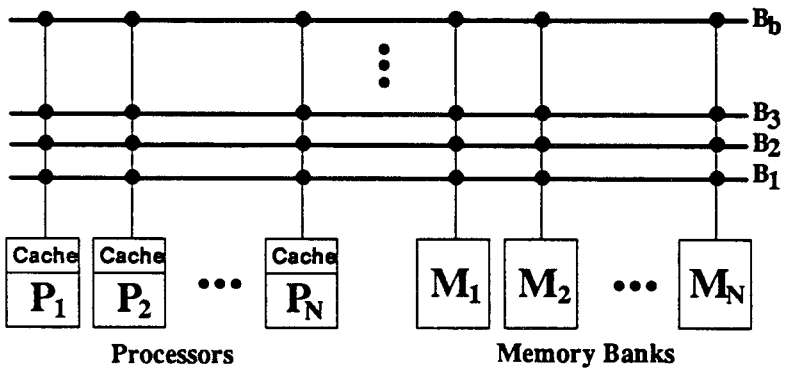


Figure 1-5. A multiple-bus architecture

arbitration. The cache structure and updating policy are critical design issues in multiple-bus systems, because of the desire to reduce bus traffic while maintaining memory consistency. Snooping caches, which monitor all bus traffic to check for cross-hits, are often used to ensure cache consistency. The Wisconsin Multicube, shown in figure 1-6, connects processors through a multidimensional grid of buses to

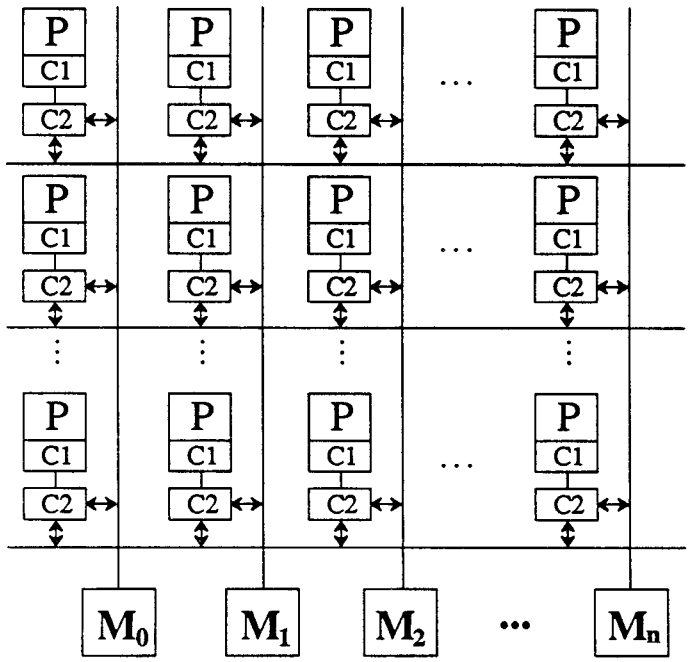


Figure 1-6. The Wisconsin multicube architecture

achieve high interprocessor communication bandwidth [Goo88]. This architecture uses a two-level cache organization, where the first level is a conventional SRAM cache designed to reduce memory latency, and the second level is a large DRAM snooping cache to minimize bus traffic. A write-through cache policy is used from first to second level, and the second level cache uses a write-back policy to reduce bus contention problems. Hierarchical bus organizations have been proposed as another possibility for bus-based multiprocessor systems [Win88].

Crossbar Switch Multiprocessors

A crossbar switch multiprocessor interconnects processors and memories through a crossbar switch, as shown in figure 1-7. By closing exactly one switch in every row and column, this configuration can support simultaneous communication between N processors and N memories, as long as no two processors access the same memory

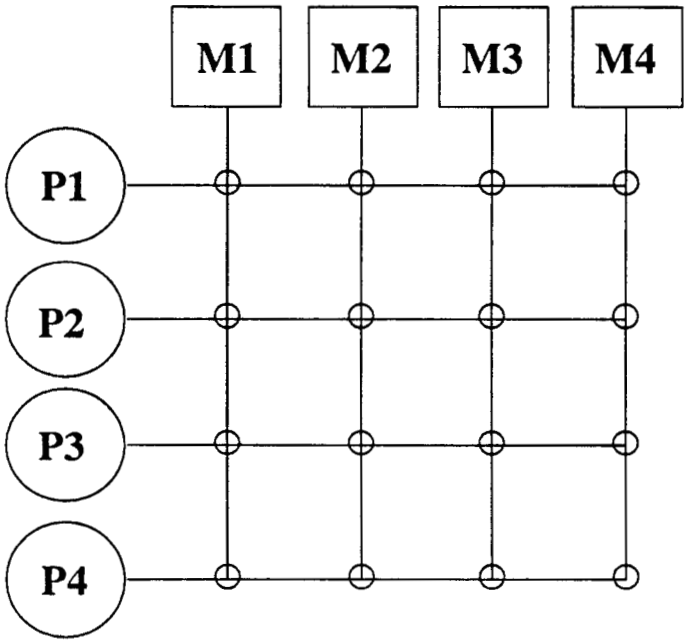


Figure 1-7. A 4x4 crossbar switch interconnection network

bank simultaneously. While this scheme yields a high processor/memory bandwidth, it incurs scaling problems due to a switching cost which increases as N^2. An example of this approach is the Carnegie Mellon multi-mini-processor (C.mmp) [Wul72].

Single-Stage Multiprocessors

Single-stage multiprocessors consist of a link interconnection pattern connected to a stage of switching elements. To route messages between processors, data is often recirculated several times through this single-stage loop until the correct destination processor is encountered. The most well-known single-stage interconnection pattern, often used as a building block in more complicated networks, is called the **perfect shuffle** [Sto71]. This network pattern, illustrated in figure 1-8, derives its name from the similarity of its structure to the shuffling of a deck of cards when the column of

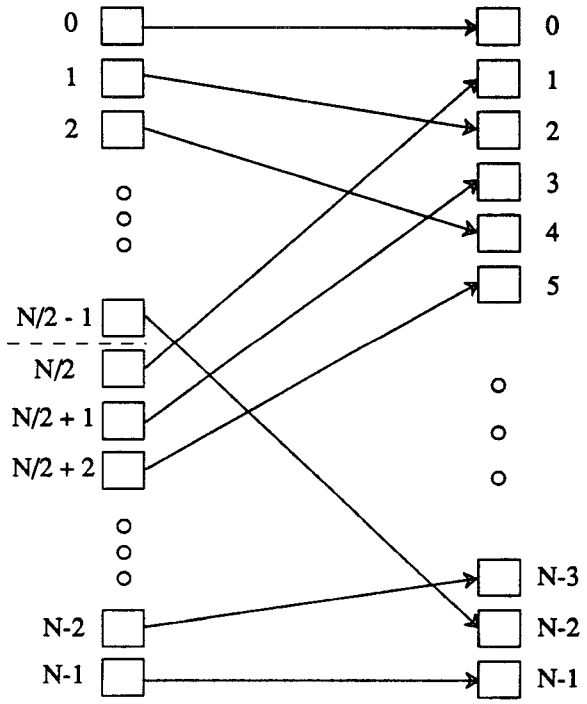


Figure 1-8. The perfect shuffle interconnection

inputs is pictured as the set of cards. If this deck is "cut" in the center, illustrated by the dotted line in the figure, and shuffled perfectly, the stack of outputs then precisely corresponds to the position of the shuffled inputs. An alternate view can be obtained by assigning each processor (numbered top-down from 0 to N-1) its binary bit representation. For each source processor, the bit representation of its destination processor can be found from a cyclic rotation of its bit representation one position to the left. This implies that a message will return to its originating processors after exactly logN shuffles. The NYU Ultracomputer is an multiprocessor which uses the perfect shuffle interconnection augmented with nearest-neighbor connections [Sch80]. Appending a stage of switches to the perfect shuffle, as shown in figure 1-9, results in the well-known shuffle-exchange network [Che81]. This interconnection scheme is especially suited for solving problems which lend themselves to a recursive "divide and conquer" approach, such as sorting [Bat68], FFT's, and matrix multiplication.

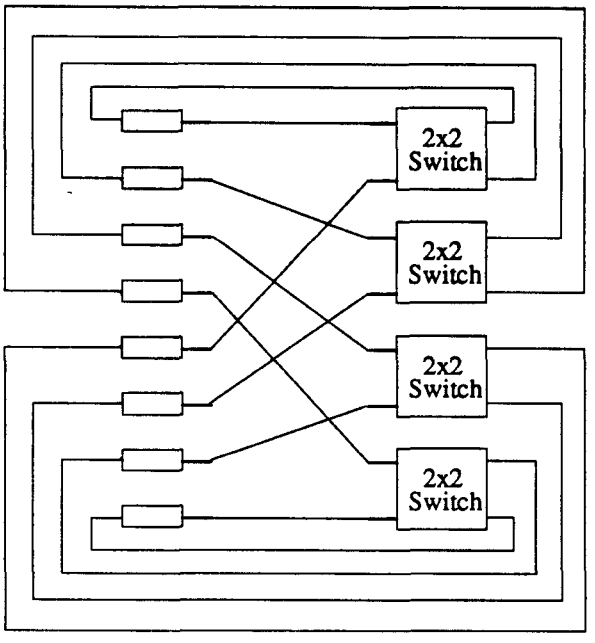


Figure 1-9. A shuffle-exchange network

Multistage Interconnection Network Multiprocessors

An instance of the final class of multiprocessors generally consists of hundreds or thousands of processors connected to memory banks through a multistage interconnection network (MIN) [Fen81]. In its most general form, the network is a switching fabric which dynamically routes messages between N inputs and M outputs, and is constructed by appending multiple stages of $a \times b$ crossbar switches. The most common form of MIN connects N inputs to N outputs using several stages of 2x2 or 4x4 crossbar switches. The 4 possible switching configurations for a 2x2 switch are shown below in figure 1-10. A control bit, normally the ith bit of a $\log_2 N$ bit routing tag, can be used to steer an input at the ith stage to the proper output, with a zero-bit routing the connection through the upper output, and a one-bit routing the connection through the lower output. Notice that the two nonconflicting states, shown in a) and b), have complementary control bits on their inputs, while the two conflicting states,

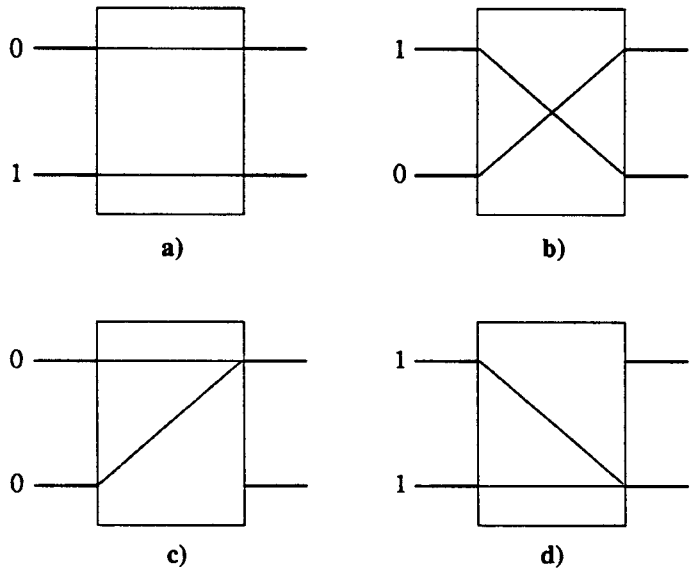


Figure 1-10. The possible configurations of a 2x2 switch

shown in c) and d), have the same control bits on their inputs. To eliminate the conflicting states, one control bit is usually chosen to control the switch, with the other required to be its complement.

The wide variety of interconnection networks available offer a range of performance-cost tradeoffs between a shared bus and full crossbar interconnect [Kru83] [Bhu89]. The two most commonly-used performance measurements are latency, which reflects the delay in transferring a message from source to destination, and bandwidth, which indicates the number of messages exchanged per unit time. MIN's are commonly classified according to the following criteria:

1) Control Strategy
2) Switching Methodology
3) Blocking Characteristics

The control strategy can be **centralized**, in which a central controller physically separate from the data routing hardware handles the switching elements, or **distributed**, in which the switching is handled by the individual elements themselves.

Switching methodologies can be classified as being **circuit switched**, in which a dedicated connection between source and destination ports is maintained for the duration of the data transfer, or **packet switched**, in which fixed-length packets of information are addressed to their destinations and may be stored at intermediate points in the network. To draw an analogy, circuit switching corresponds to the connection operation used when setting up a telephone call, while packet switching resembles the letter-forwarding operation used by the postal service. In general, circuit switching is more efficient for long message transmissions, while packet switching is more effective for short request/reply type messages.

Blocking characteristics refer to a network's capability of supporting dynamically changing permutations of input/output connections, with many of the results dating back to research performed for the telephone switching network [Clo53] [Ben62]. Networks which can support an arbitrary permutation of inputs connected to outputs at all times are referred to as being **nonblocking,** with the crossbar and Clos networks being the most notable examples. Networks which can support all possible permutations by rearranging its existing connections to support a new, unassigned input/output pair are designated as being **rearrangeable nonblocking,** the Benes network [Ben65] being a prime example. Networks which support only a subset of input/output permutations are referred to as **blocking** networks. Blocking networks are more commonly used than nonblocking networks in multiprocessor systems due to their lower hardware switching costs. Packet-switched blocking networks can be further subclassified according to how collisions are handled. If two messages require the same output port, a **buffered** network transmits one message and uses a queue to store the other, while an **unbuffered** network transmits one message and discards the other, forcing a retransmission from the source. Packets may contain a field in the header to aid in determining priorities for transmission.

Banyan Network

One of the most widely-used blocking networks is the *banyan* network, named for its structural similarities to an East Indian fig tree [Gok73]. This interconnection network, shown in figure 1-11, provides a unique path from any input to any output. The network structure, which consists of $\log_2 N$ stages of N/2 2x2 crossbar switches each, is recursively synthesized from smaller banyan networks. Constructing an N x N banyan network (N inputs, N outputs), requires two N/2 x N/2 banyans placed one

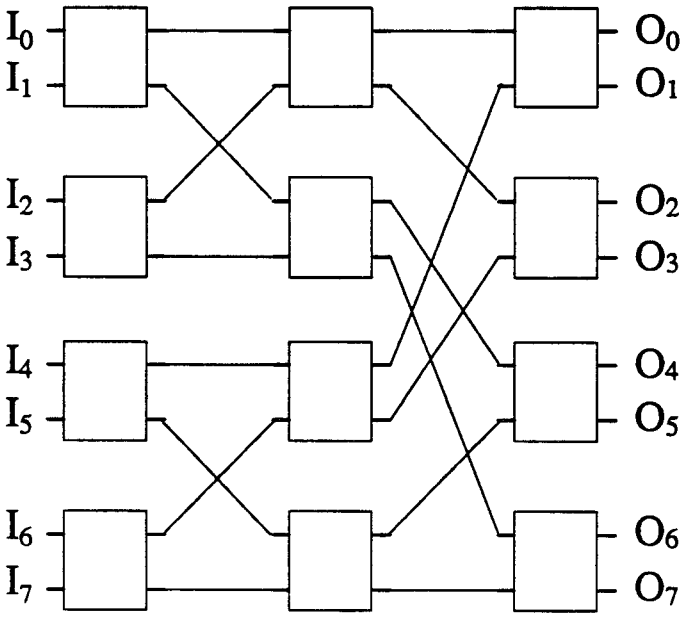


Figure 1-11. An 8x8 banyan network

above the other, with an additional stage of N switches appended on the right. The perfect shuffle interconnection is then used to interconnect the additional stage to the two smaller banyans. The base network, connecting 4 inputs to 4 outputs, is also derived using the perfect shuffle. The appeal of this structure stems from the switching cost, which grows as $N\log_2 N$ as opposed to the crossbar's N^2 cost growth. The delay through the network increases as $\log N$.

Omega Network

Another widely-used interconnection is the *omega* network, which was originally proposed as an alignment network for array processors [Law75]. This interconnection also consists of $\log_2 N$ stages of N/2 2x2 crossbar switches, but uses the perfect shuffle interconnection in each stage. An 8 x 8 omega network is shown below in figure 1-12. The omega network, as well as the banyan network mentioned earlier, have the advantageous property of being self-routing, so that a central controller is unnecessary. If

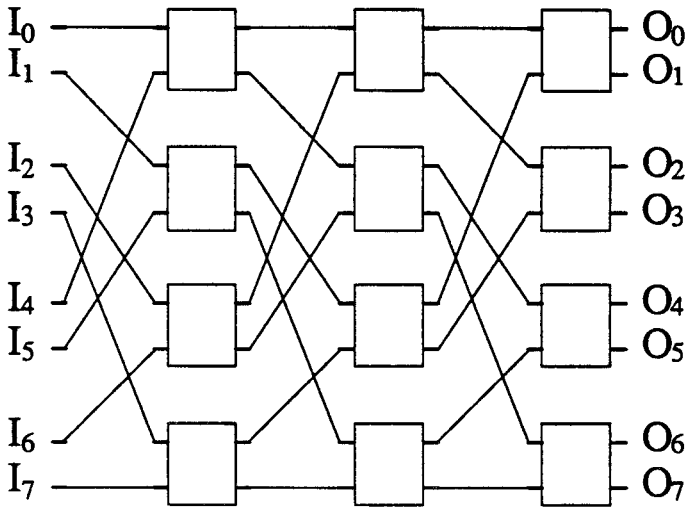


Figure 1-12. An 8x8 omega network

the outputs are numbered top-down from 0 to N-1, routing a message to any output port can be easily accomplished by using the binary representation of the output number. If $b_1 b_2 \cdots b_{\log N}$ represents the binary expansion of the desired output port, then at switching stage i, the message is routed out the upper switch output if bit $b_i = 0$ and out the lower switch output if bit $b_i = 1$. The IBM Research Parallel Processor (RP3) and BBN Butterfly are examples of commercial multiprocessors interconnected using an omega network.

A *delta network* connects A^n inputs to B^n outputs through n stages of AxB crossbar modules, where each stage is connected through an A-shuffle [Pat81]. An A-shuffle of C playing cards is constructed by dividing the deck into A equal piles of C/A cards each and picking cards in a circular fashion (one card from the first pile, one card from the second pile...) until all the cards have been picked up. The perfect shuffle is a two-shuffle under this formalism. The self-routing property of the delta network is obtained by extending the routing algorithm mentioned earlier to base B.

While these multistage interconnections provide an advantageous cost/performance ratio and support simple self routing schemes, the failure of a single link or switch can lead to blockage of several input/output paths. Not surprisingly, an increasing amount of attention is being devoted to techniques for increasing fault-tolerance [Adam87] [Kum87], including adding extra stages, adding extra links, increasing the number of network ports, increasing the switch size, and using adaptive routing methods.

Multicomputers

A **multicomputer** consists of a collection of processing elements (PE's) interconnected in a fixed topology. Each PE contains a processor, local memory, and communication hardware. In contrast with multiprocessors, each processor has a private address space, used to access its own local memory. The cooperating tasks of a parallel algorithm execute asynchronously on different processors, and communicate through message-passing. Hardware costs and packaging constraints limit each PE's connections to a small subset of the total processing network, so that messages must typically be routed through many intermediate PE's before reaching their final destination. Typical multicomputer topologies, shown below in figure 1-13, include the star, ring, hypercube, tree, and mesh [Ree87]. Early multicomputer projects include the Columbia Homogeneous Parallel Processor (CHoPP), the NASA Finite Element Machine (FEM), and the Berkeley X-Tree. The Intel iPSC2 is a commercial hypercube multicomputer [Arl88], where each node consists of a 16 Mhz 80386 processor, 16 Mbytes of dynamic RAM, and the Direct-Connect routing hardware [Nug88]. The Inmos transputer and torus routing chip [Dal86] were designed as building blocks for multicomputers, so that larger multicomputers can be constructed by merely connecting more of these elements together. This scalability is often touted as one of the

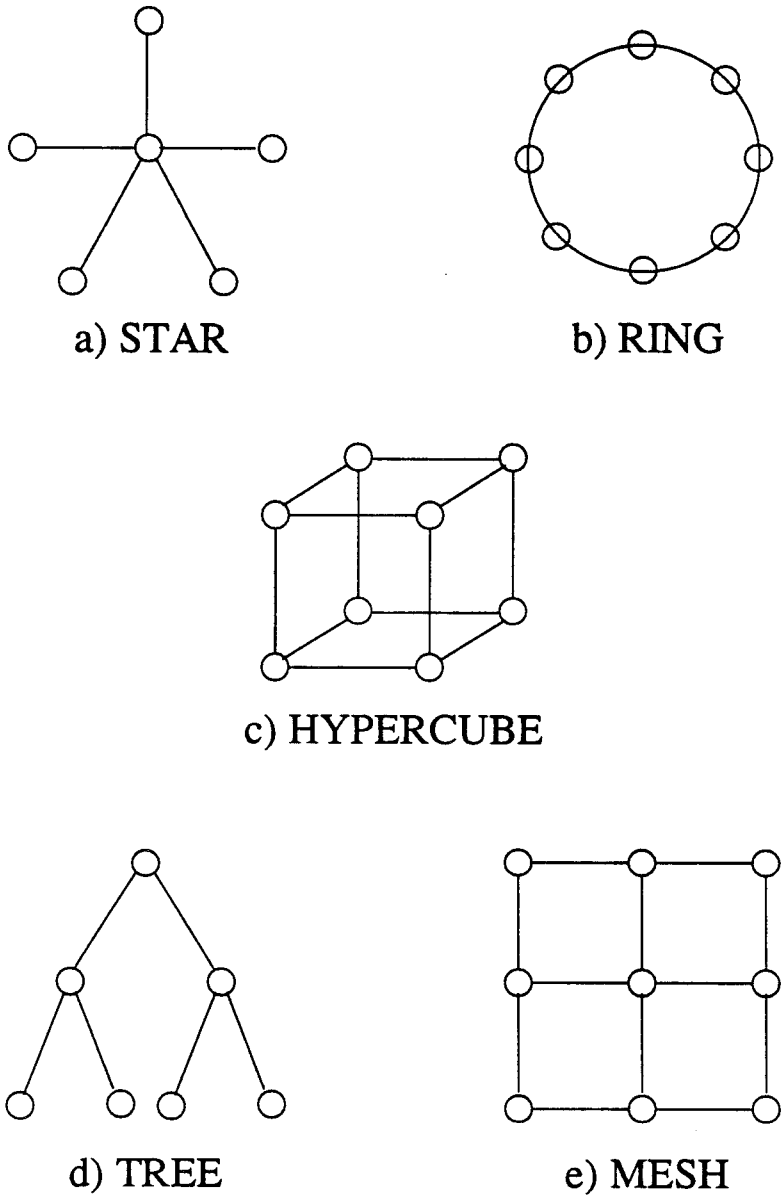


Figure 1-13. Typical multicomputer topologies

inherent advantages of multicomputers over multiprocessors. A disadvantage is that the latency in transferring messages across multicomputer nodes is typically higher than that for multiprocessors, which requires that parallelism be exploited at a larger grain level. For example, the Intel iPSC2, which uses circuit-switching to transfer messages, requires 350 us to set up a connection between processors, and transfers

data at a rate of 2.8 Mbytes/s [Arl88].

Because messages must typically be routed through intermediate nodes before reaching their final destination, the chosen routing strategy plays a major role in determining the communication network performance. In a classical store-and-forward network, messages are divided into packets. Each packet is stored at an intermediate node, only being advanced to the next node along the path after the entire packet has been completely received. To increase transmission efficiency, many networks use **wormhole** [Dal87] rather than store-and-forward routing. This technique subdivides packets into flits, where a flit is the smallest unit of data that a channel can transmit or receive. Instead of waiting until a packet has completely arrived, the wormhole routing approach allows a PE to advance each flit to the next PE along the path as soon it is received. The flits comprising a message become spread throughout the path between source and destination, so that it is possible for the first flit of a message to reach the destination before the last flit has left the source. This technique severely reduces the latency in transmitting messages, as shown in figure 1-14. If L is the message length, W is the channel width, D is the number of interprocessor hops between source and destination, and T_c is the channel cycle time, then $T_c[D \times \frac{L}{W}]$ is the latency for store-and-forward routing as shown in the top figure, while $T_c[D + \frac{L}{W}]$ is the latency for wormhole routing, as illustrated in the bottom figure. By immediately forwarding data flits, the number of channel cycles is converted into an additive rather than a multiplicative term in the latency expression. A similar technique, called **virtual cut-through** [Ker79], buffers messages at intermediate nodes when the header flit is blocked to prevent blocked messages from clogging the network.

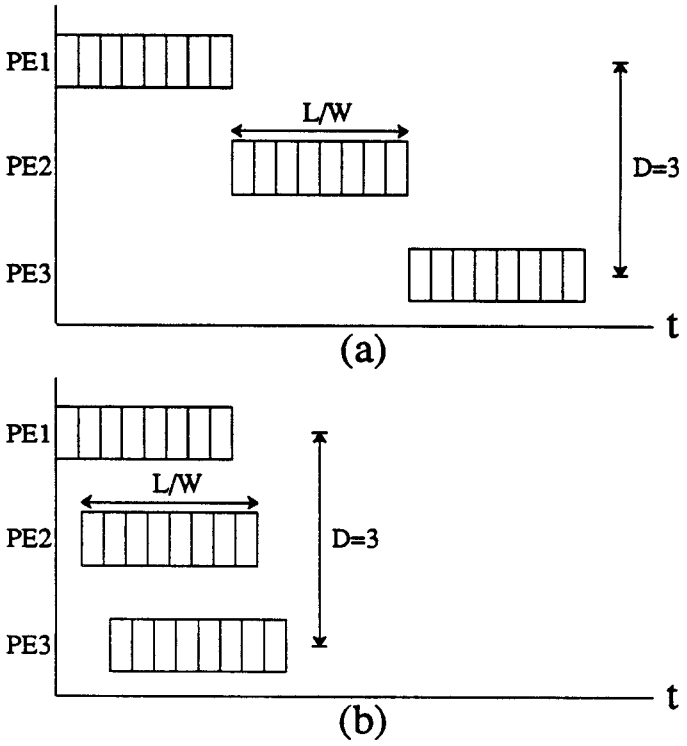


Figure 1-14. Latency of store-and-forward routing vs wormhole routing

1.2. SOFTWARE

While significant advances have been made in the hardware aspects of parallel computation, the software aspects have progressed at a slower rate, hindering progress in the field as a whole. Perhaps the most pressing software requirement is the need for a portable, general-purpose parallel-programming paradigm which can effectively express the parallelism in an application, yet spares the user from having to explicitly coordinate parallel execution.

1.2.1. The PRAM

The most widely used programming model for parallel computation is the Parallel Random Access Machine (PRAM), which is an idealized shared-memory computation

model [Kar88]. The PRAM consists of a set of processors, each with local memory, which communicate through a global shared memory. In each time step, every processor can perform a single computation, or read or write any cell of the global shared memory or its own local memory. PRAMs can be separated into different classes according to the allowed possibilities for simultaneous read and write access from different processors. The most restrictive model is the Exclusive Read Exclusive Write (EREW) PRAM, which prohibits concurrent access to the same memory location from different processors for both reading and writing. The Concurrent Read Exclusive Write (CREW) PRAM allows different processors to simultaneously read the same memory location but forces writes to be done serially. The Concurrent Read Concurrent Write (CRCW) model allows both concurrent read and concurrent write, with various possibilities existing for resolution of simultaneous write conflicts. Interestingly, it has been shown that the power of these models differ at most by a factor of O(log P), where P is the number of processors. Although not physically realizable, this model is useful for proving complexity results and for gaining insight into the structure of parallel computation. The drawback to this model is that it requires the programmer to find and explicitly direct parallel execution in the algorithm. While this policy is manageable for simple sorting and searching routines, it quickly becomes intractable for larger problems. Another problem is that the model is tied to the shared-memory machine, so it is not suitable for general parallel programming.

1.2.2. Parallelizing Compilers

An approach which negates these difficulties uses conventional languages coupled with parallelizing compilers. Users are free to program in C, Fortran, Pascal, Lisp, or any other sequential language. The burden for parallelism detection and partitioning

falls solely on the compiler, which must generate the proper code for each processor in the target architecture. An example of this approach is Parafrase, a restructuring compiler for conventional FORTRAN programs constructed at the University of Illinois [Kuc81]. The first phase of Parafrase converts the high-level language format into a dependence graph representation suitable for performing optimization and restructuring steps. Scalar renaming, subscript expansion, node splitting, forward substitution, and loop distribution all help to expose as much parallelism as possible in this machine-independent stage. The second phase maps this description into the specific architecture. The Bulldog compiler, developed as part of the ELI (Enormously Long Instruction) project at Yale, is another example of this approach. However, the use of a conventional sequential language hinders the parallelizability of parts of the program, incurring the substantial speedup penalty reflected by Amdahl's law.

1.2.3. Parallel Languages

The primary argument against parallelizing compilers is that they are constrained by the semantics of the program; that is, the compiler is not allowed to rewrite the algorithm being employed. However, for many applications, the existing sequential algorithm is not amenable to efficient parallelization. Rethinking an algorithm from first principles can often lead to a more efficient realization which exploits the available concurrency more effectively. The overriding goal then, must be to find a language whose attributes implicitly capture the inherent parallelism, yet frees the programmer from explicit handling of the multiple processor coordination. There have been many attempts to date, including Multilisp, Concurrent Prolog, CSP, Occam, Ada, and Linda. Many of these are extensions to current languages, using constructs such as parbegin/parend, fork/join, or doall to start and stop parallel activities.

Backus claims that the structure of conventional languages is based on the programming style of the von Neumann computer [Bac78]. He likens program variables to memory cells, and relates control statements to test and jump operations. By comparing program assignment statements to fetch, store and arithmetic functions, Backus designates the assignment statement as the von Neumann bottleneck of programming languages. He notes that each assignment statement produces a one word result and claims that conventional programming (e.g. For loops, do loops) is primarily concerned with how words flow through the assignment bottleneck. He advocates a functional programming approach in which programs are merely functions without arguments.

Functional Languages

Languages which operate entirely through the application of function to values are called **functional**, **applicative**, or **dataflow** languages. This programming style, based on the mathematical lambda calculus [Chr41][Ros82], is data-driven, so that data dependencies control the sequencing of instructions, rather than a program counter.

Conventional imperative languages accomplish much of their work through the use of side effects, the most common example occurring when a procedure modifies a variable in the calling program. Since variables are commonly tied to specific memory locations in imperative languages, any change in the sequencing of instructions causes a change in the order of assigned values to a memory cell, which may modify the final outcome. The ability to parallelize conventional languages is therefore limited by the presence of side effects.

Side effects are prohibited in functional languages. Functions copy their arguments ("call by value" instead of "call by reference") to avoid changing their values in the calling program. By avoiding side effects, functional language reap the benefits of the Church-Rosser property, which states that any sequence of actions satisfying the data dependencies will produce the same final result. Examples of dataflow languages include pure Lisp, the Value Algorithmic Language (VAL) developed at MIT, and the Irvine Dataflow language (ID) developed at UC-Irvine and MIT. A shortcoming of these languages is the memory inefficiency introduced by the isolation of inputs and outputs which is necessary to avoid side effects. If a function wishes to access an array, the entire array must be copied and passed as a value.

Synchronous Data Flow

Synchronous Data Flow (SDF) [Lee87], a special case of data flow, is a programming methodology in which algorithms are described as directed graphs, where the nodes represent computations with known execution times, the arcs represent data paths, and the number of data samples produced and consumed by every node on each invocation is known statically (at compile time). In addition to being a natural description for digital signal processing (DSP) algorithms, this data-driven description exposes the inherent concurrency which can be exploited by parallel hardware. An example SDF graph is shown below in figure 1-15. The numbers at the tail and head of each arc indicate the number of data units produced and consumed by the respective nodes. For example, node A consumes three data units and produces one data unit on each invocation. The notation 2D on the arc between nodes A and B indicates the presence of 2 logical delays, where a logical delay (equivalent to z^{-1} in signal processing) represents an initial data unit in the first-in, first-out (FIFO) buffer between the nodes.

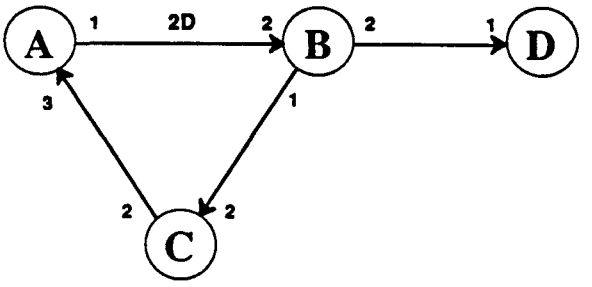


Figure 1-15. A synchronous data flow graph

In this case, because of the feedback loop involving nodes A, B, and C, these initial data samples are necessary to avoid a deadlock condition, where each node waits for data from its predecessor. Using SDF graphs, deadlock avoidance and bounded memory requirements can be guaranteed at compile-time.

1.3. PARALLEL PROCESSING OVERHEADS

In addition to the software difficulties encountered in programming parallel computations, there are many other factors which limit the attainable speedup in a parallel processing environment.

1.3.1. Interprocessor Communication

The most significant performance detriment to parallel processing systems are the overheads created when processors exchange data. Excessive interprocessor communication (IPC) can cause a "saturation effect," in which the addition of processors actually causes a decrease in throughput [Chu80]. There are two main aspects to the IPC overhead, the **communication delay**, and the **resource contention**. Communication delay refers to the time required to transfer data between the source and destination processors. Resource contention is caused by the need for processors to share

communication resources, such as busses, interconnection links, or memory modules. Excessive contention for a particular memory module in switching network multiprocessors has been found to cause so-called "hot spots" in the network, which are analogous to traffic jams. Such hot spots not only degrade traffic in the vicinity of the hot spot, but can degrade all network traffic due to a phenomenon called **tree saturation**, in which the tree of switches rooted at the hot spot and extending to all the processors at the leaves have all their queues filled to capacity. Message combining, in which messages headed to a common destination are combined [Pfi85] has been used in the NYU Ultracomputer and IBM RP3 to reduce hot spot traffic. Other approaches include using a software approach to spread accesses over memory modules [Yew87], or using feedback to limit memory accesses headed toward hot spots [Sco89].

To illustrate the detrimental effects of IPC on speedup, consider the example graph shown in figure 1-16, which will be scheduled on a three processor single shared-bus multiprocessor. When IPC costs are neglected, the optimal schedule which gives

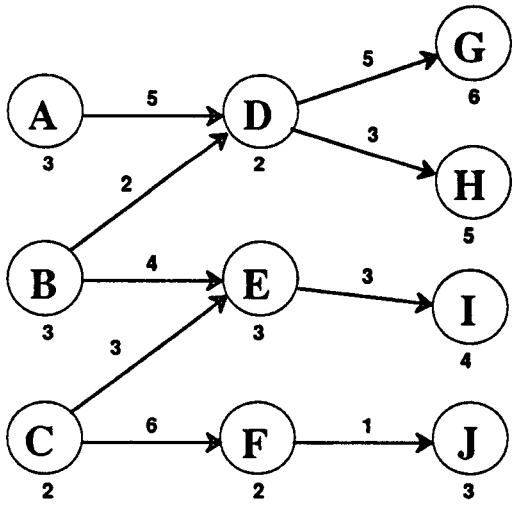


Figure 1-16. An example graph

speedup 3 is shown below in figure 1-17. Assuming for simplicity that the number of data units required to transfer D data units is just D time units, the schedule including communication delay is shown in figure 1-18, which has makespan 15, yielding a speedup of 2.2. However, this schedule allows multiple communications to occur simultaneously, whereas a physical shared-bus multiprocessor only allows one processor to access the bus at any time. The schedule shown in figure 1-19 results when we include the effect of bus contention. This schedule, which has makespan 19, yields a

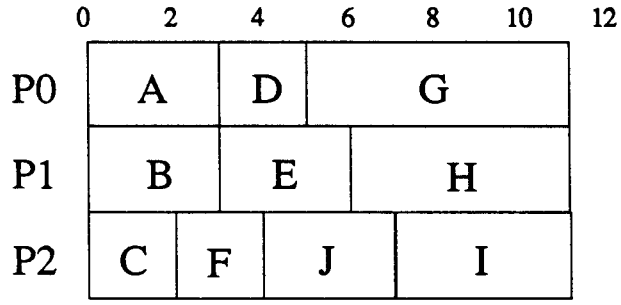


Figure 1-17. Schedule with no IPC cost

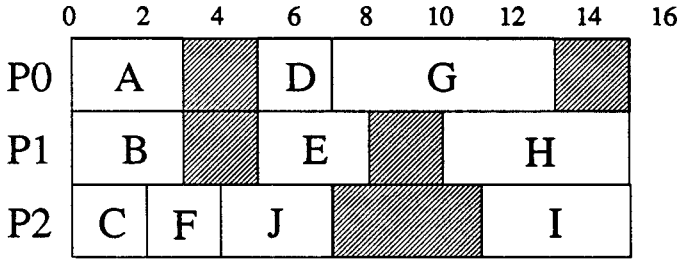


Figure 1-18. Schedule including communication delay

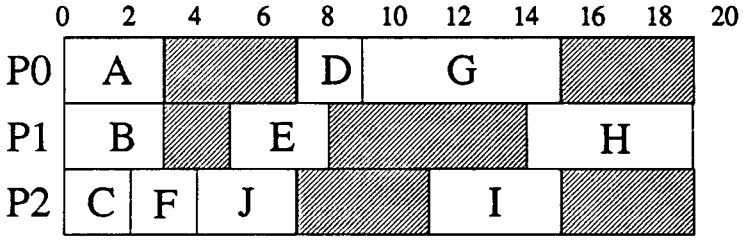


Figure 1-19. Schedule including communication delay and resource contention

speedup of only 1.73. Clearly, the effects of interprocessor communication must be accounted for when scheduling for parallel processors.

1.3.2. Synchronization

Various forms of synchronization are necessary to allow cooperation between processors, creating additional overheads in parallel processing systems. Shared memory multiprocessors require two basic types of synchronization. The first, called **mutual exclusion**, prohibits multiple processes from simultaneously accessing the same shared memory location. A region of code containing shared variables which must be executed in mutual exclusion is called a **critical section**. Processors that support specialized instructions which can perform two actions atomically, such as *test-and-set*, *replace-add*, or *swap*, can implement hardware locks to enforce mutual exclusion [Ray86]. Software solutions are possible as well, the most common form using non-negative integer counting variables called **semaphores** and their associated atomic P and V primitives [Dij68]. If $s = 0$, P(s) halts the invoking process until s is positive. If $s > 0$, P(s) decrements s by 1 and allows the invoking process to continue. V(s) increments s by 1. So if s is initially set to 1, the sequence {P(s), critical section, V(s)} enforces mutual exclusion on the critical section. Conditional critical sections or monitors can be used to handle more general situations [And85].

A second type of synchronization, called **condition synchronization**, is needed to ensure a proper partial ordering of events. The most common technique used to provide proper sequencing is called **barrier synchronization**, in which a barrier is placed within the code for each participating processor. Processors reaching the barrier are required to wait until every participating processor encounters the barrier, at which point execution may proceed. An extension of this scheme, called fuzzy barrier

synchronization, attempts to reduce the time that the processors spend in spin-lock waiting for all processors to encounter the barrier. Instead of a single line of code, the fuzzy barrier specifies a section of code in each processor in which synchronization can take place [Gup89]. In message-passing multicomputers, **send** and **receive** primitives are commonly used to enforce sequence control.

1.3.3. Load Balancing

Assigning a heavy computation load to some processors and a light load to others can also cause a degradation in speedup. A proper **load balance** distributes the computation load evenly across all the processors to maximize efficiency. The ability to load balance depends heavily on the program partitioning and the efficacy of the scheduling algorithm used to map the program onto the physical architecture. Notice that load balancing and minimization of interprocessor communication are conflicting goals, because whereas load balancing tends to disperse tasks across different processors, minimization of IPC tends to cluster nodes on just a few processors. The trade-off between these two objectives is an important scheduling issue.

1.4. CONCLUSION

Regardless of what hardware architecture or software programming paradigm is used, there are fundamental difficulties that arise when trying to make processors cooperate on a common problem. The efficient coordination of parallel processors requires both a program partitioning which matches the available hardware, and a scheduling strategy which considers the physical hardware architecture and includes the interprocessor communication and synchronization overheads created by data exchange.

2

SCHEDULING

The simplest problems often have the hardest solutions

— Anonymous

Scheduling theory encompasses a vast body of literature which spans many disciplines, including operations research, industrial engineering, electrical engineering, applied mathematics, and computer science [Con67] [Cof73]. Scheduling problems are classified into a few major groups.

The general **job shop problem** consists of n jobs $\{J_1, J_2, \ldots J_n\}$ which must be processed through m machines $\{M_1, M_2, \ldots M_m\}$. Each job must pass through each machine exactly once; however, the order of passage through the machines may vary from job to job. If the order in which the machines are encountered is constrained to

be the same for each job, this is called a **flow shop problem**. Further details concerning these two scheduling classes can be found in [Fre82].

As opposed to job-shop and flow-shop scheduling, **project scheduling** is normally associated with a one-time project, such as the construction of a skyscraper or an airplane. The two most popular project scheduling techniques are the **Critical Path Method** (CPM) and the **Project Evaluation and Review Technique** (PERT) [Hor80]. These methods model a project as a network of tasks connected by precedence constraints, and identify the critical or bottleneck path through the network. The critical path method starts from a network flow graph. While tasks can reside on either nodes or arcs, we will assume that the nodes represent the activities, and the arcs represent the precedence constraints. Without loss of generality, this flow graph can be assumed to have exactly one initial node and one terminal node. An example project network graph is shown below in figure 2-1, where the estimated task duration t(i) is indicated directly below each task i.

The CPM method first traverses the graph in the forward direction to find the earliest possible start times for each task. After setting the earliest start time (EST) of the ini-

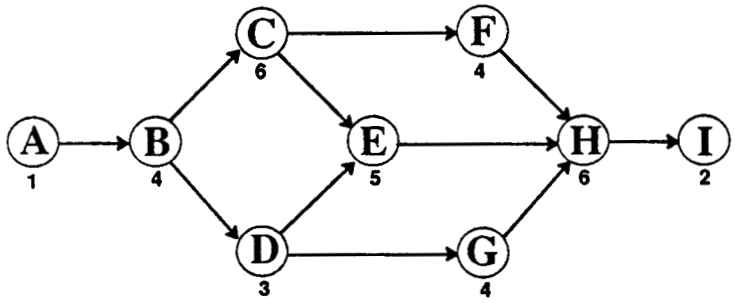


Figure 2-1. An example network flow diagram

tial task to 0, the procedure sweeps through the graph, assigning each task i its earliest start time $EST(i)$ according to the formula

$$EST(i) = \max_j [EST(j) + t(j)] \tag{2.1}$$

where the maximization operation is taken over all predecessor activities j.

Second, the CPM technique traverses the graph in the backward direction to find the latest possible start times if the project is to be completed at the earliest possible time. The latest start time of the terminal task is initially set to its earliest start time, which was calculated in the forward pass. The procedure then sweeps through the graph backwards, assigning latest start times $LST(i)$ according to the formula

$$LST(i) = \min_j [LST(j) - t(j)] \tag{2.2}$$

for all successor activities j. Third, the CPM method assigns each task its **total slack**, defined as the difference between the latest and earliest start times. This quantity represents the amount of time the task can be delayed without affecting the smallest project duration. Tasks that have zero slack are said to lie on the **critical path**, because any delay in executing any of these tasks results in an extended project duration. Tasks with nonzero slack have some flexibility because their start times can be adjusted within the limits of the slack time without changing the project finishing time. The earliest start time, latest start time, and total slack for each task in figure 2-1 are shown in table 2-1 below. After examining this table, the critical path

Node	A	B	C	D	E	F	G	H	I
Earliest Start	0	1	5	5	11	11	8	16	22
Latest Start	0	1	5	8	11	12	12	16	22
Total Slack	0	0	0	3	0	1	4	0	0

Table 2-1. Earliest start times, latest start times, and total slack for each task

through the graph can easily be identified as {A B C E H I}.

The PERT technique extends the CPM method by incorporating uncertainty in the task duration times. Three values are specified for each task duration estimate : a most likely duration, an optimistic duration, and a pessimistic duration. A probability distribution (often a beta distribution) is specified using these three values and the CPM formulation can be used to make probabilistic statements about the project duration. Notice that by allowing all tasks to be executed at their earliest start times, the CPM/PERT formulation implicitly assumes that an infinite number of processing resources are available.

2.1. MULTIPLE PROCESSOR SCHEDULING

Scheduling for multiple processor systems has a very rich and distinguished history [Bus74] [Cof76] [Gon77] [Hu61] [Gar78] [Ram72] [Ull73]. The processor scheduling problem is to map a set of precedence-constrained tasks $\{T_i\}$ i = 1 ... n, onto a set of processors $\{P_k\}$ k = 1 ... p to minimize a specified objective function, such as schedule length or mean flow time. An acyclic precedence graph is commonly used to describe the interrelationships among the tasks, where an arc A_{ij} directed from task T_i to T_j indicates that T_i must precede T_j in execution. This problem differs from project scheduling in that there are a fixed number of processing resources, which makes the problem harder. Multiple processor scheduling strategies can be divided into **preemptive** and **nonpreemptive** techniques. In preemptive scheduling, a task which is executing on a processor can be interrupted in the middle of execution to allow another task to be executed, with the original task resumed at a later time. In nonpreemptive scheduling, any task which has started execution on a processor must

be allowed to run continuously until completion. In general, preemptive strategies generate better schedules than nonpreemptive techniques due to their greater flexibility; however, preemptive methods also incur additional context-switching overhead. In this thesis, we restrict our attention to nonpreemptive scheduling methods.

2.1.1. Scheduling Complexity

A key issue in the study of processor scheduling is the complexity of scheduling problems. If n denotes some reasonable measure of the input size of an algorithm (e.g. number of nodes), we say that the running time of a program $T(n)$ is $O(f(n))$ if there are constants c and n_o such that $T(n) \leq cf(n)$ for all $n > n_o$. A program whose running time is $O(f(n))$ is said to have growth rate $f(n)$, and we refer to an efficient algorithm as one which has a polynomial growth rate. However, most scheduling problems belong to a class of problems for which no such solutions have been found. We briefly introduce this subject and direct readers to [Gar79] for further details.

Informally, class P refers to the set of decision problems which can be solved by a deterministic algorithm in polynomial time, while class NP refers to the set of decision problems which can be solved by a nondeterministic algorithm in polynomial time. However, the notion of a solution becomes ambiguous in the nondeterministic case, because when faced with a range of possibilities, a nondeterministic algorithm has the ability to instantly guess the correct choice. All that remains is to verify the solution in polynomial time. Thus, a nondeterministic polynomial-time solution is more accurately described as polynomial-time verifiability; the time required to search for the correct solution is excluded. Although nondeterministic algorithms are obviously more powerful than deterministic algorithms (i.e. P $\subseteq$ NP), there has not been a

single problem which has been proven to be in NP but not in P. That is, the question of whether P=NP is still open. A problem is referred to as being **NP-complete** if it belongs to the set of "hardest" problems in NP, which are equivalent in the sense of polynomial time reducibility. In other words, if any of the NP-complete problems can be proven to lie in class P, then all problems in NP also lie in P. Conversely, if any NP-complete problem can be proven intractable, then all NP-complete problems are similarly intractable. To reiterate, the fact that a problem is NP-complete does not prove that any algorithm which constitutes a solution must be exponential in complexity, but rather that the problem belongs to a class for which no polynomial-time solutions have been found. The prevailing belief is that such solutions do not exist.

The notion of strong NP-completeness addresses the fact that some NP-complete problems rely upon the possibility of extremely large input numbers to gain intractability. An example is the PARTITION problem, which asks if a set of objects with individual sizes s(a) can be partitioned into two subsets such that the sum of the object sizes in each subset are equal. Although the problem as stated is NP-complete, the introduction of a bound on the maximum size of any object causes the problem to fall into class P. To distinguish these types of problems, the concept of a pseudo-polynomial time algorithm was introduced, which have complexity bounded by a polynomial function in two variables: the length of the input, and the maximum number contained in the input. Such algorithms can be regarded as being slightly more powerful than polynomial-time algorithms because they allow the possibility of requiring an exponential amount of time when faced with inputs containing exponentially large numbers. Number problems which are NP-complete in the strong sense are distinguished from their weaker counterparts in that they cannot be solved by a

pseudo-polynomial time algorithm unless P=NP.

For the nonpreemptive processor scheduling problem where the goal is to construct a minimal-length schedule, there are two special cases in which efficient optimal algorithms are known to exist. The first case is when all the tasks have equal execution times and there are only 2 processors, and the second case is when the tasks have equal execution times and the precedence constraints form a rooted tree. If one increases the number of processors in the first case, or give the tasks unequal execution times, the problem immediately falls in the class of NP-complete problems. For this reason, algorithms which obtain optimal solutions are usually discarded in favor of heuristics which obtain a fairly good suboptimal solution in a reasonable time.

2.1.2. List Scheduling

Perhaps the most widely used scheduling heuristic is the list scheduling algorithm. List scheduling is a technique in which tasks are assigned priorities and placed in a list, sorted in order of decreasing priority. Nodes whose predecessors have been completed are designated as being **ready** (for execution). A global time clock serves to regulate the scheduling process. Processors which are idle at the current time are designated as being **available** (for assignment). When a processor is available, the first ready node in the list is assigned to be executed on that processor. After assignment, the processor is removed from the available processor list, the node is deleted from the priority list, and this process is repeated until the available processors have been exhausted. The time clock is then incremented until some processors finish execution of their allotted tasks and are available once again. The algorithm terminates when all nodes have been scheduled.

List scheduling can lead to some surprising scheduling results, as shown by Graham in his classic work on "scheduling anomalies" [Gra69]. In this paper, Graham demonstrated the counter-intuitive notions that increasing the number of processors, reducing the execution times of some tasks, or weakening the precedence constraints can all lead to an increase in makespan.

The most well-known list scheduling strategy is the HLFET scheme [Ada74], or critical path algorithm, an extension of Hu's basic work [Hu61]. In this procedure, the endnodes of the graph are connected to a dummy terminal node N_t and the priority of each node N_i is set equal to its **level**, defined as the sum of execution times along the longest directed path from N_i to N_t. List scheduling is then invoked using these priorities. To minimize confusion in terminology, we will refer to these levels as **static levels**, denoting the static level of node N_i as $SL(N_i)$. Several studies have revealed that the HLFET algorithm demonstrates near-optimal performance when IPC costs are not included [Ada74] [Koh75]. The success of this technique stems from the accurate representation of a node's priority by its static level, which causes each successive scheduling step to shorten the longest path to completion.

Scheduling without consideration of communication costs is now considered a mature field of investigation. Therefore, in the remainder of this thesis, we will only consider those algorithms which take interprocessor communications into account.

2.2. A SCHEDULING TAXONOMY

To classify scheduling algorithms, we use the taxonomy presented in [Lee89], which divides scheduling strategies into the four classes shown below.

1) Fully Dynamic
2) Static Assignment
3) Self-Timed
4) Fully Static

The scheduling operation is subdivided into three operations: *node assignment* (assigning nodes to processors), *node ordering* (determining the ordering of nodes on each processor), and *node timing* (designating the exact time interval for node execution). Scheduling algorithms are then categorized according to whether these operations occur at compile-time or at run-time as shown in figure 2-2. The fully dynamic case incurs the most run-time overhead, because all three of these operations are performed at run-time. In static assignment, nodes are assigned processors at compile-time but the sequencing and timing operations are performed at runtime based on data availability. Self-timed scheduling adds the sequencing information for each processor at compile-time, but determines the exact firing instants at run-time. The fully static case specifies all three operations at compile-time, thereby incurring the least amount of run-time overhead. However, in moving from the fully dynamic to the fully static case, the number of algorithms which are applicable under these scheduling classes decreases substantially. The domain of applicability in the fully static case is limited to the class of algorithms which fit the SDF model and have deterministic

	assignment	ordering	timing
fully dynamic	run	run	run
static-assignment	compile	run	run
self-timed	compile	compile	run
fully static	compile	compile	compile

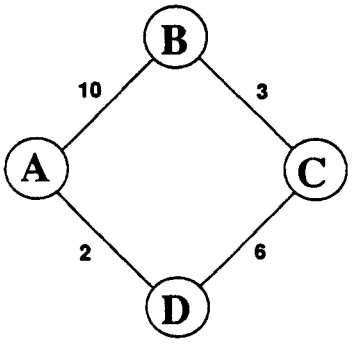
Figure 2-2. A categorization of scheduling algorithms

node execution times. Notice that the other scheduling classes can be derived from fully static scheduling by discarding one or more pieces of information.

2.2.1. Static Assignment Algorithms

Static assignment, or task allocation algorithms, address the "mapping problem" by assigning tasks to processors at compile-time, but performing the sequencing and timing operations at run-time. These schemes usually attempt to minimize some weighted function of the computation and communication costs. Notice that minimizing such functions is not the same as minimizing the makespan, because precedence constraints have not been taken into account as a result of the lack of sequencing information. For this reason, application of such algorithms is usually limited to data flow machines or restricted to situations in which all tasks are independent. Graph theoretic, integer programming, clustering, and queuing approaches have all been used for static assignment.

One popular approach is the network flow technique proposed by Stone [Sto77], which is based on the classic Max Flow/Min Cut theorem of Ford and Fulkerson. This scheme is intended for use in a heterogeneous processor environment and attempts to minimize the total sum of execution and communication costs. The idea is to formulate the assignment problem as a network flow diagram so that cutsets of the diagram correspond to processor assignments in a one-to-one fashion. Using this formulation, the optimal assignment corresponds to the minimum weight cutset in the network flow graph, which is obtained using the $O(n^5)$ algorithm of Ford and Fulkerson, as modified by Edmonds and Karp [Edm72]. Consider the example graph shown in figure 2-3, which will be mapped onto a heterogeneous two-processor system. The arc weights represent the communication cost in separating the adjoining nodes on



Execution Times

	P0	P1
A	6	10
B	5	9
C	6	3
D	4	1

Figure 2-3. An example graph

different processors, while the execution times of each node on processors P0 and P1 are shown in the adjacent table. The corresponding network flow diagram is shown below in figure 2-4, which is constructed by adding nodes S0 and S1, representing processors P0 and P1 respectively. Arcs are then added from each node N_i to S0, weighted with the execution time of N_i on P1. Similarly, arcs are added from each node N_i to S1, weighted with the execution time of N_i on P0. Each set of edges separating nodes S0 and S1 (cutset) corresponds in a one-to-one fashion with a module assignment, where the cost of the assignment is the sum of weights of the

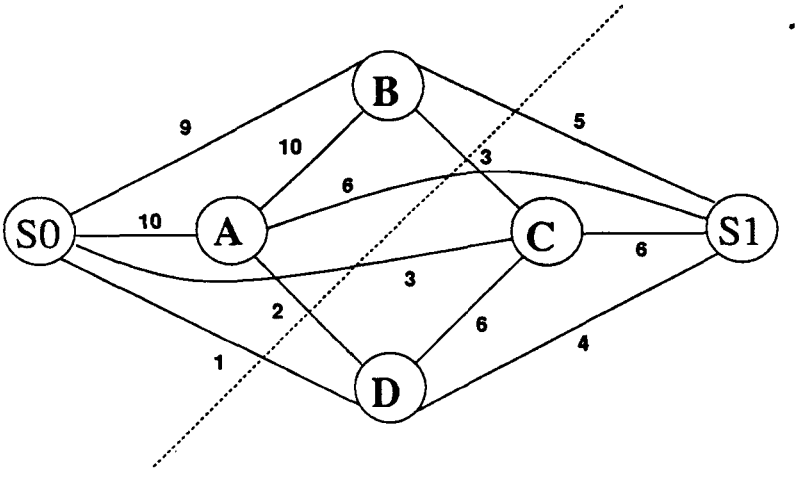


Figure 2-4. The network flow graph and optimal cutset

edges that were cut. All that remains is to use the Ford-Fulkerson algorithm to find the minimum-weight cutset, which for this example is shown by the dotted line in figure 2-4. This optimal cutset partitions nodes A and B on P0 and nodes C and D on P1. The total sum of execution and communication costs for this assignment is 20. If there are p processors, where $p > 2$, then the arcs from N_i to S_j are assigned a weighted sum of the execution times of N_i on each processor, and a p-way cutset is used to find the optimal assignment.

While extremely elegant, this technique becomes computationally intractable for more than three processors. Another factor limiting its usefulness is that minimizing the total sum of computation and communication time does not usually achieve good load balancing. In the homogeneous processor case, this algorithm will always assign every task to a single processor.

An extension of Stone's technique which remedies these problems has been suggested by Lo [Lo88]. To achieve manageable complexity, the p-processor network is converted into a 2-processor network by treating the processor under consideration (P_i) as a single entity, and combining all other processors into a super-processor group. All edges between a task and the processor nodes (S_j) within the super-processor group are replaced with a single edge whose weight is the sum of the weights on the original edges. The Max Flow/Min Cut algorithm is applied to this 2 processor system to find the tasks to be assigned to processor P_i. After applying this procedure to each processor, the assigned nodes are removed, the edge weights are redefined to reflect the partial assignment, and these steps are repeated. The procedure terminates when no further assignment occurs. If all nodes are assigned, the solution is provably optimal (under Stone's criterion). If some tasks have still not been assigned processors, some

additional phases are invoked. To avoid the algorithm's propensity to use very few processors, Lo proposes the use of "interference costs" which promote concurrency by pushing tasks onto different processors with a repulsive force which is dependent upon the relative amount of computation and communication costs incurred.

Bokhari proposes another scheme intended for heterogeneous processors which minimizes the total sum of execution time and communication time costs [Bok81]. Programs are expressed as invocation trees, where the nodes represent functions, and the arcs represent data paths. The time to execute node i on processor m, E_{im}, and the communication cost for transferring D_{ij} data units between any processors p and q, $S_{pq}(D_{ij})$, are assumed to be known in advance. This method transforms the invocation tree into an assignment graph by appending a dummy source node at the root of the invocation tree, appending dummy terminal nodes at the leaves of the invocation tree, and expanding each node of the invocation tree into a layer of subnodes, one subnode per processor. Each subnode is labeled with a pair of numbers (i,m) which denotes the assignment of node i on processor m. For example, the invocation tree shown at the left of figure 2-5 is expanded into the assignment graph shown at the right of figure 2-5. The directed arc connecting subnode (i,m) to subnode (j,n) has weight $E_{jn} + S_{mn}(D_{ij})$ reflecting the total cost in executing node j on processor n, given that node i has been assigned to processor m. An assignment tree is defined as a tree which connects the source node to all terminal nodes and contains exactly one node from each layer of the assignment graph. Each assignment tree corresponds uniquely to an assignment of nodes to processors, where the sum of the arc weights in the assignment tree reflects the total cost of this assignment. The problem has thus been reduced into finding the minimum weight assignment tree in the assignment

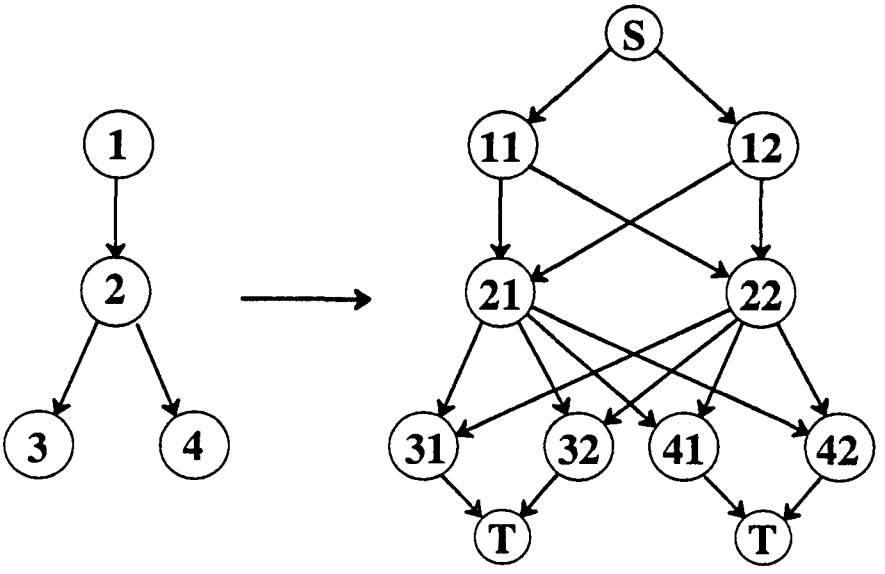


Figure 2-5. Expansion of invocation tree into assignment graph

graph, which can be performed in $O(np^2)$ using a dynamic programming approach.

Although this algorithm provides an optimal solution in this special case (precedence relationships form a tree), it does not necessarily minimize the finishing time. Processors are assumed to have infinite processing capability, because the same processor can be used in different branches in the tree at the same time. This method also suffers the same deficiency as Stone's method in the homogeneous processor case.

To promote a more balanced load distribution, Chu and Lan propose a heuristic which attempts to minimize the load of the bottleneck processor [Chu87], where the load reflects both execution and communication time costs. This scheme first combines nodes into groups using ratios to indicate the relative importance of execution, communication, and precedence effects. The groups are then assigned to processors to minimize the load of the most heavily utilized processor.

Gylys and Edwards propose two module clustering schemes for static assignment [Gyl76]. The first method fuses together the two modules with maximum intermodule communication between them if there is a processor eligible to execute both of them. If not, the next most expensive pair is considered and the process is repeated until all eligible pairs of modules have been combined. The second method defines a distance function between modules based on the amount of data transferred. Starting from an initial set of cluster centroids (center of mass), the procedure searches for the module with smallest distance from a cluster centroid and merges the two if all constraints are satisfied. The centroids are repetitively updated after each merging step and the algorithm terminates when there are no further combination steps.

2.2.2. Fully Static Scheduling

As opposed to static assignment scheduling, fully static scheduling allows computation of the schedule makespan at compile-time, based on the given execution and communication time estimates. This class is especially important for signal processing applications because of the need to satisfy real-time constraints. Most DSP algorithms fit the SDF model and are eligible for fully static scheduling.

In practice, self-timed schedules are usually used when executing programs on actual hardware because they provide a greater robustness when execution time estimates are inaccurate. However, self-timed schedules are normally derived from fully static schedules by discarding the timing information and adding synchronization primitives to ensure proper ordering. For these reasons, we concentrate our attention on the fully-static scheduling class in which minimizing makespan is the scheduling goal. The four most popular approaches to this problem are limited search strategies, modified list scheduling techniques, clustering algorithms, and simulated annealing.

Limited Search Strategies

Greenblatt and Linn propose a branch and bound technique for scheduling [Gre87]. Branch and bound is an orderly method of exhaustive search which eliminates possibilities through calculation of bounds on partial solutions. The search space can be envisioned as a tree with each arc representing a possible action taking place, such as node N_i being scheduled on processor P_j. Since each successive action builds on the actions which have taken place earlier, the set of partial solutions branches out exponentially as the tree depth increases. The complete set of solutions lies at the leaves of this tree. Branch and bound algorithms try to prune each branch as early as possible, rather than enumerate each complete solution in a sequential fashion. A bound on the performance of a partial solution is calculated and if this bound is poorer than the best solution encountered so far, this branch (and all following branches) are pruned from the tree, because no solution along this path can possibly be optimal. The most important consideration is the choice of the static evaluation function (SEF) used to calculate the bound, which in the current context must estimate the final schedule length from a partial scheduling of nodes to processors. A tradeoff exists between the quality of the estimate and the search time. To ensure optimality, the SEF cannot overestimate the schedule length, lest the optimal solution be pruned away. At the same time, if the SEF severely underestimates the schedule length, very few solutions will be pruned away, causing an enormous search time.

In this technique, the authors first propose a function called OPT which is supposedly guaranteed to underestimate the schedule length. If f_i denotes the time that processor i becomes free in the partial schedule, F = max $\{f_i\}$, W is the sum of the execution times of all unassigned nodes, and p is the number of processors, then

$$OPT = F + \max\left[0, \sum_{i=0}^{p-1} \frac{W - f_i}{p}\right] \quad (2.3)$$

However, consider the example partial 2-processor schedule shown in figure 2-6 and assume there is one task left to be scheduled which has execution time 8. Clearly, scheduling this task on P1 produces a makespan of 10; however, the OPT function has a value of 12, which overestimates the actual schedule length. This counterexample shows that using this OPT function could result in the optimal solution being pruned away. In keeping with the authors' objective, we propose a new function OPT2 which will always underestimate the schedule length as

$$OPT2 = F + \max\left[0, \frac{1}{p}\left(W - \sum_{i=0}^{p-1}(F - f_i)\right)\right] \quad (2.4)$$

The authors go on to propose two additional suboptimizing heuristic functions. The first heuristic function attempts to include idle times, which may result from precedence constraints or interprocessor communication:

$$H1 = OPT + (NumberofRemainingArcs) \times \frac{IdleTimeinPartialSchedule}{ArcsConsumedinPartialSchedule} \quad (2.5)$$

The second heuristic function tries to estimate schedule length by assuming that the ratio of the length of the partial assignment to the number of tasks assigned stays approximately constant.

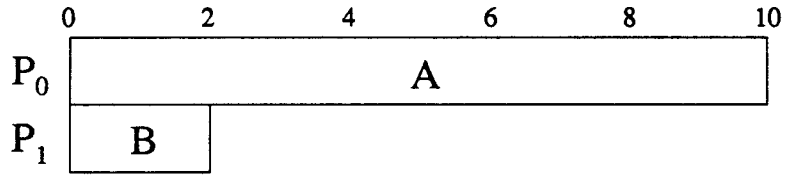


Figure 2-6. Partial 2-processor schedule

$$H2 = \frac{Length of Partial Assignment}{Fraction of Tasks Assigned} \quad (2.6)$$

H2 was found to outperform H1 at the cost of increased searching time. The complexity of these branch and bound schemes severely restricts the size of the problem instances which can be solved using this approach. The largest input graphs tested in the study contained only eight nodes.

List Scheduling

Yu proposes a technique which modifies the classical HLFET list scheduling technique to account for communication delay in the context of a fully-interconnected processor network [Yu84]. At each step, this technique picks the node with highest static level, and schedules it on the *available* processor which completes its execution at the earliest time. It then removes the node from the list of ready nodes, removes the processor from the list of available processors, and continues this procedure until the list of available processors is exhausted. At this point the global clock is updated until some processors finish execution of their appointed tasks and become available for assignment once again. While the basic idea of this algorithm is sound, the use of the classical list-scheduling methodology leads to a performance flaw which will be exposed in Chapter 3. Yu goes on to propose more complicated techniques which use combinatorial matching algorithms to pair sets of nodes and processors at each step.

Clustering

Clustering algorithms have gained wide popularity for scheduling in the presence of interprocessor communication costs. By forcing nodes which communicate heavily onto the same processor, these strategies produce mappings which avoid excessive IPC cost. Since processor assignments need only be assigned for each cluster, rather

than for each node, clustering also reduces the time needed for scheduling.

The *linear clustering* technique, proposed by Kim and Browne [Kim88], iteratively applies a critical path algorithm to transform the graph into a virtual architecture graph (VAG), which consists of a set of linear clusters and the interconnections between them. A linear cluster is a degenerate tree in which every node has at most one immediate predecessor and one immediate successor. At each step, the algorithm groups together the most expensive directed path (in computation and communication) into a single linear cluster. The clustered nodes are removed from the graph and this process is iteratively repeated until the entire graph has been divided into clusters. After some refinement procedures, the algorithm applies graph-theoretic techniques to map the virtual architecture graph onto the physical processor architecture.

Sarkar presents a complete partitioning and scheduling approach in [Sar89]. Programs are described using a hierarchical graphical representation called IF1, an intermediate form for applicative languages in which nodes represent operations and edges denote data paths. Nodes can be simple, so that the outputs are direct functions of the inputs, or compound, so that a node represents a complete subgraph in itself. The partitioning and scheduling technique contains four main phases:

1) Cost Assignment
2) Graph Expansion
3) Internalization
4) Processor Assignment

The cost assignment phase estimates computation and communication costs in the program. Profile data is used to estimate node execution times. Communication times are determined by examining the number of data units to be transferred, which implicitly assumes a homogeneous communication architecture. The graph expansion

phase attempts to expose sufficient parallelism in the graph to accommodate the number of given processors, while at the same time limiting the number of tasks to a manageable number. A depth-first traversal of the IF1 graph is used to generate increased concurrency through recursive subdivision of compound nodes. Compound node N_i is expanded into its constituent subnodes if there is insufficient parallelism to keep P-1 processors busy while node N_i is executing. A granularity threshold factor limits the number of nodes by lower bounding the size of the nodes that can be expanded. A second traversal is used to merge very small IF1 nodes into a single task.

The *internalization* clustering phase clusters nodes together in an attempt to minimize the schedule length on an unbounded number of processors. The algorithm initially places each node in a separate cluster and considers the APEG arcs in descending order according to the amount of data transferred over each arc. Given arc A_{ij} (which connects nodes N_i and N_j), the algorithm merges the clusters containing these nodes ($C(N_i)$ and $C(N_j)$) to "internalize" any communications between nodes in these respective clusters. The algorithm accepts this cluster merging step if it does not increase PARTIME, the estimate of the parallel execution time of this clustered graph on an infinite number of processors. Otherwise, the clusters are unmerged and the next arc is considered.

The parallel execution time estimate for the given set of clusters is computed in a manner resembling classical CPM (critical path method) methodology. Forward and backward passes through the graph are used to find the earliest and latest start times for each node, where the communication costs between nodes in separate clusters are included. However, this procedure differs from the CPM techniques in that nodes in the same cluster are constrained to be executed sequentially on the same processor.

To enforce this constraint, the algorithm sorts the nodes in each cluster in increasing order of their latest starting times and appends additional precedence constraints to ensure this ordering. This approach is in accordance with Jackson's rule for scheduling nodes according to nondecreasing deadlines.

When the list of APEG arcs is exhausted, a processor assignment phase uses a modified list scheduling approach to map the finished clusters to the physical processors. The procedure temporarily shifts each unassigned cluster onto each of the processors in turn and estimates the parallel execution time in each case. The cluster is mapped onto the processor which yields minimum PARTIME. The internalization approach is motivated by the observation that if two nodes are assigned to the same processor in the infinite processor case, they should also be assigned to the same processor in the finite processor case.

Simulated Annealing

Simulated annealing is an optimization technique which mimics the annealing process by which a physical system reaches a state with globally minimum energy [Kir83] [Haj85]. Annealing is the process of melting a substance and then decreasing the temperature very slowly, spending a lot of time at temperatures near the freezing point to allow the substance to form a regular, stable, crystal lattice. In simulated annealing, an objective, or cost function is used to measure the degree of "goodness" of some property of the system at each state, and a temperature schedule which decreases very slowly (usually logarithmically) is used to determine the probability of accepting a step to a neighboring state which worsens the objective function. If V(s) is the value of the minimization objective function at the current state, and V(s') is the value at a potential next state, then this new state is accepted with probability

$$p_k = \exp\left[-\frac{V(s') - V(s)}{k_b T}\right]. \qquad (2.7)$$

This means that steps in the direction of improving the objective function are always accepted, while steps in the direction to worsen the objective function are taken with a probability which decreases as the temperature decreases.

This procedure should be contrasted with the classical iterative improvement technique, in which movement to a neighboring state is only permitted if the new state shows an improvement in the objective function over the previous state. By only allowing steps in the direction of increasing improvement, this procedure tends to get stuck in local minima, instead of finding the global minimum. The physical analogy of this technique is the rapid quenching of a material from a high temperature down to T = 0, which usually results in a metastable final state.

A limitation of the simulated annealing process is that it executes very slowly due to the logarithmic temperature schedule. It should therefore only be used in applications in which computation time is not an important issue.

A simulated annealing scheduling scheme proposed by Bollinger and Midkiff [Bol88] generates moves by pairwise exchange of nodes. This algorithm first applies a process annealing phase to assign nodes to processing elements in a multicomputer, then uses a connection annealing phase to schedule traffic onto interconnection links. The cost function used in the process annealing phase is $\sum_{j,k}[w_{jk} \times d(PE_j, PE_k)]$, where w_{jk} represents the number of data units passed between nodes j and k, and d(PE_j, PE_k) represents the distance in hops between processing elements j and k. At very low temperatures, the cost function is modified slightly so that moves which increase $\max_{j,k}[w_{jk} \times d(PE_j, PE_k)]$ are rejected. Next, using the node mapping computed in

the process annealing stage, the connection annealing phase attempts to route each connection using using a cost function which is the sum of the traffic intensities in the network links supporting the connection. This work only addressed the limited case in which the number of nodes did not exceed to the number of processing elements.

Another communication traffic scheduling scheme was proposed by Bianchinni and Shen [Bia87], which used a traffic smoothing algorithm based on a fluid flow model incorporating link capacities and traffic flows. A weakness of both communication traffic scheduling methods is the dichotomy introduced in first scheduling nodes onto processors and then scheduling communications onto links. Both works acknowledge that the assignment of nodes to processors has a substantial impact on the ease of traffic scheduling. Furthermore, neither of these techniques handle contention for shared communication resources.

2.3. THE SCHEDULING PROBLEM

The problem we are addressing is the nonpreemptive compile-time scheduling of precedence graphs onto multiprocessor architectures, as shown in figure 2-7. Our scheduling goal is to minimize the makespan when all the interprocessor communication overheads are included, which is equivalent to maximizing the speedup. This scheduling problem is NP-complete in the strong sense, even if there are an infinite number of processors available [Sar89].

The input to the scheduling algorithm is an acyclic precedence expansion graph (APEG) G = {N,A}, where N is a set of nodes (tasks) $\{N_i\}$ i = 1 ... n which represent program computations, and A is the set of directed arcs $\{A_{ij}\}$ which represent both precedence constraints and data paths. Each arc A_{ij} carries label D_{ij} which specifies

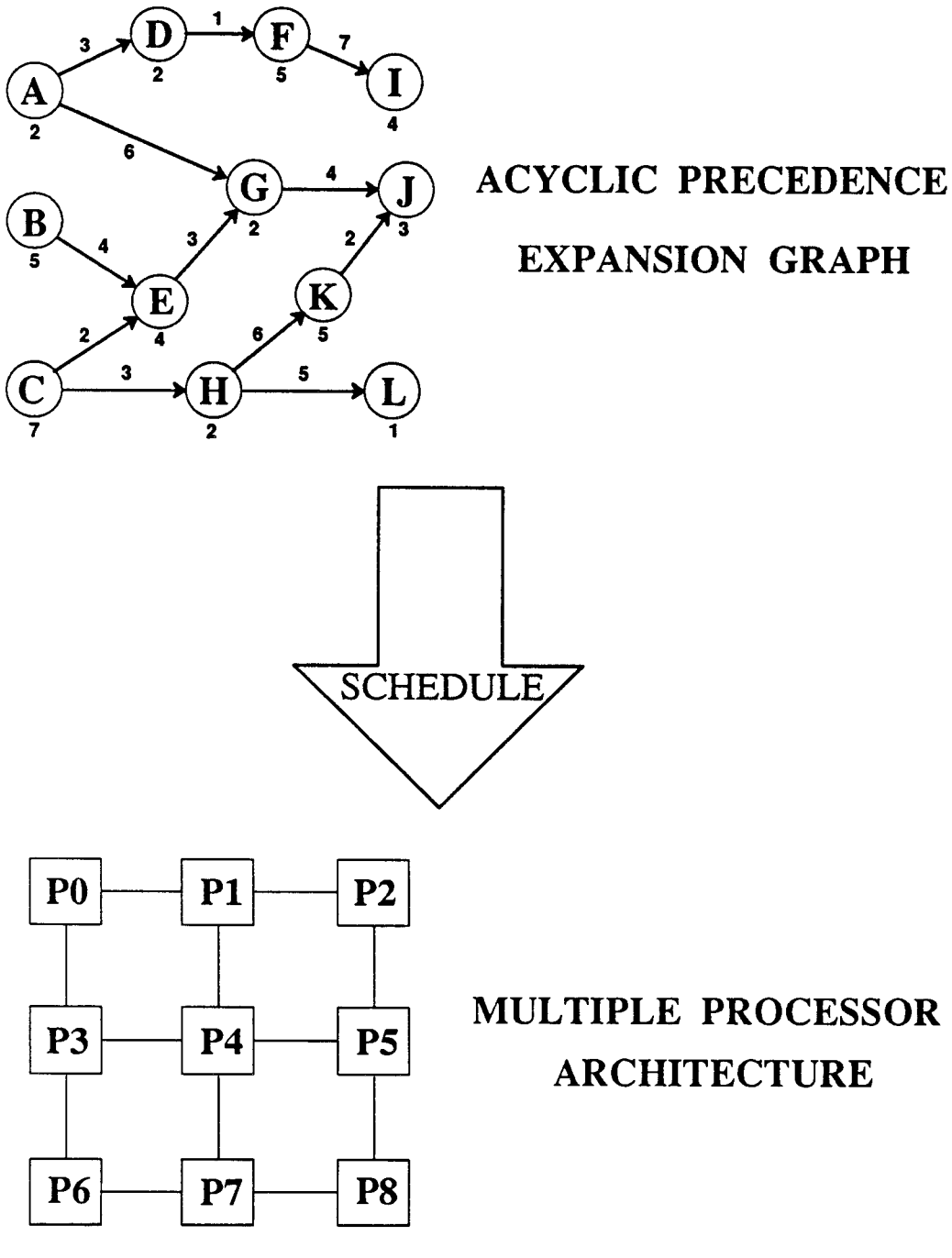


Figure 2-7. The scheduling context

the amount of data (in bits, bytes, or words) that N_i passes to N_j on each invocation. These graphs can derived from algorithmic descriptions which fit the synchronous data flow model using the algorithm given in Appendix I. We assume without loss of

generality that each APEG has exactly one terminal node. This condition can be enforced by connecting multiple endnodes to a dummy terminal node N_t. The target architecture contains a set of potentially heterogeneous processors $\{P_k\}$ k = 1 ... p. We assume that a fairly accurate estimate of the execution time of node N_i on every processor P_j, $E(N_i, P_j)$, is available at compile-time for each node-processor pair. If node N_i cannot be executed on processor P_j, the execution time $E(N_i, P_j)$ is infinite. For scheduling purposes, we assume that each node is indivisible; no attempt will be made to utilize intranode parallelism.

The scheduler is part of an interactive design system for digital signal processing (DSP) called Gabriel [Bie90], which allows rapid prototyping of new DSP algorithms and architectures. A block-diagram of this design environment is shown in figure 2-8. Using a graphical user interface, an algorithm designer can quickly construct new signal processing algorithms by connecting together blocks which represent DSP functions. These blocks range in granularity from simple operators such as adders or multipliers, to higher level functions such as FFT's or FIR filters, to complex signal processing subsystems such as filter banks or speech coders. After specifying the desired architecture (using a set of parameters which capture its characteristics), the user invokes the scheduler, which responds with feedback information concerning the makespan, the degree of processor utilization, and the time spent in interprocessor communication. The designer can use this information to answer questions such as:

"Can this algorithm be run in real-time on this architecture?",
"Are more processors necessary?",
"Are there too many processors?",
"Is more interprocessor communication bandwidth needed?",
"Would a different processor topology be more effective?",
"Can be the problem be scaled upward in size?",
"Would a different implementation of this algorithm be more efficient?",
"Is a different algorithm needed for this problem?"

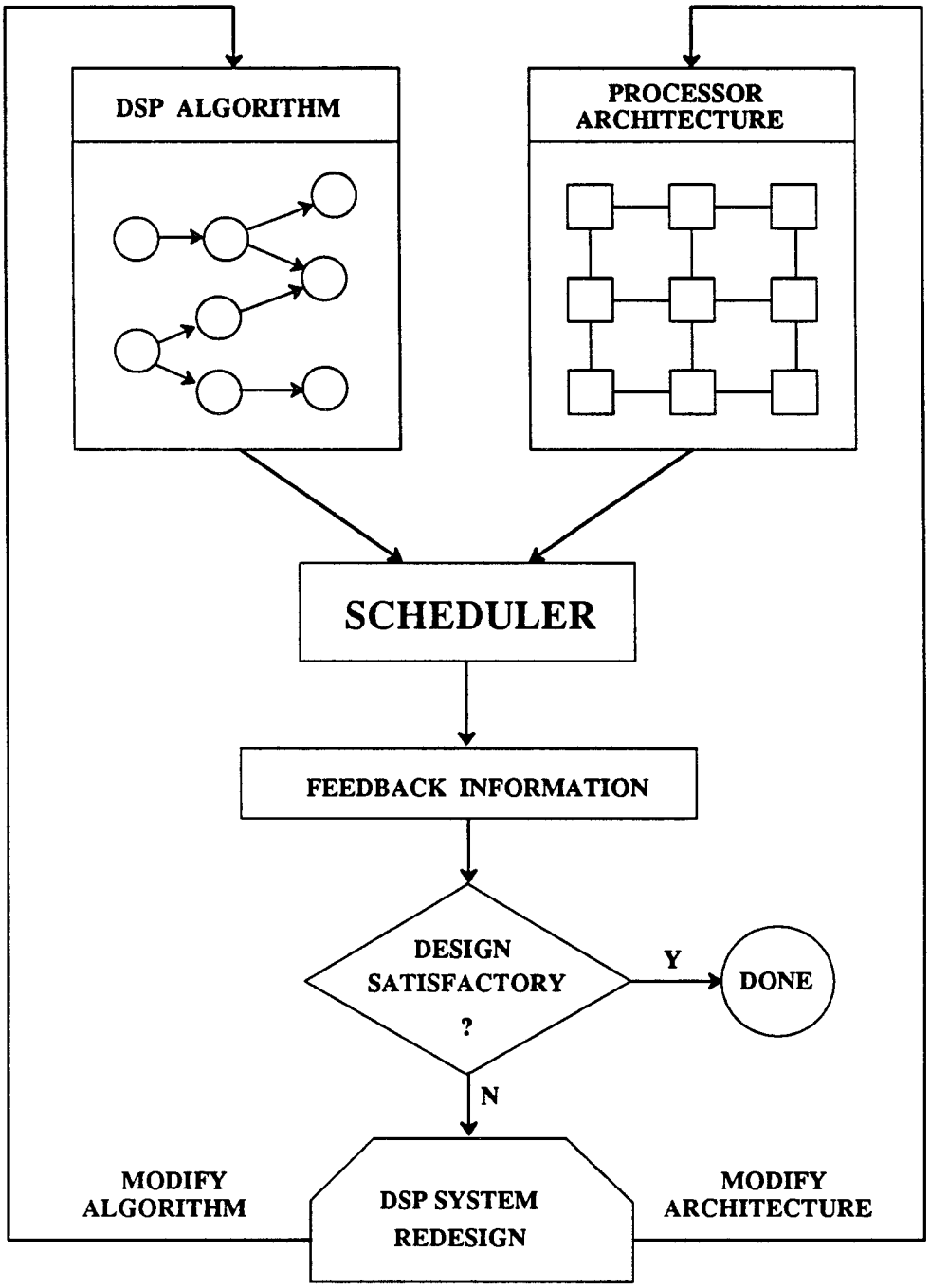


Figure 2-8. The DSP prototyping environment

Both the algorithm and architecture can be successively refined in several design iterations to maximize performance.

This environment imposes several stringent constraints on the permissible scheduling strategies. First, the interactive nature of the design approach requires that the scheduling technique execute rapidly. The designer should not have to wait more than a few minutes for feedback information, which necessitates the use of a fast heuristic technique. Second, the scheduling technique must be flexible enough to handle mixed-granularity task graphs. This requires an ability to adjust the tradeoff between the amount of parallelism utilized and the amount of communication overhead incurred. Third, the scheduling scheme must be adaptable to the diverse architectures encountered in digital signal processing. Fourth, both the communication delay and resource contention aspects of the interprocessor communication overhead must be accounted for to permit maximum hardware efficiency.

3

DYNAMIC LEVEL SCHEDULING

I have gotten a lot of results. I know 50,000 things that won't work.

— Thomas A. Edison

Our first compile-time scheduling heuristic, called **dynamic-level scheduling**, is based on an extension of the classical HLFET list scheduling strategy. This technique assumes that the processor architecture contains dedicated communication hardware so that communication and computation can be overlapped. It accounts for interprocessor communication overheads and constraints imposed by the interconnection topology, so that shared resource contention can be avoided. It is applicable to heterogeneous-processor architectures, which are often encountered in signal processing because of the extreme computation demands of many DSP applications.

3.1. HANDLING INTERPROCESSOR COMMUNICATION

If interprocessor communication can be obtained for free, all available task parallelism can be utilized without cost. That is, given enough processors, an optimal schedule can always be constructed by invoking all simultaneously executable nodes on different processors. The reality of nonzero IPC cost induces a tradeoff between the amount of parallelism utilized and the amount of communication overhead incurred. Addressing this tradeoff requires consideration of the task grain size, the amount of parallelism in the graph, the number of processors in the architecture, the processor interconnection topology, the transmission and synchronization delay overheads, and the possibility of communication resource contention.

3.1.1. The IPC Model

To accommodate the multitude of different interprocessor and processor-memory interconnection topologies, we use a single, flexible communication time model. The time C needed for communication is expressed as

$$C = (x + yD) \circledast_k zH \tag{3.1}$$

where D is the amount of data needed to be transferred (in bits, bytes, or words); H is the number of interprocessor or processor-memory hops between source and destination; x, y, z, and k are constants which specify the processor topology. The notation $\circledast_k$ signifies a special operator defined as follows:

$$\circledast_k = \begin{cases} + & \text{if } k = 0 \\ \times & \text{if } k = 1 \end{cases} \tag{3.2}$$

This operator allows H to be incorporated as an additive or a multiplicative term. We give a few examples below to demonstrate the wide applicability of this model.

Single-bus shared memory multiprocessors can be modeled using constant x to represent the time needed to obtain and release the bus, constant y to reflect the time necessary for transmission and synchronization, and setting constants z and k to zero.

Message-passing multicomputers which propagate datagrams in a store-and-forward manner incorporate the number of interprocessor hops is a multiplicative term in the communication time (k=1). In this scenario, x models any constant header processing time, y represents transmission time, and z is set to unity. Multicomputers using virtual cut-through or wormhole routing immediately forward data at intermediate processing elements as soon as it is received, making the number of hops an additive term (k=0). In this situation, x represents virtual circuit set-up and tear-down time, y reflects the number of channel cycles to transmit a single unit of data, and z represents any forwarding delay at intermediate processors.

In large-scale shared memory multiprocessors, resource reservation and routing can be performed statically to remove the possibility of blocking and the need for intermediate switch buffering, so that the number of hops is an additive term (k=0). Again, x models any constant setup time, y represents transmission and synchronization overhead per data unit, and z reflects the delay through each switching node.

The dynamic level scheduling approach can actually target architectures which are not characterizable using this communication model. Multiprocessors with multiple busses, multicomputer networks with heterogeneous interprocessor links, and even topologies which mix the architectural classes described above can be accommodated. The requirement is that enough information must be supplied to allow the communication time between every pair of processors to be calculated deterministically if resource availability is assured.

3.1.2. Communication Scheduling

Scheduling in the presence of IPC contains two main aspects: assigning processors to computation nodes (mapping), and allocating communication resources for interprocessor data transfers (traffic scheduling). As mentioned earlier, these two problems have traditionally been dealt with separately. A task allocation algorithm first assigns nodes to processors in accordance with some objective function, and then a routing algorithm performs interprocessor traffic scheduling upon the specified node mapping. This separation is impractical, because the ease with which the traffic scheduling can be performed is directly dependent on the properties of the mapping; the best isolated assignment of nodes to processors is invariably suboptimal after simultaneous consideration of both communications and computations.

By addressing both issues concurrently, our scheduling strategy averts overloaded communication resources by adjusting the node-processor mapping accordingly. Just as computations are scheduled upon processors, communications are scheduled upon IPC resources by dedicating the resources used in a data transfer for the duration of the transmission. With guarantee of resource availability, the communication time can be calculated deterministically (or upper bounded) using the locations of source and destination processors, the amount of data to be transferred, and the characteristics of the communication architecture. A routing algorithm, employed by the scheduler, uses knowledge of previous resource usage to reserve a path between source and destination processors for the duration of this time window.

For illustrative purposes, consider the scheduling of the APEG from figure 3-1 onto the target architecture shown in figure 3-2, which consists of four processors interconnected through four full-duplex interprocessor links. For simplicity, assume that the

time needed to communicate D units of data between any two processors is merely D time units, regardless of the distance between them. The upper chart in figure 3-3 shows a possible scheduling of nodes onto processors, while the lower chart in figure 3-3 shows the corresponding scheduling of communications onto links. This simultaneous consideration of spatial (routing) and temporal (scheduling communication time windows) aspects of IPC eliminates the possibility of shared resource contention, which ensures deterministic behavior.

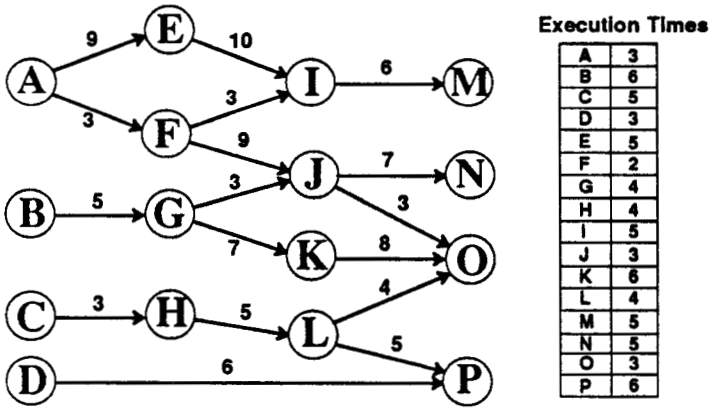


Figure 3-1. An example APEG

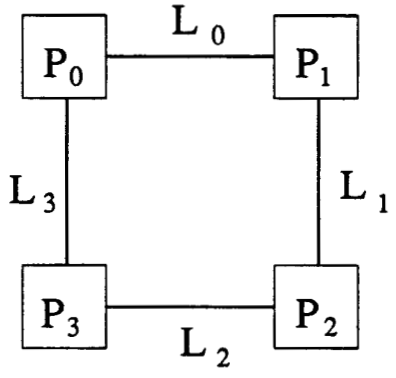


Figure 3-2. A four-processor target architecture

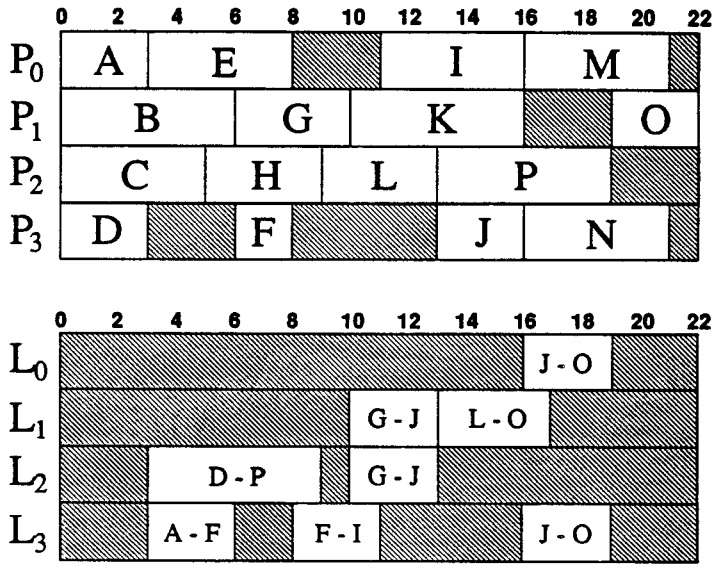


Figure 3-3. Scheduling of nodes onto processors and communication onto links

To permit wide retargetability without sacrificing efficiency, we divide the scheduling algorithm into two components. The first component contains the fixed, architecture-independent scheduling routines, while the second component contains the architecture-dependent communication resource scheduling and routing routines. This division permits special-purpose routines optimized for a particular architecture to be employed within the second component. A specific interface is defined at the boundary, which allows the topology dependent sections to be interchangeable, as illustrated in figure 3-4. This design makes the scheduler well-suited for an object-oriented programming environment. Each "architecture object" contains its own communication resource scheduling and routing routines.

3.2. DYNAMIC LEVELS

We first derive dynamic levels for the homogeneous processor case. At each scheduling step, a list scheduling algorithm performs two tasks. It selects the next node to

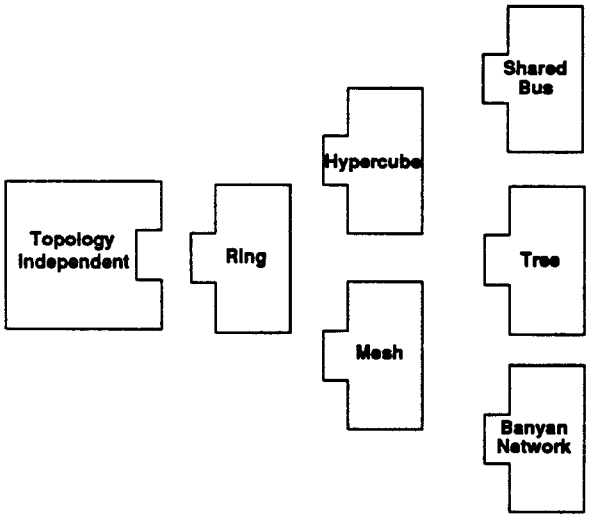


Figure 3-4. The processor topology dependent portions are interchangeable

schedule, and chooses the processor to execute the node on. The HLFET algorithm makes these selections independently at each step, causing poor performance in the presence of IPC. Whereas the static levels used in HLFET are fixed, we introduce a new quantity whose value changes throughout the scheduling process. This **dynamic level**, denoted $DL(N_i, P_j, \Sigma(t))$, reflects how well node N_i and processor P_j are matched at state $\Sigma(t)$, where $\Sigma(t)$ encompasses both the state of the processing resources (previously scheduled nodes), and the state of the communication resources (previously scheduled data transfers) at global time t. We first define $DA(N_i, P_j, \Sigma(t))$ to be the earliest time that all *data* required by node N_i is *available* at processor P_j at state $\Sigma(t)$. This quantity, calculated within the topology-dependent section of the scheduler, represents the earliest time at which all data transfers to node N_i from its immediate predecessors can be **guaranteed** to be completed with all communication resources reserved in advance. The dynamic level is now expressed as

$$DL(N_i, P_j, \Sigma(t)) = SL(N_i) - \max[t, DA(N_i, P_j, \Sigma(t))]. \qquad (3.3)$$

At each step, the ready node and available processor which maximize this expression are chosen for scheduling.

The dynamic level has a straightforward interpretation. The maximization term represents the earliest time that node N_i can start execution on processor P_j, because the node cannot start execution until the current time, and it cannot be invoked until all the data from its predecessors has been received. So the dynamic level $DL(N_i, P_j, \Sigma(t))$ is the difference between the static level of node N_i and the earliest time the node can start execution on processor P_j. This expression, which simultaneously incorporates both execution and communication time aspects, is intuitively appealing. When considering prospective nodes for scheduling, a node with large static level is desirable because it indicates a high priority for execution. When comparing candidate processors for assignment, a later starting time is undesirable because it indicates either a busy processor or a long interval required for interprocessor communication. Processors which incur a large amount of IPC are penalized by a later starting time, which lessens the dynamic level. The algorithm evaluates dynamic levels (which may be negative) over all combinations of ready nodes and available processors to find the best node-processor match for scheduling at the current state. We designate this approach the Highest *Dynamic* Levels First with Estimated Times (HDLFET) algorithm. This method uses the classical list scheduling formulation with dynamic levels.

To gain an indication of the performance improvement obtainable through dynamic levels, we used the random graph generator given in Appendix II to construct task graphs containing between 50 and 250 nodes, and scheduled these graphs onto a 16-processor mesh. Node execution times and nearest-neighbor communication times

were chosen randomly from the same uniform distribution. We investigated several options for node and processor selection. The first method initially selects the available processor with smallest index and then chooses the ready node which maximizes the dynamic level with this processor. The second method initially selects the ready node with highest static level and chooses the available processor maximizing the dynamic level with this node. The third method examines all possible combinations of ready nodes and available processors and chooses the node-processor pair maximizing the dynamic level. In each case, we compared the percentage improvement in speedup obtained over the HLFET approach using independent node and processor selection. Communication costs were included in all cases.

The curves in figure 3-5 show speedup improvement compared against graph parallelism, where graph parallelism is measured as

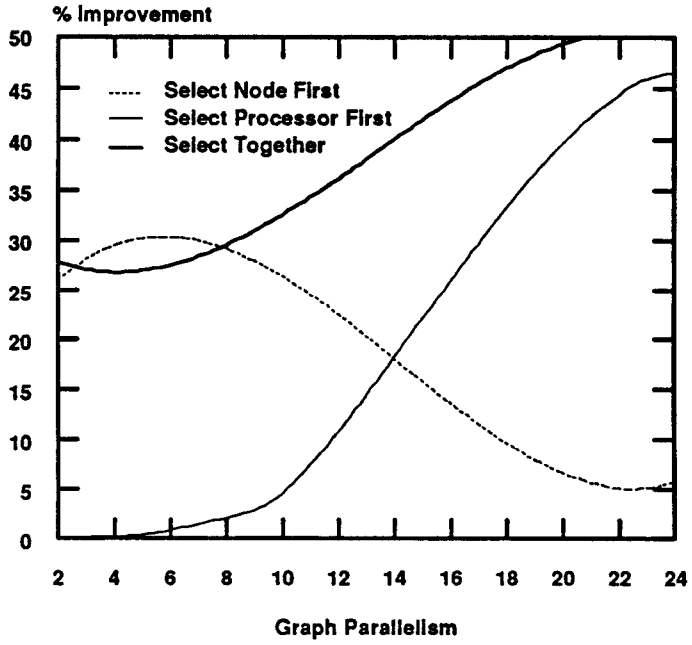


Figure 3-5. Percent improvement in speedup of HDLFET over HLFET

$$\left\lceil \frac{1}{\max_j SL(N_j)} \sum_{i=1}^{n} E(N_i) \right\rceil \tag{3.4}$$

This is a lower bound on the number of processors required to execute the graph in time bounded by the critical path (the longest path from any initial node to any terminal node) if IPC costs are not included. Each point in figure 3-5 represents the average percentage improvement in speedup over HLFET obtained for all graphs with the specified amount of graph parallelism.

The initial processor selection technique exhibits little improvement when the amount of graph parallelism is small compared to the number of processors, but starts to gain improvement rapidly as the amount of parallelism increases. This phenomenon can be simply explained by examining the number of nodes ready for execution at each step. When the number of processors far exceeds the graph parallelism, there are very few ready nodes at each scheduling step, in many cases only a single node. By initially selecting the processor, the algorithm forces this single ready node to be executed on the processor, and therefore performs the same steps as the HLFET scheme using independent node and processor selection. As the amount of graph parallelism increases, the number of ready nodes at each step increases, permitting a better match between nodes and processors.

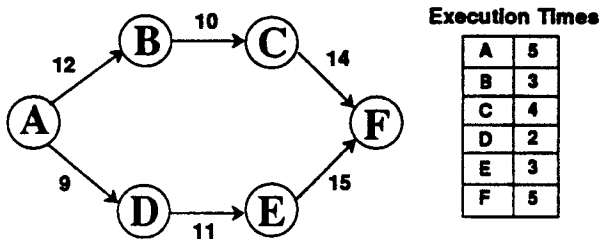
Conversely, the initial node selection technique exhibits large improvement when the number of processors exceeds the amount of graph parallelism, but tapers off as the parallelism is increased. This can be accounted for by considering the available processors at each step. When the number of processors far exceeds the graph parallelism, there is little contention for processors. Each node is able to select its "preferred" processor which incurs little communication overhead out of the available processor list. As the amount of parallelism increases, parallel paths in the graph begin to share

processing resources, and scheduling steps may have multiple ready nodes desiring a common processor. As processors are successively removed from the available processor list, the node with highest static level is often forced to be executed on one of the remaining available processors for which excessive IPC is incurred. The performance degradation is exacerbated as the amount of parallelism is further increased.

Selecting the highest dynamic level ready-node, available-processor pair out of all combinations retains good improvement throughout the entire range, increasing slightly as the amount of graph parallelism increases. This method does not incur the increased-parallelism performance degradation exhibited by the initial node selection method because it has more flexibility in choosing nodes. At any step, a node which does not have the highest static level may form the best match (in dynamic level) with an available processor, and therefore be selected for scheduling. Instead of forcing execution on unsuitable processors, this strategy allows a ready node to wait until a successive time step when its desired processor is again available. This method demonstrates superior performance over all the other techniques, attaining speedup improvements of over 50% in comparison with the HLFET algorithm. However, this performance is attained at the price of added computational complexity.

3.2.1. Processor Selection Revision

While the use of dynamic levels significantly improves performance, the HDLFET algorithm still exhibits the list scheduling deficiency of being unable to idle "available" processors. Consider the graph shown in figure 3-6, and for simplicity, assume a two-processor system with communication model C = D, so that the number of cycles needed for IPC equals the number of data units. The optimal schedule executes every node on a single processor while idling the other processor completely, a solution



Execution Times

A	5
B	3
C	4
D	2
E	3
F	5

Figure 3-6. A fine-grained precedence graph

which is unattainable under the current list scheduling methodology. This inability to idle processors is an inherent flaw in the algorithm, which requires that all available processors be assigned nodes for execution before the global clock can be updated to replenish the supply. This philosophy, which attempts to exploit as much task parallelism as possible, is no longer valid in the presence of IPC, and causes poor scheduling performance in a mixed-grain environment.

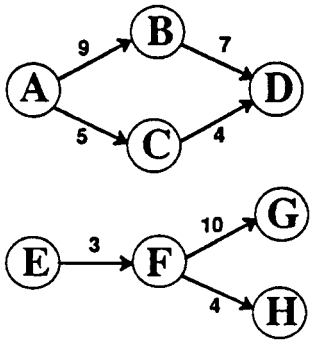
To remedy this difficulty, we alter the fundamental operation of the algorithm by removing the global timeclock used to update the current time at each scheduling step. There is no difficulty with causality because scheduling is performed at compile-time, not at run-time. Since processors are no longer classified as being "busy" or "available", all processors can be considered candidates for scheduling at each step. This allows the same processor to be chosen in consecutive scheduling steps, and permits some processors to have nodes scheduled far in advance of other processors. This revision necessitates a few changes in the dynamic level. Since the notion of a global time no longer exists, the state of the processing and communication resources $\Sigma(t)$ becomes Σ. We let $TF(P_j, \Sigma)$ represent the time that the last node assigned to the jth processor finishes execution and redefine the dynamic level as

$$DL(N_i, P_j, \Sigma) = SL(N_i) - \max\left[DA(N_i, P_j, \Sigma), TF(P_j, \Sigma)\right]. \qquad (3.5)$$

The revised algorithm, which operates without a global clock and uses the dynamic

level shown in (3.5) is the Dynamic Level Scheduling (DLS) algorithm.

To illustrate the effect of the global clock removal more clearly, we contrast the scheduling steps taken by the HDLFET algorithm and the DLS algorithm using the APEG shown in figure 3-7. This example is scheduled onto a two processor system where the processors are assumed to be interconnected by a full-duplex link. Both scheduling methods will select a node and processor simultaneously using dynamic levels at each scheduling step. As shown in figure 3-8, the scheduling steps taken by the two approaches start to diverge after nodes A and B have been scheduled on P0 and nodes E and F have been scheduled on P1. At this point, the HDLFET algorithm's global clock is at time 5, when P1 is the only processor available for scheduling, and nodes C, G, and H are ready for execution. After evaluating dynamic levels for these three nodes with P1, the algorithm finds that node C forms the best match with P1. It therefore schedules node C on P1, schedules communication A to C on the link from P0 to P1 in the interval [3,8], and updates the global clock to time 6, when P0 becomes available. After evaluating dynamic levels for nodes G and H with P0, the algorithm schedules node H on P0 and communication F to H on the link from P1 to P0 in the interval [5,9]. The global clock is updated to time 11, freeing P1 for



Execution Times

A	3
B	3
C	3
D	7
E	2
F	3
G	5
H	6

Static Levels

A	13
B	10
C	10
D	7
E	11
F	9
G	5
H	6

Figure 3-7. An example APEG

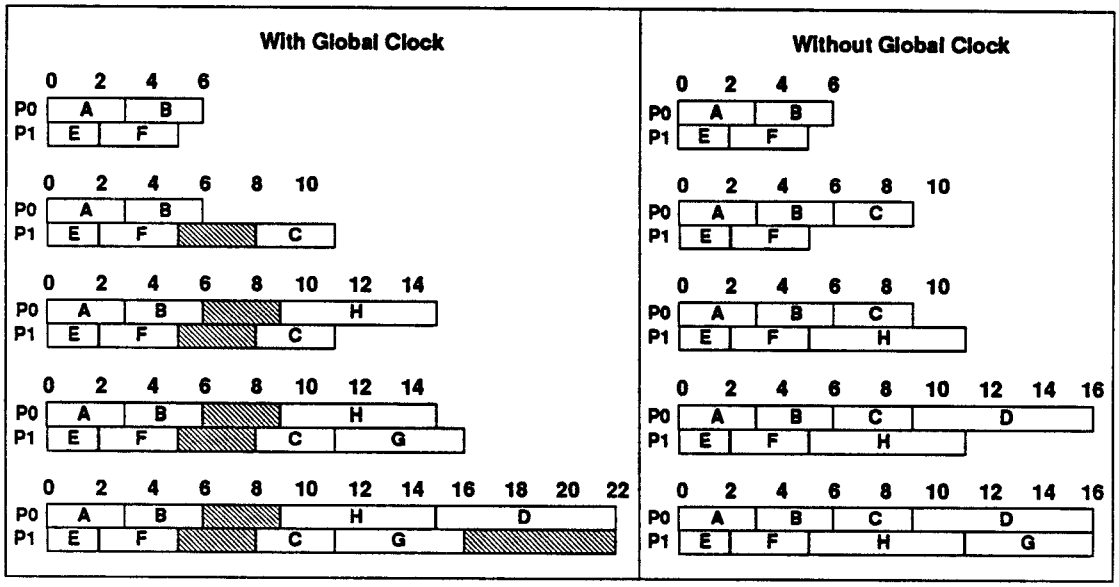


Figure 3-8. Scheduling progression with and without the global clock

the scheduling of either node G or D. Although node D has higher static level than node G, it has a lower dynamic level with P1 at the current state. The input data from node B can not be guaranteed to be available until time 15, because the previous communication from node A to node C has reserved the link from P0 to P1 until time 8. Node G is therefore scheduled on P1. Finally, node D is scheduled on P0, with communication from C to D being scheduled in the interval [11,15] on the link from P1 to P0. This approach yields a makespan of 22 time units.

In contrast, after nodes A, B, E, and F have been scheduled, the DLS algorithm evaluates dynamic levels for nodes C, G, and H with both processors P0 and P1, and discovers that node C and P0 form the best match. Node C is subsequently scheduled on P0 and node D is immediately released into the list of ready nodes even though it can not be executed until time 9. Next, after evaluating dynamic levels for each of nodes D, G, and H with P0 and P1, the DLS approach selects and schedules node H on P1. The final two steps schedule node D on P0 and node G on P1, which results in an

optimal schedule with makespan 16. Since additional processors are incorporated only as they are needed, the DLS approach constructs schedules which exhibit a natural "clustering" of nodes which communicate heavily, without sacrificing efficient use of the communication resources.

The performance curves in figure 3-9 show the percentage improvement in speedup obtained over HLFET for both the HDLFET and DLS algorithms. While the two methods exhibit comparable performance at modest levels of graph parallelism, the DLS approach exhibits sharply increasing performance as the amount of parallelism increases, displaying average speedup improvements exceeding 75% over the HLFET algorithm. This illustrates its enhanced ability to assign each node its "preferred" processor. Notice that the DLS algorithm using initial node selection does not exhibit the performance degradation displayed by the HDLFET approach as the graph parallelism increases, but rather exhibits nearly equal performance as simultaneous node and

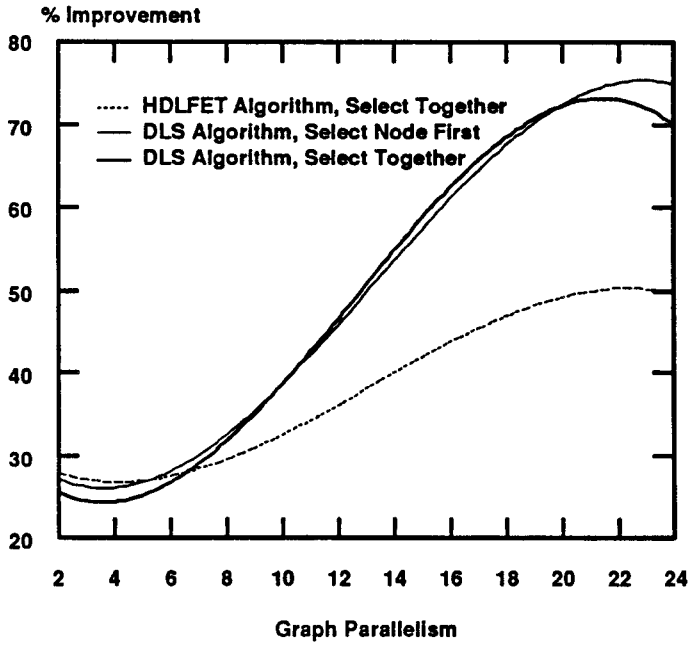


Figure 3-9. Performance improvement in speedup over HLFET

processor selection. A likely explanation is that the greater freedom in processor selection allows the same scheduling steps to occur in these two cases, but in a different order. In practice, we found the speedup improvement to be even higher. When scheduling signal processing algorithms onto a four-processor shared-memory multiprocessor, the DLS algorithm often exhibits speedup improvements exceeding 100% when the parallelism greatly exceeds the number of processors. The strength of the DLS algorithm lies in its ability to effectively overlap communication with computation. Consider the example shown in figure 3-10, which will be scheduled onto a 4-processor single shared-bus architecture. If we assume that the time required to communicate each data unit is five time units, the DLS algorithm produces the optimal schedule shown at the top of figure 3-11. The communication schedule, shown at the bottom of figure 3-11, removes the possibility of contention for the bus.

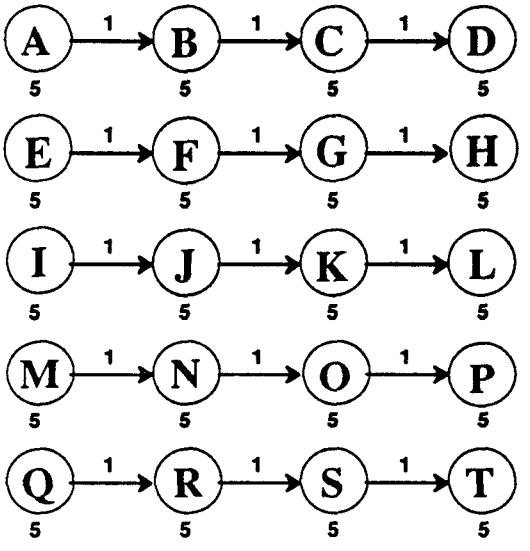


Figure 3-10. A graph example

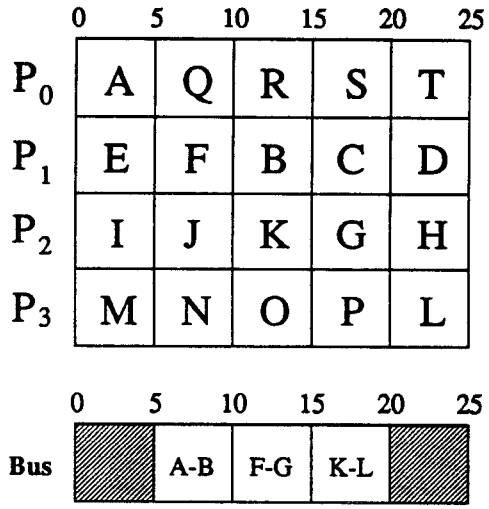


Figure 3-11. Computation and communication schedules

3.2.2. Algorithm Streamlining

The dynamic level scheduling algorithm can be streamlined to provide faster execution without significant degradation in performance.

Initial Node Selection

In addition to the performance gain attained through removal of the global clock, the ability to gain analogous performance using initial node selection yields a big savings in execution time because examination of all node-processor combinations is no longer necessary. After investigating several methods for initially selecting a single ready node, we narrowed the choices down to the following two possibilities:

$$\max_i \{SL_i + C_{adj}[\max_k (D_{ki})]\} \tag{3.6}$$

$$\max_i \{SL_i + \sum_k C_{adj}(D_{ki})\} \tag{3.7}$$

D_{ki} represents the number of data units passed from node k to node i, while $C_{adj}(D)$ denotes the time needed to communicate D data units between adjacent processors.

The first method selects the ready node which has the largest sum of the static level and the communication time to transfer the largest number of data units passed into the node from any immediate predecessor. The second method selects the ready node which maximizes the sum of the static level and the sum of the adjacent processor communication times for *each* data transfer from the ready node's predecessors. Equation (3.6) performs slightly better than equation (3.7). The maximum communication time is more important than the sum of the communication times, presumably because it may be possible to avoid only one of the IPC costs if the predecessors are located on different processors. In addition, the DLS algorithm is particularly effective at overlapping communication with execution of other nodes which can be immediately invoked, thereby obtaining the other communications at virtually no cost.

Limiting Processor Selection

To further decrease the time required for scheduling, we can reduce the number of processors for which dynamic levels are evaluated with the given node. For many multicomputer networks (e.g. mesh, hypercube), a scheme based on a center of mass principle proves to be effective. This method identifies the processor locations for each predecessor of the candidate node to be scheduled, and obtains the number of data units to be communicated in each transfer. Using the predecessor processor positions in an appropriate coordinate system, and the number of data units as a weighting function, the technique calculates the center of mass of the data and rounds it to the nearest processor location. It then limits candidate processors to those within a fixed radius of this center of mass processor, with a few processors located outside this range included to promote spreading of the load. Consider the example shown in

figure 3-12, in which the partial graph shown at the left is being scheduled onto the 16 processor mesh on the right of the figure. Nodes A, B, and C have already been scheduled, with their processor locations indicated on the mesh diagram, and the problem is to find candidate processors for node D. We arbitrarily designate the processor at the lower left corner of the mesh to be the origin. When considering the predecessors of node D, we find that there are 10 data units coming from node A on processor (1,1), 2 data units coming from node B on processor (0,3), and 6 data units coming from node C on processor (1,3). Calculating the center of mass of the data, we find

$$(Xcm\ ,\ Ycm) = \left[\frac{10(1)+2(0)+6(1)}{10+2+6}\ ,\ \frac{10(1)+2(3)+6(3)}{10+2+6} \right] = (\frac{16}{18}\ ,\ \frac{34}{18}) \approx (1\ ,\ 2)(3.8)$$

All processors within two hops of processor (1,2) are then considered candidates for scheduling node D. Intuitively, the center of mass calculation compels each predecessor processor to pull the candidate node toward it, with an attractive force proportional to the amount of data communicated. While this technique is not intended as a panacea for all architectures, it is reasonable to assume that similar techniques to limit pro-

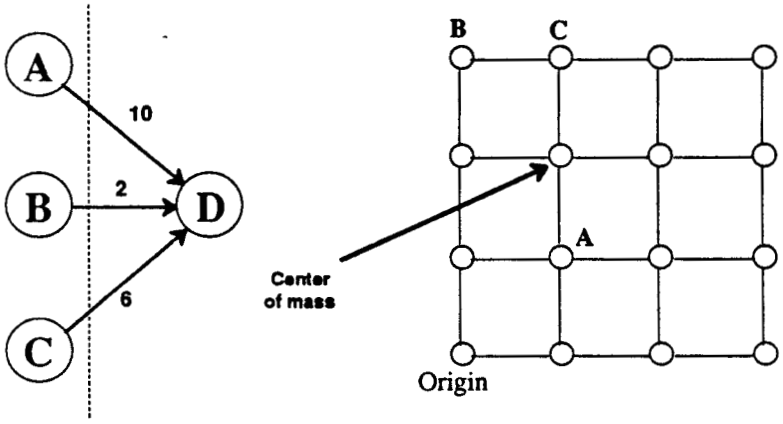


Figure 3-12. Center of mass example

cessor selection can be developed for each topology with minor performance penalty.

The diagram in figure 3-13 illustrates the operations performed by the streamlined algorithm at each scheduling step, where each operation is classified as residing in the fixed or topology-dependent component. The fixed component begins by selecting a single node according to method 1 in the previous section. It passes the chosen node to the topology dependent section, which returns a list of candidate processors for scheduling this node. The topology independent section evaluates dynamic levels for each processor using the tentative routing and resource reservation routines contained in the topology dependent section. After obtaining the processor which maximizes the dynamic level, the algorithm schedules the node, invoking the permanent routing and resource reservation routines. The simplicity of this scheme permits execution speeds suitable for a prototyping environment. The current lisp implementation schedules a 200 node graph onto 16 processors in less than a minute on a Sun-3/60 workstation.

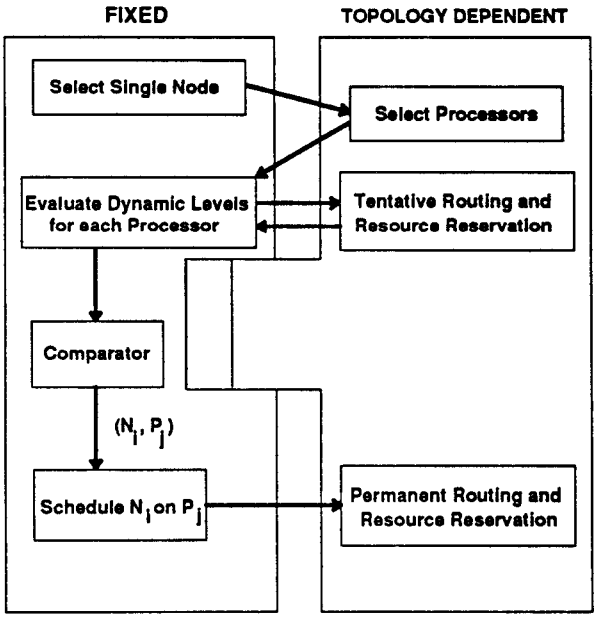


Figure 3-13. The algorithm specification

3.3. HETEROGENEOUS PROCESSOR EXTENSION

The presence of heterogeneous processors further complicates the efficacious matching of nodes with processors because it may be advantageous to separate nodes which lie on an entirely sequential path onto different processors to exploit specialized hardware features. In addition to the tradeoff between exploitation of parallelism and IPC cost, this environment introduces a new tradeoff between varying processor computation speeds and IPC cost.

The dynamic level expression requires several modifications to accommodate a heterogeneous processing environment. Since the static level $SL(N_i)$ loses its meaning when execution times vary between processors, we define the adjusted median execution time of node N_i, denoted $E^*(N_i)$, to be the median of the execution times of node N_i over all the processors, with the stipulation that the largest finite execution time is substituted if the median itself is infinite. $SL^*(N_i)$ correspondingly denotes the static level of node N_i, calculated using these adjusted median execution times. To account for the varying processing speeds, we define

$$\Delta(N_i, P_j) = E^*(N_i) - E(N_i, P_j). \tag{3.9}$$

A positive $\Delta(N_i, P_j)$ indicates that processor P_j executes node N_i more rapidly than most processors, while a negative $\Delta(N_i, P_j)$ indicates the reverse. By incorporating these terms, our first extended dynamic level $DL_1(N_i, P_j, \Sigma)$ quickly follows:

$$DL_1(N_i, P_j, \Sigma) = SL^*(N_i) - \max\left[DA(N_i, P_j, \Sigma), TF(P_j, \Sigma)\right] + \Delta(N_i, P_j) \tag{3.10}$$

The static level component indicates the importance of the node in the precedence hierarchy, giving higher priority to nodes located further from the terminus (in adjusted median execution time). The maximization term reflects the availability of

the processing and communication resources, penalizing node-processor pairs which incur large communication costs. The delta term accounts for the processor speed differences, adding priority to processors which execute the node quickly, and subtracting priority from processors which execute it slowly. Note that when all processors are identical, $\Delta(N_i, P_j) = 0$ and $SL^*(N_i) = SL(N_i)$, causing this formula to converge to the homogeneous processor dynamic-level expression.

The chart in table 3-1 shows the mean percentage speedup improvement obtained from using extended dynamic level $DL_1(N_i, P_j, \Sigma)$ for node and processor selection over the classical HLFET method using static levels. We obtained these values from scheduling 100 randomly generated graphs onto a 16 processor mesh, where the graphs were constructed using the random graph generator in Appendix II. The node execution times and the amount of data transferred between nodes were uniformly distributed over [5,15], and the speed ratio between fastest and slowest processors was 16. To simulate the effects of special hardware features, we constrained several randomly chosen nodes to be executed on a small subset of processors. We investigated three cases: when a single ready node maximizing equation (3.6) was considered for scheduling at each step, when the three ready nodes with highest values of equation (3.6) were considered at each

Percentage Improvement in Speedup Over HLFET					
Algorithm	Mean	Std. Deviation	Minimum	Median	Maximum
DL1, 1 Node	125.33%	54.02%	59.24%	111.35%	340.09%
DL1, 3 Nodes	127.71%	48.08%	66.96%	113.04%	293.16%
DL1, All Nodes	132.58%	50.81%	70.07%	118.64%	355.45%

Table 3-1. Speedup improvements in using $DL_1(N_i, P_j, \Sigma)$ for node and processor selection

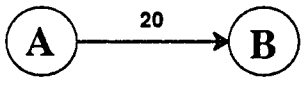
step. The performance increases slightly with the number of nodes considered at each step. As before, these experiments using randomly generated graphs should be considered indicative, rather than evidential, of performance with practical applications.

3.3.1. Descendant Consideration

This natural dynamic level extension overlooks several subtle effects introduced when different types of processors are present. Although $DL_1(N_i, P_j, \Sigma)$ indicates how well node N_i is matched with processor P_j, it fails to consider how well the descendants of N_i are matched with P_j. Consider the simple example shown in figure 3-14, in which node A passes 20 data units to node B. Since P_0 executes node A faster than P_1,

$$DL_1(A, P_0, \Sigma) > DL_1(A, P_1, \Sigma), \tag{3.11}$$

causing node A to be scheduled on P_0. However, because node B cannot be executed on P_0, the scheduling of node A on P_0 forces an interprocessor communication of 20 data units, which we assume for simplicity to take 20 time units as shown in the top diagram of figure 3-15. If descendant B had been considered during the scheduling of node A, both nodes would have been scheduled on P_1, as shown in the bottom diagram in figure 3-15. This example illustrates that it is not always advantageous to schedule a node on the processor which executes it most rapidly, due to this descendant effect.



Execution Times

	P0	P1
A	5	8
B	∞	7

Figure 3-14. A descendant consideration example

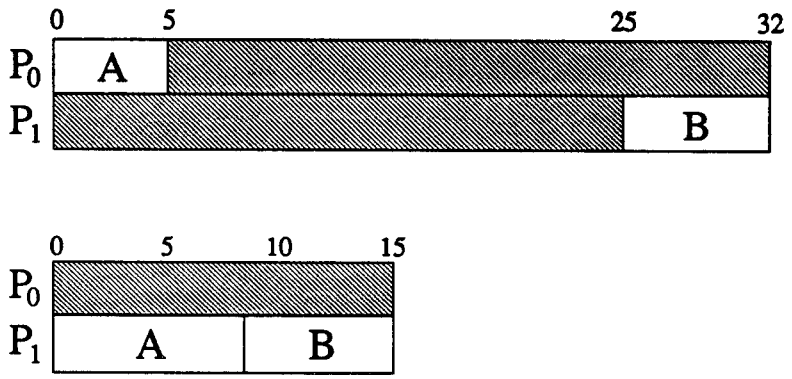


Figure 3-15. Schedules for possible placements of nodes A and B

For each node N_i, we consider the descendant to which N_i passes the most data, designating this node as $D(N_i)$, and the amount of data passed between them as $d(N_i, D(N_i))$. Recalling that $E[D(N_i), P_j]$ indicates how quickly node $D(N_i)$ executes on processor P_j, we define another term $F[N_i, D(N_i), P_j]$ to indicate how quickly $D(N_i)$ can be completed on any other processor if N_i is executed on P_j:

$$F[N_i, D(N_i), P_j] = C_{adj}[d(N_i, D(N_i))] + \min_{k \neq j} E[D(N_i), P_k]. \quad (3.12)$$

This is a lower bound on the time needed to finish execution of $D(N_i)$ on any processor other than P_j after N_i finishes execution on P_j. We then define a descendant consideration term as

$$DC(N_i, P_j, \Sigma) = E^*[D(N_i)] - \min\{E[D(N_i), P_j], F[N_i, D(N_i), P_j]\}, \quad (3.13)$$

and update the dynamic level expression to

$$DL_2(N_i, P_j, \Sigma) = DL_1(N_i, P_j, \Sigma) + DC(N_i, P_j, \Sigma). \quad (3.14)$$

The descendant consideration term is the difference between the adjusted median execution time of $D(N_i)$ and a lower bound on the time required to finish execution of $D(N_i)$ after N_i finishes execution on P_j. This indicates how well the "most expensive" descendant of node N_i matches up with processor P_j, if N_i is scheduled on P_j. If P_j executes $D(N_i)$ more rapidly than most processors, $DC(N_i, P_j, \Sigma)$ becomes

$E^*[D(N_i)] - E[D(N_i), P_j]$, causing an increase in $DL_2(N_i, P_j, \Sigma)$. Conversely, if P_j executes $D(N_i)$ very slowly compared with most processors, $DC(N_i, P_j, \Sigma)$ becomes negative, causing a decrease in dynamic level. If $E[D(N_i), P_j]$ is equal to the median execution time of $D(N_i)$, the descendant term is appropriately zero. The term is also zero when all processors are homogeneous, reflecting the fact that all processors can execute the descendant equally well.

The percentage improvement in speedup over HLFET in using $DL_2(N_i, P_j, \Sigma)$ for node and processor selection is shown in table 3-2 for the same three cases. When using DL2, the mean and median performance improve in every case (1 node, 3 nodes, All nodes) over the corresponding measurements for DL1, reflecting the importance of descendant consideration.

Note that if N_i is scheduled on processor P_j, its "most expensive" descendant $D(N_i)$ is not necessarily scheduled on the same processor. One reason for this is that $D(N_i)$ may itself have a descendant which executes slowly on P_j. Consider the graph shown in figure 3-16. Through evaluation of $DL_2(A, P_j, \Sigma)$ for each processor, the procedure determines that node A prefers P_0. The descendant consideration term is zero in this case, because node B executes equally quickly on every processor. However, the immediate scheduling of node B onto P_0 leads to an inefficiency because node B's

Percentage Improvement in Speedup Over HLFET					
Algorithm	Mean	Std. Deviation	Minimum	Median	Maximum
DL2, 1 Node	129.29%	51.90%	65.52%	112.04%	338.10%
DL2, 3 Nodes	135.45%	54.97%	62.91%	121.05%	327.98%
DL2, All Nodes	138.97%	53.88%	79.44%	123.32%	333.96%

Table 3-2. Speedup improvements in using $DL_2(N_i, P_j, \Sigma)$ for node and processor selection

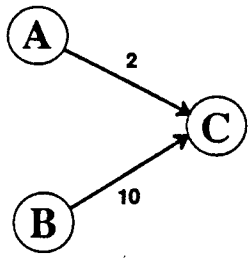
Execution Times

	P0	P1	P2
A	3	6	6
B	5	5	5
C	8	4	4

Figure 3-16. Another descendant consideration example

descendant, node C, executes slowly on P_0. While node C could be shifted onto P_1 or P_2, this incurs the communication of four data units between B and C. It is more effective to postpone the scheduling of B until its dynamic level $DL_2(B, P_j, \Sigma)$ is calculated, because after incorporating the descendant consideration term for C, the procedure will correctly shift node B onto either P_1 or P_2.

The example shown in figure 3-17 illustrates another factor to account for when performing descendant consideration. When node A is scheduled, it will use node C in its descendant consideration term by default. However, since node C receives more data from node B, the processor location of node B will likely play a more important role in placement of node C than the location of node A. We therefore designate node B as the **data-dominating ancestor** of node C, denoting this as $B \rightarrow C$. If a node is not the data-dominating ancestor of its most expensive descendant, it may not be necessary to consider $D(N_i)$ when scheduling N_i. To investigate this factor, we intro-



Execution Times

	P0	P1	P2
A	3	6	6
B	5	5	5
C	8	4	4

Figure 3-17. Node B is the data-dominating ancestor of node C

duce an additional dynamic level $DL_3(N_i, P_j, \Sigma)$ defined as

$$DL_3(N_i, P_j, \Sigma) = \begin{cases} DL_2(N_i, P_j, \Sigma) & \text{if } N_i \rightarrow D(N_i) \\ DL_1(N_i, P_j, \Sigma) & \textit{else} \end{cases} \quad (3.15)$$

The performance results using $DL_3(N_i, P_j, \Sigma)$ are displayed in table 3-3. After comparing tables 2 and 3, the performance gain obtained using DL3 appears to be insignificant.

3.3.2. Resource Scarcity

In addition to the descendant consideration effect, a heterogeneous processing environment also introduces a cost associated with a node **not** being executed on a certain processor. This phenomenon occurs because effective processing resources may be scarcer for some nodes than others at certain states. To illustrate this cost, consider a situation in which an APEG is being scheduled onto a five processor system and the partial schedule at the current state Σ' is shown in figure 3-18. There are two nodes left to be scheduled, M and N, whose execution times on each of the processors are shown in table 3-4 below. After evaluating dynamic levels for M and N, the algorithm finds that both nodes would like to be scheduled on P_0 at the current state. If node M is initially scheduled on P_0 in the time interval [8,12], the earliest time that node N can finish execution is at time 16 on P_0, as shown in figure 3-19.

Percentage Improvement in Speedup Over HLFET					
Algorithm	Mean	Std. Deviation	Minimum	Median	Maximum
DL3, 1 Node	130.38%	57.61%	63.40%	114.83%	383.42%
DL3, 3 Nodes	133.62%	50.15%	70.78%	121.21%	340.09%
DL3, All Nodes	139.37%	58.64%	76.02%	124.18%	379.17%

Table 3-3. Speedup improvements in using $DL_3(N_i, P_j, \Sigma)$ for node and processor selection

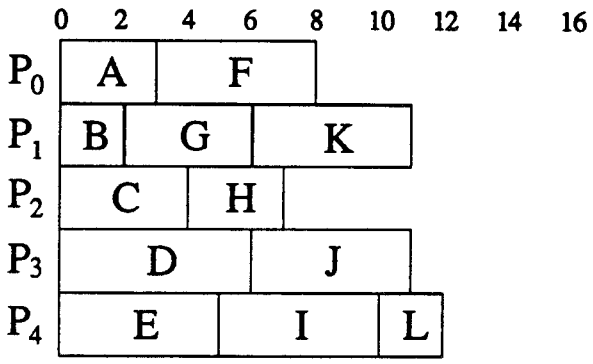


Figure 3-18. The schedule at state Σ'

Node	P0	P1	P2	P3	P4
M	4	8	5	8	8
N	4	8	∞	8	8

Table 3-4. Execution times for nodes M and N

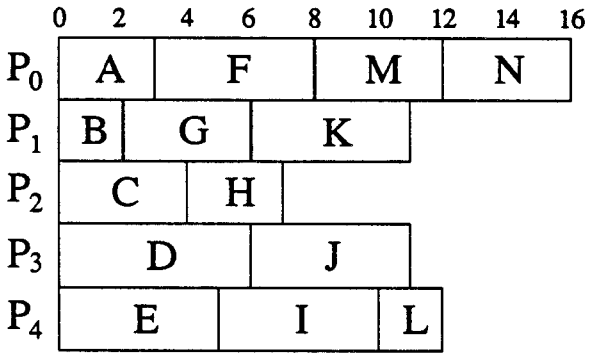


Figure 3-19. Initially scheduling M on P_0 gives makespan 16

However, if node N is initially scheduled on P_0, node M can be scheduled on P_2, leading to the schedule with makespan 12 shown in figure 3-20. There is a negligible cost if node M is not assigned to P_0 at state Σ', because P_2 can execute the node almost as rapidly. Conversely, there is a large cost if node N is not assigned to P_0 at this state because P_0 is the only processor which executes node N so quickly.

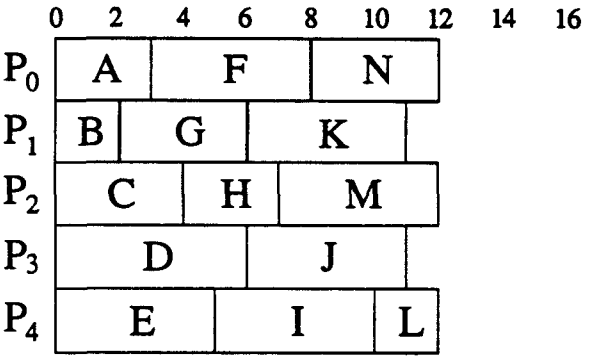


Figure 3-20. Initially scheduling N on P_0 gives makespan 12

This example illustrates that while $\Delta(M, P_0)$ and $\Delta(N, P_0)$ account for processor speed variations, they fail to consider how important it is for nodes M and N to obtain processor P_0 at the current state. In this case, node N should have greater claim to P_0, because its cost in not obtaining it is much higher. This cost in depriving a node of a certain processor stems from the relative scarcity of the processing resources for each node at the current state. Its existence indicates that a more careful apportionment of nodes to processors is necessary in an inhomogeneous environment.

This situation arises at scheduling steps which have 2 or more ready nodes simultaneously desiring the same processor. To characterize this resource scarcity cost, we first define

$$DL(N_i, P_{j^*}, \Sigma) = \max_j DL(N_i, P_j, \Sigma), \tag{3.16}$$

which designates P_{j^*} as a preferred processor of node N_i. That is, j^* is the index of a processor which maximizes the dynamic level with node N_i at state Σ. We then define the cost in not scheduling node N_i on its preferred processor at the current state as:

$$C(N_i, P, \Sigma) = DL(N_i, P_{j^*}, \Sigma) - \max_{k \neq j^*} DL(N_i, P_k, \Sigma). \tag{3.17}$$

$C(N_i, P, \Sigma)$ is the difference between the highest dynamic level and the second

highest dynamic level for node N_i over all processors at the current state. In other words, if a node other than N_i is scheduled on processor P_{j^*} at state Σ, the cost is the amount of dynamic level which is lost in having to settle for the "next best" processor. Notice that this cost is only nonzero when there is a single preferred processor P_{j^*} such that for all $k \neq j^*$

$$DL(N_i, P_{j^*}, \Sigma) > DL(N_i, P_k, \Sigma). \tag{3.18}$$

If this condition is not satisfied, the cost is zero because even if another node is scheduled onto one of N_i's preferred processors, N_i will still have at least one processor with which it can attain the same dynamic level (assuming communication resources are still available). $C(N_i, P, \Sigma)$ is a function of the entire set of processors P, because calculating this term requires evaluation of dynamic levels for node N_i over all processors at the current state. We now define a "generalized" dynamic level as follows:

$$GDL(N_i, P, \Sigma) = DL(N_i, P_{j^*}, \Sigma) + C(N_i, P, \Sigma). \tag{3.19}$$

We now consider several ready node candidates simultaneously, and the node which maximizes $GDL(N_i, P, \Sigma)$ is scheduled on its preferred processor. The scheduling results obtained using generalized dynamic levels are shown in table 3-5. GDL1 refers to the generalized dynamic level calculated using DL_1 as its base level. Similarly, GDL2 uses DL_2 as its base level, and GDL3 uses DL_3 as its base level. As indicated by these results, the generalized dynamic levels provide a significant performance improvement, especially when they are evaluated over all ready nodes. However, the algorithm which considers all ready nodes at each step has complexity $O[n^3 p\ f(p)]$, where n is the number of nodes, p is the number of processors, and the function used to route a path between two given processors on the targeted architec-

Percentage Improvement in Speedup Over HLFET					
Algorithm	Mean	Std. Deviation	Minimum	Median	Maximum
GDL1, 3 Nodes	133.23%	49.35%	74.83%	119.12%	342.18%
GDL1, All Nodes	145.82%	55.50%	60.26%	136.42%	384.21%
GDL2, 3 Nodes	138.18%	57.27%	72.26%	116.81%	364.65%
GDL2, All Nodes	145.92%	58.82%	72.41%	129.55%	389.36%
GDL3, 3 Nodes	137.30%	53.48%	78.26%	122.66%	357.35%
GDL3, All Nodes	149.33%	60.41%	82.64%	132.84%	400.00%

Table 3-5. Speedup improvements in using generalized dynamic levels

ture is $O[f(p)]$. Since this algorithm is too time-consuming for use in an interactive environment, we recommend the method which chooses three candidate nodes at each step and uses GDL2 for node and processor selection. The complexity of this scheme is $O[n^3 + n^2 p\ f(p)]$, which can be trimmed further if the generalized dynamic levels are only evaluated over a chosen subset of processors.

3.4. ROUTING ALGORITHMS

The scheduling of communications invoked by the dynamic-level scheduling approach necessitates the use of routing algorithms to choose a path for data transfer between source and destination processors. The DLS algorithm maintains a data structure for each communication link between processors, indicating the time intervals the link is used. By checking these link records for each considered routing path, the scheduler can find the earliest time that required data can be made available at its destination processor. The routing functions lie within the topology-dependent portion of the scheduler, permitting a specialized routing routine to be used for each different processor architecture. Since the routing function lies in the "critical path" of the algorithm (it is used each time a dynamic level is evaluated), the routing pro-

cedures must be simple and fast. We restrict our attention to shortest path algorithms, so that the chosen path between two processors always contains the minimum number of processor hops. Example routing functions are given below for the mesh and hypercube architectures.

For two mesh processors which are separated by v vertical hops and h horizontal hops, there are $\frac{(v+h)!}{v!h!}$ possible shortest paths between them. Since it is inefficient to check each possible path, we investigated several possible routing algorithms for this topology and found that a three-line routing procedure proves effective. This technique only looks at paths which can be constructed using three or fewer straight lines. The diagram in figure 3-21 shows the six possible routes permitted by the three-line method between diagonally opposed processors in a 16-processor mesh. In addition to being simple and quick, this method actually permits more

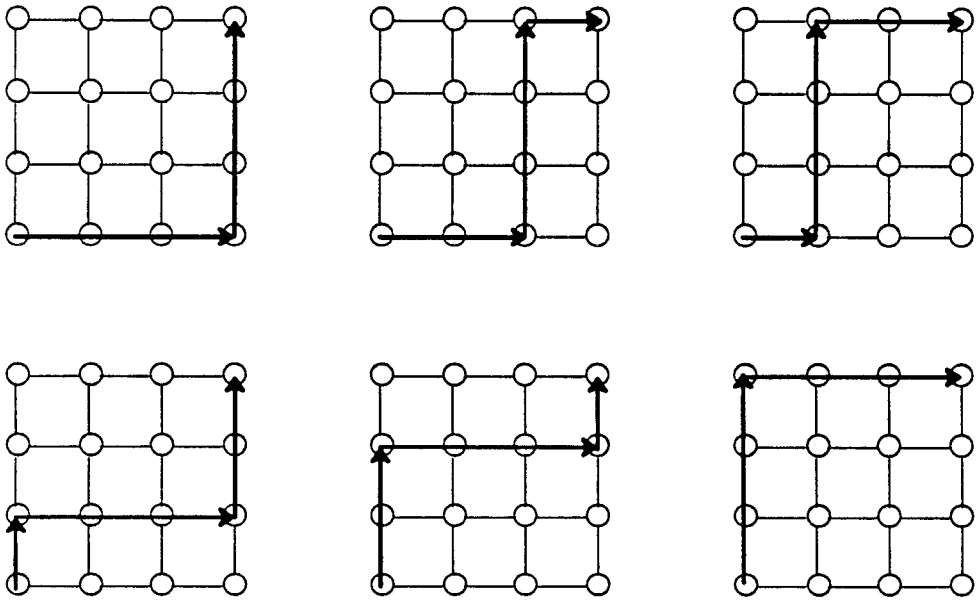


Figure 3-21. Six possible routes between diagonally opposed processors

communications between different pairs of processors to take place simultaneously than more complicated routing methods. This is due to the difficulty in fitting together paths which have many twists and turns on the mesh simultaneously.

The processors in a hypercube can be assigned binary addresses such that adjacent processors have unity hamming distance; that is, the addresses of nearest-neighbor processors differ in exactly one bit position. An example address assignment with this property is shown for an eight-processor hypercube in figure 3-22. For two processors which differ in b bit positions, there are exactly b! shortest paths between them. For large b, this requires many paths to be pruned away for efficient turnaround. For this topology, an algorithm using a priority queue proves effective in reducing the number of search paths. This routing method maintains a queue of partial paths, sorted in priority according to the earliest time the message can be routed through every link in the partial path, earliest times first. Initially, the queue contains values for each link stemming out from the source processor. At each step, the algorithm grabs the top element in the queue, extends the partial path by one link toward the destination processor, determines the earliest message completion time for this partial path, and inserts this new path into the queue sorted in order. A partial path is only deleted

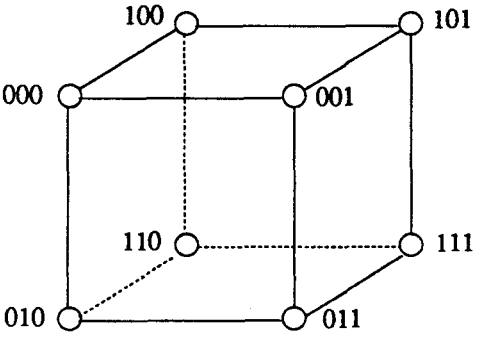


Figure 3-22. An eight-processor hypercube

from the queue when all of its possible extensions toward the destination have been examined. This procedure is repeated until a complete path from source to destination processor remains at the top of the queue. The completion time for this path, C^*, is guaranteed to be the earliest time the message can be routed, because any other path would have to use one of the partial paths below it, which have completion times greater than or equal to C^*.

To illustrate this procedure, consider an example in which node A finishes execution on processor 000 at time 5, at which time it sends 2 data units to node B. The scheduler is considering execution of node B on processor 111, and wishes to find the earliest time the data can be routed from processor 000 to processor 111. Assume that the data transfer between these processors takes 5 time units, and that the current link records are as given in figure 3-23. As shown in part a) of figure 3-24, the priority queue initially contains the earliest completion time (ECT) for the message in each of the three links stemming out of source processor 000. Even if the link has no previous reservations, the earliest completion time is 10, because node A finishes execution at time 5, and an additional 5 time units are required for data transmission. In the first

Link	Times of use
000-001	(0,6)
000-010	(2,4)
000-100	
001-011	(0,6)
001-101	
010-011	
010-110	(1,3),(3,6)
011-111	(4,8)
100-101	(4,7)
100-110	
101-111	(1,4)
110-111	(1,3)

Figure 3-23. The link records at the current state

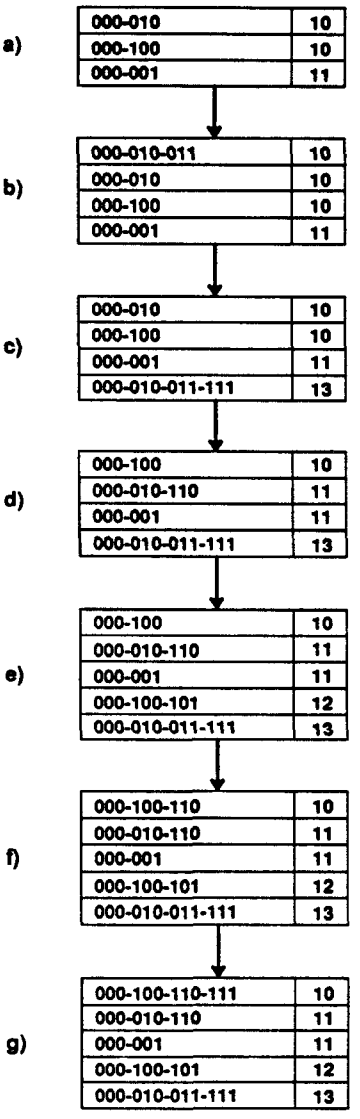


Figure 3-24. The steps taken in the priority queue

step, the algorithm grabs the element at the top of the queue, link 000-010, and extends it one link further to obtain partial path 000-010-011. Since the ECT of this partial path is still time 10, this partial path remains at the top of the queue, as shown in part b) of figure 3-24. The algorithm then extends this path one link further to obtain path 000-010-011-111. However, since the link from 011-111 is already being used in the time interval (4,8), the ECT for this path is 13, causing this path to drop to

the bottom of the queue as shown in part c) of figure 3-24. Partial path 000-010-011 is deleted from the queue because the only shortest path from 000-010-011 to 111 has already been considered. Link 000-010, which is at the top of the queue is then extended to form partial path 000-010-110, which has ECT 11. Since both shortest path from 000-010 to 11 have been considered, link 000-010 is deleted, causing link 000-100 to rise to the top as shown in diagram d) in figure 3-24. Partial path 000-100-101 with ECT 12 is then obtained, as shown in diagram e), and the next two steps extend link 000-100 to successive paths 000-100-110, and 000-100-110-111. Since this last path is a complete route from source to destination which remains at the top of the queue, the earliest time the message can be routed is its ECT time 10.

3.5. SCHEDULING ENHANCEMENT TECHNIQUES

While the DLS methods are promising, they are heuristics designed primarily for fast scheduling which can be improved using several additional techniques.

3.5.1. Weighting Factor

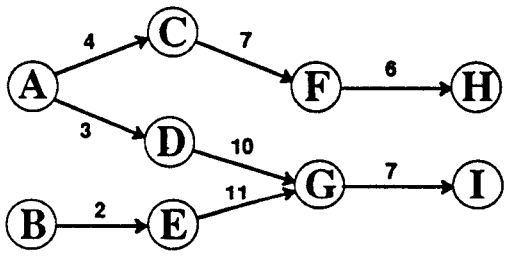
One possible technique to improve scheduling performance is to incorporate weighting factors α $(0 < \alpha < 1)$ and β into the dynamic level expression, so that $DL_1(N_i, P_j, \Sigma)$ equals

$$SL^*(N_i) - [\alpha\, DA(N_i, P_j, \Sigma) + (1-\alpha)\, TF(P_j, \Sigma)] + \beta\Delta(N_i, P_j). \qquad (3.20)$$

In an iterative scheduling scheme, one could adjust factors α and β based on the results of the previous schedule. Increasing α gives more emphasis to IPC, while decreasing α gives more weight to load balancing. β can be adjusted depending on the variance of execution speeds for a given node over all the processors.

3.5.2. Forward/Backward Scheduling

Due to its method of operation, dynamic level scheduling has more difficulty scheduling graph structures which converge than those which diverge. For this reason, it is often advantageous to schedule the precedence graph in the backward as well as the forward direction. Consider the precedence graph G shown in figure 3-25, and assume that the architecture consists of two processors interconnected by a full-duplex data link. We will assume for simplicity that the amount of time needed to communicate D data units is just D time units. After applying each of the algorithms discussed in section 3, the best result obtained from scheduling G in the forward direction is shown in figure 3-26, which has a makespan of 25 time units. To schedule graph G backwards, one simply schedules the graph $\tilde{G}$ shown in figure 3-27, in which every arc in G is inverted. This reverses both the precedence constraints and the direction of every data transfer between nodes. Scheduling this reversed graph using DL_2 with



Execution Times

	P0	P1
A	3	5
B	5	6
C	2	4
D	4	4
E	5	3
F	2	3
G	4	6
H	5	6
I	4	2

Figure 3-25. Original graph G

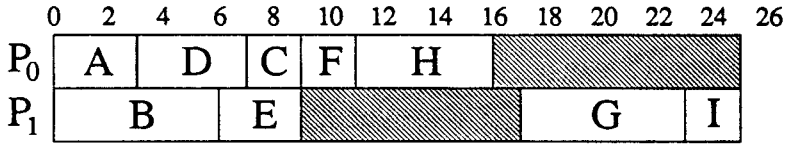
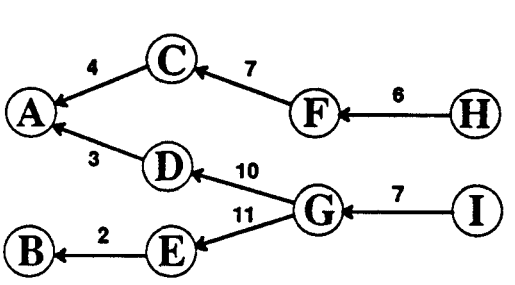


Figure 3-26. Schedule for graph G



Execution Times

	P0	P1
A	3	5
B	5	6
C	2	4
D	4	4
E	5	3
F	2	3
G	4	6
H	5	6
I	4	2

Figure 3-27. Inverted graph $\tilde{G}$

only a single node considered for scheduling at each step results in the schedule shown below in figure 3-28, which has a makespan of 20. Clearly, any admissible schedule for the inverted graph is a legal schedule for the forward graph if the time scale is inverted, as shown for this example in figure 3-29.

The simplest approach for using forward-backward scheduling is just to schedule a graph G in both the forward and backward directions, keeping the better result of the two. If reverse scheduling produces the better result, an admissible schedule can be obtained by inverting the time scale in the schedule for $\tilde{G}$. Alternatively, the steps executed when scheduling $\tilde{G}$ can be stored and repeated in reverse order when

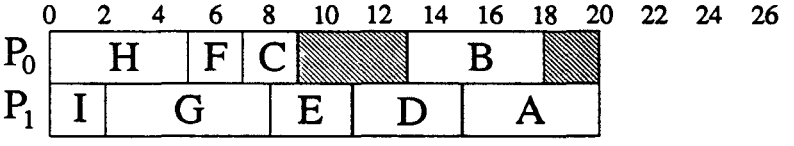


Figure 3-28. Schedule for inverted graph $\tilde{G}$

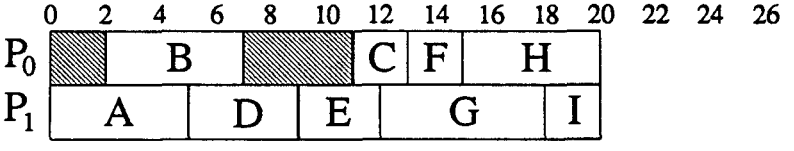


Figure 3-29. Inverted schedule for graph $\tilde{G}$

scheduling graph G in the forward direction. The schedule obtained from using the stored scheduling steps in this example is shown below in figure 3-30. Rescheduling graph G in the forward direction is not strictly necessary; however, it allows the initially executable nodes to start at time zero as in figure 3-30, instead of being staggered, as in figure 3-29. When rescheduling in the forward direction, it is important to preserve both the processor assignments and the specific paths used to route communications between processors. In addition, the scheduling and routing steps taken during backward scheduling must be executed in precisely reversed order to obtain the proper outcome. The results obtained from keeping the best schedule in forward-backward (FlB) scheduling are shown in table 3-6 below for the DL2 and GDL2 algorithms.

3.5.3. Precedence Constraint Appendage

A more productive use of backward scheduling can result from applying the information obtained in scheduling $\tilde{G}$ to aid in scheduling the original graph G. The scheduler can obtain clues as to how upcoming nodes should be scheduled by examining how they were scheduled coming from the other direction. One means of storing

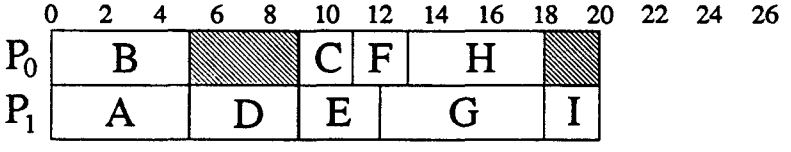
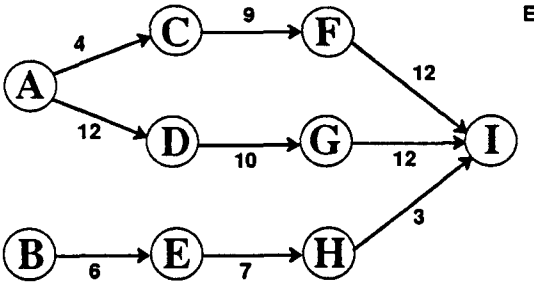


Figure 3-30. Schedule for G using the scheduling steps from $\tilde{G}$

Percentage Improvement in Speedup Over HLFET					
Algorithm	Mean	Std. Deviation	Minimum	Median	Maximum
FlB, DL2, 3 Nodes	140.96%	57.32%	76.07%	123.08%	375.70%
FlB, GDL2, 3 Nodes	145.66%	64.62%	83.23%	126.27%	447.31%

Table 3-6. Speedup improvements in using forward-backward scheduling

these clues from the backward schedule is to alter the graph G by adding precedence constraints in the forward direction to form graph G'. Since all the original precedence constraints are still present, this procedure does not alter the algorithm; every schedule which satisfies the constraints in G' is guaranteed to satisfy the constraints in G. In fact, the set of admissible schedules for G' form a subset of the set of admissible schedules for G. This technique of adding precedence constraints was first introduced in an optimal scheduling strategy for homogeneous processors when interprocessor communication is not considered [Bus74]. Additional precedence constraints were used to limit the parallelism in an application to match the number of processors. In the present context, when interprocessor communication must be accounted for, this technique can be used to limit the amount of parallelism in cases where the exploitation of such parallelism is too expensive due to communication costs. Consider the APEG G shown below in figure 3-31, which will be scheduled onto a set of 3 identical processors. After node A has been scheduled, it appears upon inspection of arcs AC and AD that nodes C and D should be placed on different processors. The dynamic level algorithm places C and D on different processors, resulting in the schedule shown in figure 3-32, which has a makespan of 33. Backwards scheduling



Execution Times

A	5
B	8
C	4
D	7
E	7
F	5
G	3
H	6
I	6

Figure 3-31. An example APEG

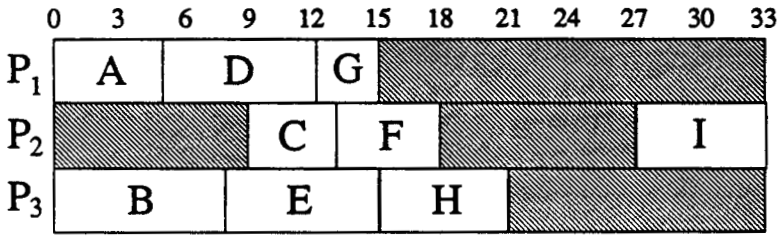


Figure 3-32. Schedule for the original graph

reveals that the parallelism between paths CF and DG should not be exploited, causing graph G' to be created by inserting a single precedence constraint between nodes C and D in graph G. Scheduling graph G' results in the schedule shown in figure 3-33, which has a makespan of 30 and uses only 2 processors.

3.6. SUMMARY AND CONCLUSIONS

Dynamic-level scheduling is a fast, single-pass, compile-time scheduling strategy which accounts for interprocessor communication overheads when mapping precedence graphs onto multiple processor architectures. This technique eliminates shared resource contention by performing scheduling and routing simultaneously to enable the scheduling of all communications as well as all computations. In heterogeneous processing environments, it accounts for varying processor speeds and delivers a more careful apportionment of processing resources. The algorithm is split

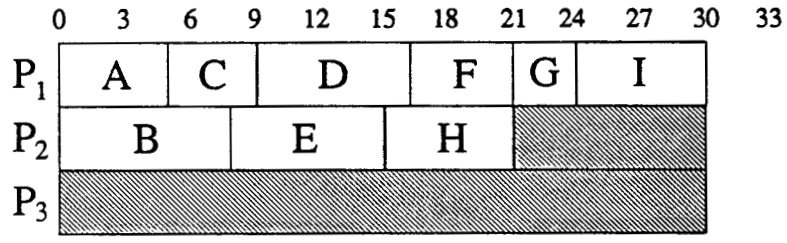


Figure 3-33. Schedule for the precedence-modified graph

into two sections to permit a rapid retargeting to any desired multiple-processor architecture by loading in the correct topology-dependent section. This approach attains good scheduling performance by effectively trading off load balancing with interprocessor communication, and efficiently overlapping communication with computation.

The main drawback to the dynamic-level scheduling approach is that it demonstrates greedy scheduling behavior, primarily as a result of the need for scheduling speed. When scheduling the partial graph structure shown figure 3-34, the algorithm schedules nodes B and C solely based on the dynamic levels of B and C with the processors. It does not consider if the total structure looks like the top graph in figure 3-35, in which case nodes B and C should probably be mapped onto the same processor to avoid excessive communication cost, or if it looks like the bottom graph in figure 3-35, in which case nodes B and C should probably be separated onto different processors to exploit the parallelism. As a result of this local view, the algorithm occasionally makes unwise scheduling decisions. However, this phenomenon is not confined solely to the dynamic-level scheduling algorithm, but rather is a property of single-pass scheduling approaches, due to the difficulty in evaluating the impact of a

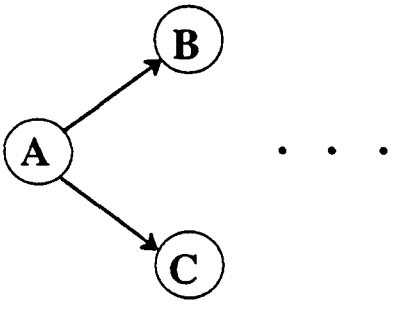


Figure 3-34. A partial graph

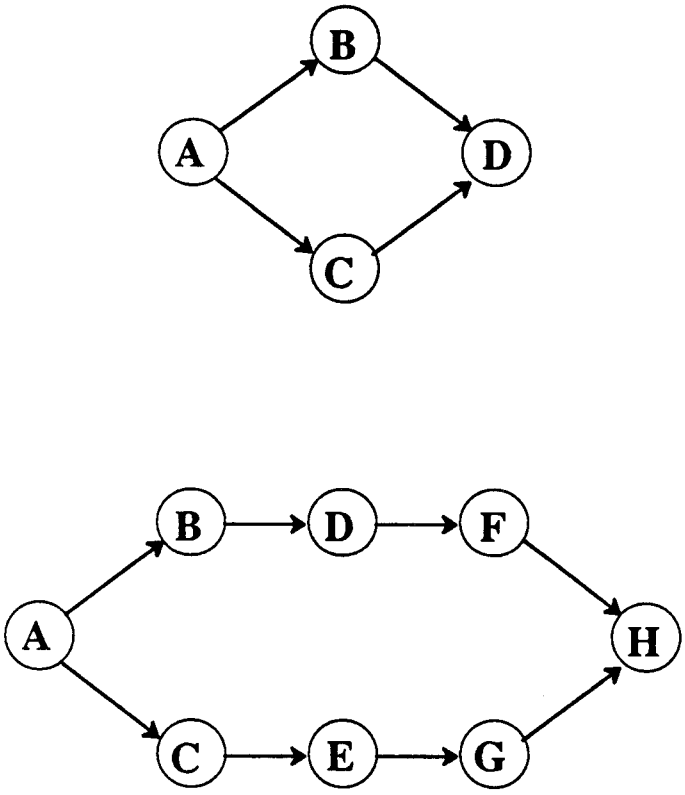


Figure 3-35. Two possible expansions of the partial graph in figure 3-34

scheduling decision at any intermediate point in the scheduling process. Node placements which appear logical when viewed locally may produce a poor end result. Although it is possible to "look ahead" one or more scheduling steps before making a decision, scheduling exhibits a phenomenon known in the artificial intelligence community as the "horizon effect". This property asserts that no matter how far one looks ahead before making a local decision, there may exist something "just over the horizon" which can render the decision detrimental to the objective. While this property does not exclude the possibility of gaining increased scheduling performance from looking ahead, it is important to realize that any improvement comes at the expense of scheduling speed, due to the increased computation required at each scheduling step.

4

DECLUSTERING

Imagination is more important that knowledge

— Albert Einstein

This chapter presents a second compile-time scheduling heuristic called **declustering**, which addresses the deficiencies exposed in the dynamic-level scheduling approach. This algorithm takes a global scheduling perspective by using new graph-analysis techniques to explicitly address the tradeoff between exploiting parallelism and incurring IPC cost. Instead of using a single scheduling pass, this technique takes an iterative scheduling approach, performing most of its computation in between scheduling iterations. Rather than expend computation to decide node placements during the actual scheduling process, this method uses computational energy more effectively by

analyzing a schedule and finding the most promising alternative placements.

The declustering method also overcomes disadvantages introduced by traditional clustering approaches. By grouping nodes into higher-granularity units, clustering techniques constrain the possible mappings of nodes to processors, limiting their ability to balance processor loads. Such algorithms also face significant difficulties when mapping clusters to the physical processor architecture. For example, consider the graph shown in figure 4-1, which naturally decomposes into the three clusters ABCDE, FGHI, and JKLM. If the architecture has three processors, this clustering proves effective, producing the schedule shown in figure 4-2 with makespan 21. Here, we have assumed that the targeted architecture is a shared-bus multiprocessor using a send/receive IPC protocol. The processors execute send and receive communication

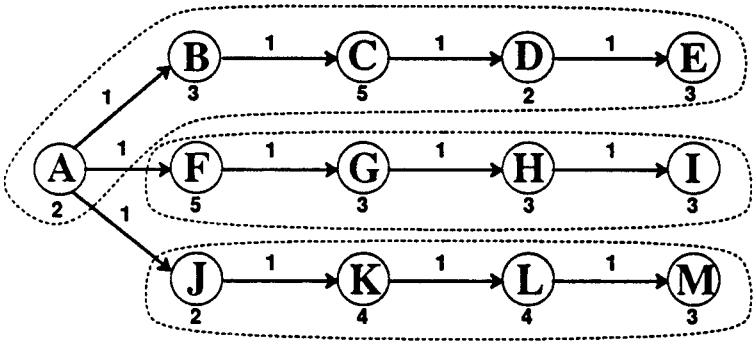


Figure 4-1. An APEG broken into three clusters

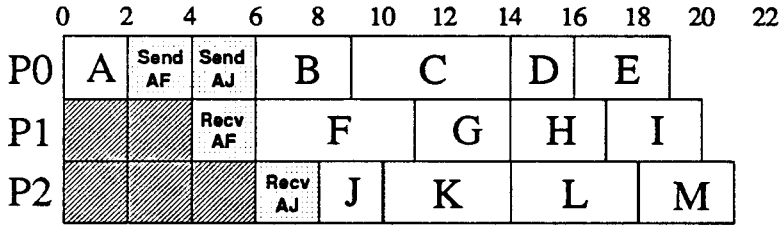


Figure 4-2. The clusters scheduled onto three processors

nodes taking two time units each, where we allow a send node for one communication to be overlapped with a receive node of another communication. These assumptions are made solely for purposes of illustration and are not intrinsic to the proposed algorithm itself. Now, if the graph in figure 4-1 is to be scheduled onto a 2-processor architecture, the clustering scheme performs poorly, producing the schedule shown in figure 4-3 with makespan 30. Both the *linear clustering* and *internalization clustering* methods described in chapter 2 will produce similar schedules, in which two clusters are scheduled on one processor and one cluster is scheduled onto the other processor. The load imbalance, which results in low processor utilization, is caused by the failure of these traditional clustering schemes to account for the number of processors in the architecture when choosing the granularity of the clusters. This difficulty in accounting for architectural considerations during scheduling is a primary motivation for the declustering approach. The question of whether a given instance of parallelism should be exploited is complex, requiring consideration of the graph structure, the node granularity, the number of available processors, the cost of IPC, and the structure of the processor interconnection topology. Furthermore, when processing resources are limited, one faces the additional problem of exploiting some parallelism instances at the expense of others.

The declustering algorithm is specifically designed to address these scheduling considerations. Similar to the dynamic-level scheduling algorithm, the declustering

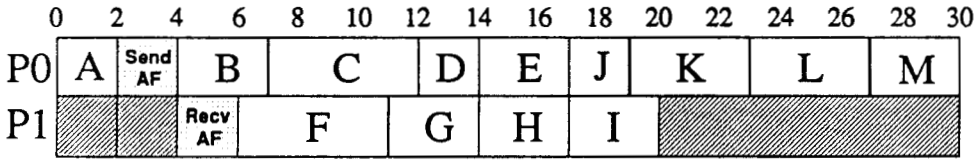


Figure 4-3. The clusters scheduled onto two processors

technique is split into topology dependent and independent components to account for the interconnection structure, and the scheduling of all communications as well as all computations removes the possibility of contention for communication resources. The declustering algorithm is divided into four main sections, as shown in figure 4-4.

The first stage presents a new clustering approach which divides the graph into elementary clusters using a novel set of parallelism analysis techniques. This phase considers the graph structure while explicitly addressing the tradeoff between exploiting parallelism and incurring IPC costs. The second stage combines the clusters in a

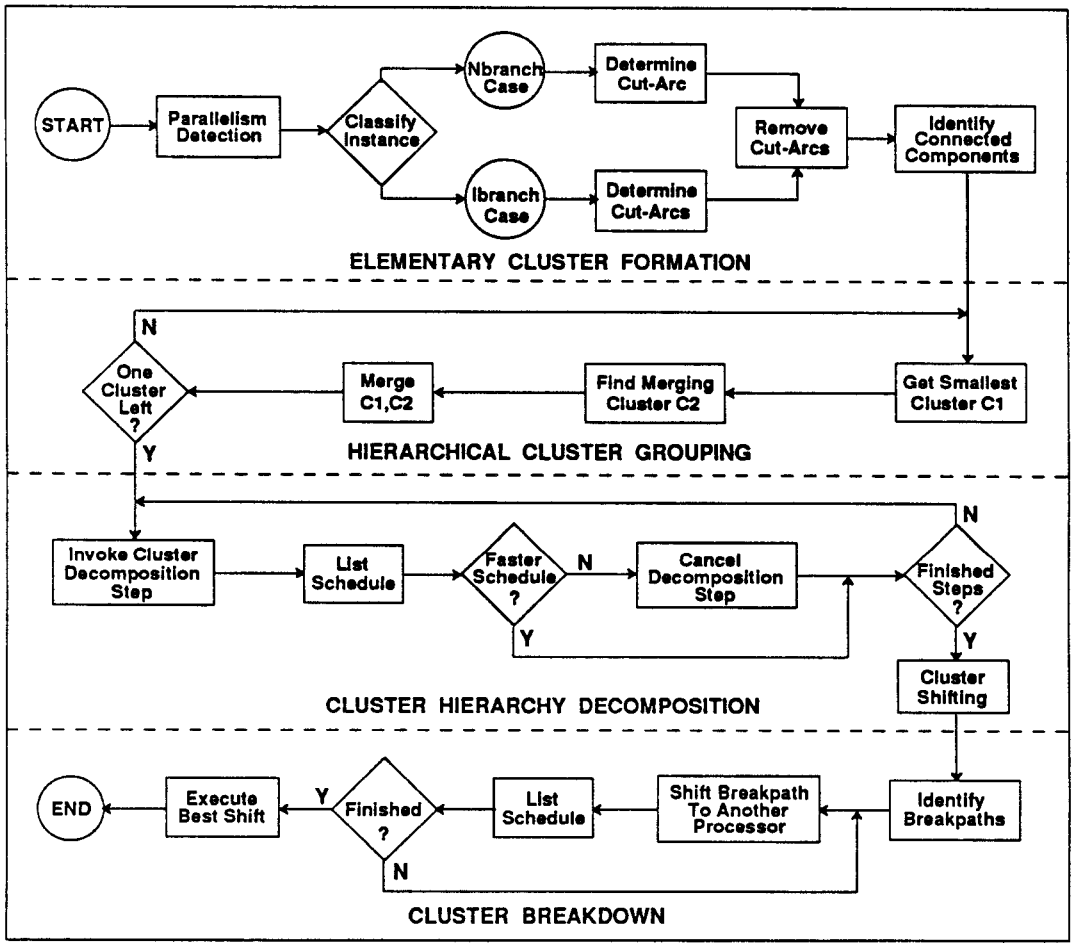


Figure 4-4. A flowchart description of the declustering algorithm

hierarchical fashion by considering the IPC and parallelism relationships between them. It effectively ranks the instances of graph parallelism in preparation for declustering. The third stage decomposes the hierarchy by systematically breaking higher level clusters into their component sub-clusters and mapping one of the sub-clusters onto a different processor. This phase insures effective use of the available processors by exploiting the parallelism instances in order of importance. The fourth stage analyzes the best schedule obtained so far and breaks down the elementary clusters if additional flexibility is needed to achieve an effective load balancing. By traversing the cluster hierarchy from top to bottom (large-grain to small-grain), this "declustering" process matches the level of cluster granularity to the characteristics of the architecture.

4.1. ELEMENTARY CLUSTER FORMATION

The **elementary cluster formation** phase consists of a new clustering procedure which decomposes the precedence graph into groups of nodes by isolating a collection of arcs which are likely candidates for separating the nodes at both ends onto different processors. These **cut-arcs** are temporarily cut, or removed from the graph and the algorithm designates each remaining connected component as an elementary cluster.

The problem of finding an effective set of cut-arcs is complex. An insufficient number of cut-arcs constrains the possible processor assignments severely, causing reduced scheduling performance. Conversely, an overabundance of cut-arcs leads to an excessive time required for scheduling. We examined several methods for selecting cut-arcs before selecting the final approach. To clarify some terminology, a **branch** node is defined as a node that has two or more immediate successors, while a

merge node is defined as a node possessing two or more immediate predecessors.

One approach considered for selecting cut-arcs designates every arc output from a branch node and every arc input to a merge node as a cut-arc. This technique is effective when the APEG arcs are homogeneous, meaning that each node sends and receives the same number of data units on each arc. Since the communication penalty for placing any two communicating nodes on different processors is exactly the same, the graph will normally be broken at locations which permit the greatest amount of parallelism to be exposed, namely the arcs output from a branch node or input to a merge node. To illustrate this point, consider the simple homogeneous graph shown in figure 4-5.

Partitioning this graph onto two processors entails breaking two arcs in the graph. However, since the graph is homogeneous, breaking any two arcs in the graph incurs the same communication penalty. To minimize schedule length, the graph should therefore be broken to release the greatest amount of parallelism, which lies between paths {B-C-D} and {E-F-G}. Therefore, the minimum makespan solution can be obtained by breaking arcs AE and DH, or AB and GH. However, in a nonhomogene-

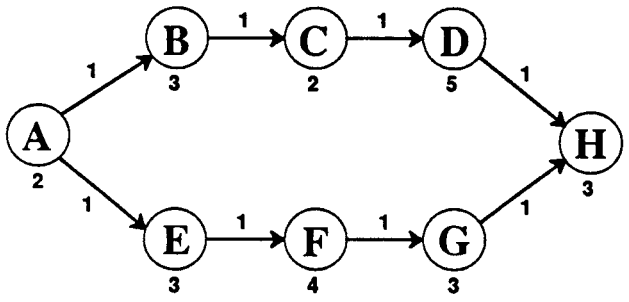


Figure 4-5. A homogeneous graph

ous setting, it may be advantageous to break arcs not directly connected to branch or merge nodes. Consider the same graph in figure 4-6, in which the number of data units on the arcs have been changed to make the graph nonhomogeneous. Now the minimum makespan solution is to cut the arcs CD and EF, neither of which is directly connected to branch node A or merge node H. This proposed approach to breaking arcs is clearly deficient in the nonhomogeneous case. Another problem is that this technique produces an excessive number of elementary clusters for dense graphs, which lengthens the time required for scheduling.

The key to effective cut-arc selection lies in skillfully trading off the amount of parallelism exploited with the amount of interprocessor communication cost incurred. Obtaining such a tradeoff requires both a method for detecting parallelism within the graph, and an effective means of comparing parallelism with communication cost.

4.1.1. Parallelism Detection

To enable parallelism detection, we assign each node N_i its **transitive closure** $TC(N_i)$, defined as the set of nodes (excluding terminal node N_t) reachable through a directed path from N_i. If node N_i lies in the transitive closure of node N_j, this implies that a precedence constraint exists from N_j to N_i. Stated another way, nodes N_i and

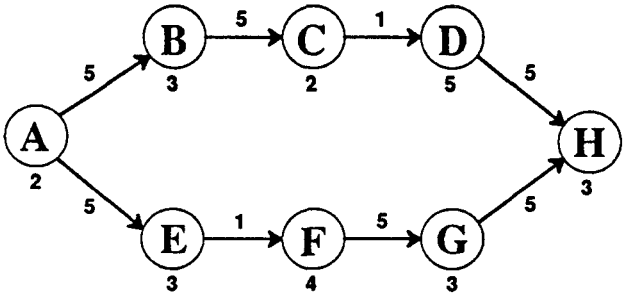


Figure 4-6. A nonhomogeneous graph

N_j can be executed in parallel if and only if each node is not contained in the other's transitive closure. While this reveals parallelism at the node level, our goal is to identify parallelism between paths of nodes. Since it is virtually impossible to account for all the path parallelism in the graph simultaneously, we use a "divide and conquer" approach which considers two paths at a time. Recognizing that parallelism is created at branch nodes, we focus attention on paths which diverge out from these nodes. The branch nodes in the graph are sorted smallest static level first, so that the branch nodes near the end of the graph are initially considered. For each branch node N_i, the algorithm obtains a list of its immediate successors $IS(N_i)$ and sorts them largest static level first. At each step, the procedure considers the first two successors in the list and computes the intersection of their transitive closures to categorize this instance into one of two classes, called the *Nbranch* and *Ibranch* cases respectively. To illustrate this procedure, consider the case in which branch node A has two immediate successors B and C.

If $TC(B) \cap TC(C)$ is empty, the paths stemming from B and C never combine. We classify this instance into the *Nbranch* (Nonintersecting branch) case and isolate the longest paths from nodes B and C to terminal node N_t for parallelism consideration. In the example Nbranch case shown in figure 4-7, the available parallelism between the two paths is min {SL(B), SL(C)}. This is the maximum overlap in execution time possible if the paths starting from B and C are separated onto different processors.

If $TC(B) \cap TC(C)$ is not empty, this instance fits into the *Ibranch* (Intersecting branch) case because the paths stemming from nodes B and C combine at some point. The node in the intersection with largest static level is identified as the first merge node at which paths starting from B and C combine, and the algorithm isolates the

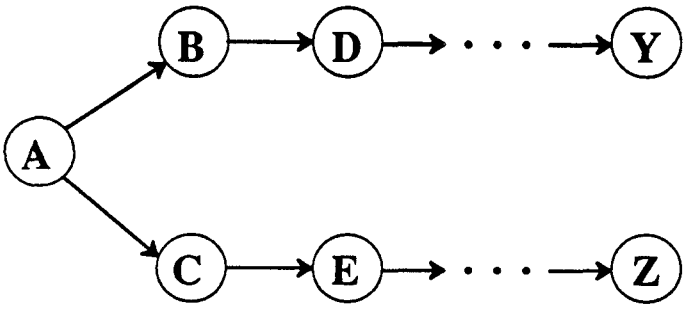


Figure 4-7. The nonintersecting branch case

longest path from B to the merge node and the longest path from C to the merge node for parallelism consideration. This situation is illustrated in figure 4-8, where node Z is the merge node. If we designate the string of nodes from B to X as path1 and the string of nodes from C to Y as path2, then the available parallelism in this case is the minimum of the sum of node execution times in path1 and the sum of node execution times in path2.

4.1.2. Parallelism Exploitation

After parallelism detection, the next step is to develop a criterion for determining whether the path parallelism can be exploited effectively. A reasonable first guess might be to discard any parallelism which is exceeded by the communication cost in

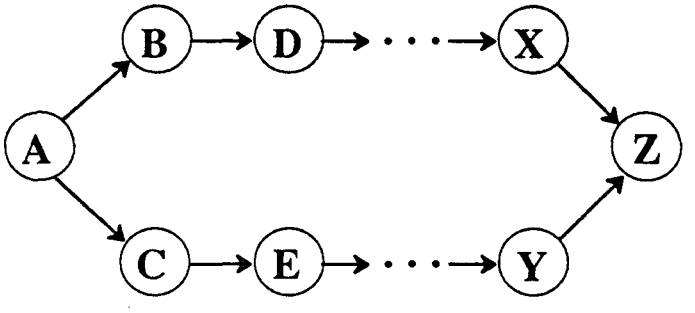


Figure 4-8. The intersecting branch case

separating paths onto different processors. However, when the scheduling goal is to minimize makespan, it is easy to find examples in which the exploitation of parallelism allows an earlier finishing time even when this condition is violated. For example, the graph shown in figure 4-9 will finish execution at time 24 if it is scheduled on a single processor. The available parallelism between the two paths is min (8 , 14) = 8, so one might expect that an IPC cost of 8 time units or greater would preclude the exploitation of parallelism in this example. However, the schedule shown in figure 4-10 finishes at time 22 even though the communication cost is 12 time units. So when the goal is to minimize schedule length, we consider exploitation of all two-path parallelism instances that can finish execution more rapidly on two processors than on one processor.

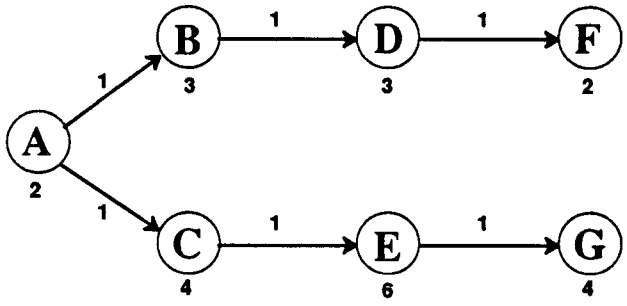


Figure 4-9. An example graph

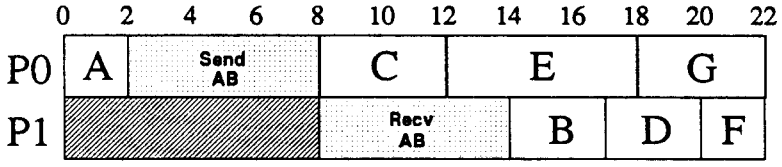


Figure 4-10. A schedule for figure 4-9 when the IPC cost is 12 time units

4.1.3. Cut-arc Determination

This method of categorizing two-path parallelism instances into the Nbranch and Ibranch classes provides a tractable means of determining cut-arcs, because finding the minimum makespan solution for each instance is a straightforward calculation in both of these cases. If the parallelism instance can be exploited effectively, the algorithm finds which single arc will be cut in the Nbranch case, or which two arcs will be cut in the Ibranch case.

To illustrate how a graph is broken into two-path parallelism instances, we demonstrate the procedure on the graph shown in figure 4-11. The algorithm first identifies nodes A and I as branch nodes. Node I, which is examined first, is found to have two immediate successors M and N, where $TC(M) \cap TC(N) = \phi$. This two-path parallelism instance, shown in figure 4-12, therefore fits into the Nbranch case. Again, assuming the architecture is a shared-bus multiprocessor which executes send and receive communication nodes taking two time units each, the algorithm quickly

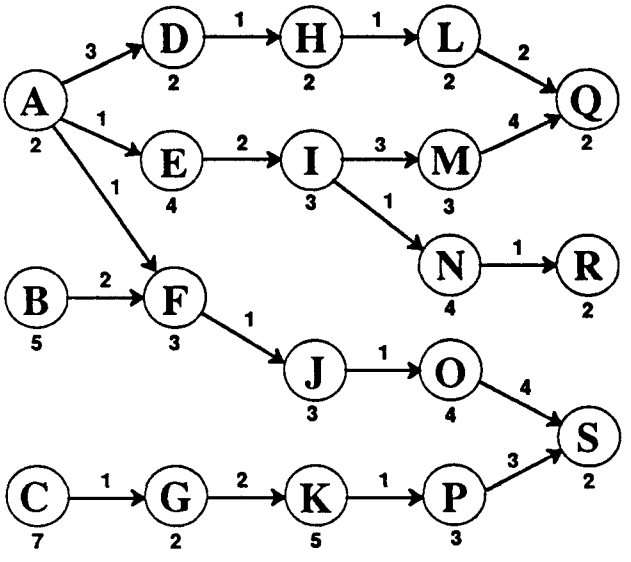


Figure 4-11. An example graph

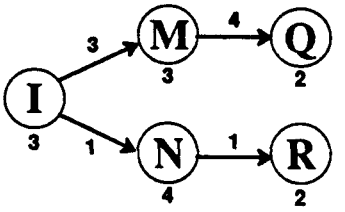


Figure 4-12. A nonintersecting branch case

ascertains that separating paths {I-M-Q} and {N-R} onto separate processors leads to the minimum makespan solution. It therefore adds arc IN to the list of cut-arcs and moves on to examine branch node A. Since node A has more than two immediate successors, the procedure sorts them into a list largest static level first : $IS(A) = [E\ F\ D]$. At each step, it considers the two remaining successors with largest static level, which are initially E and F in this case. After finding that $TC(E) \cap TC(F) = \phi$, the algorithm classifies this instance into the Nbranch case and isolates the two longest paths stemming from nodes E and F for parallelism consideration, as shown in figure 4-13. To facilitate computation of the minimum makespan solution for this instance, we code each arc in the two paths with a triad of information in the following form: (runtime_passed communication_cost runtime_ahead). The first element is the amount of execution time in the path which lies to the left of the

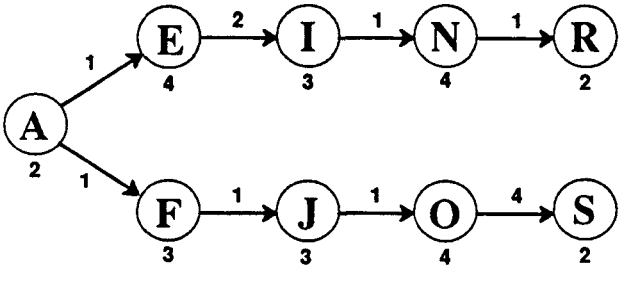


Figure 4-13. Another nonintersecting branch case

arc. The second element indicates the amount of time spent in interprocessor communication if the arc is split. The third element shows the amount of execution time in the path which lies to the right of the arc. The parallelism instance in figure 4-13 is shown with coded arcs below in figure 4-14. The IPC costs reflect the same assumptions made earlier, where send and receive nodes of two time units each must be executed for transfer of every data unit. For arcs which pass multiple data units, such as EI and OS, the time spent in IPC reflects a pipelining of communications, obtained by overlapping the receive node of one data unit with the send node of the next data unit. This pipelining allows the time $C(D)$ to communicate D data units to be expressed as $C(D) = 2^*(D + 1)$. Because these triads contain all the pertinent scheduling information, the algorithm can quickly compute the minimum makespan solution without having to repeatedly schedule the two paths. In this case, the optimum solution is to cut arc AF. Since this cut-arc lies in the path containing successor node F, the algorithm removes node F from the list of successors, so that $IS(A) = [E\ D]$.

The algorithm then examines the next two successor nodes, D and E, and determines that node Q is the first merge node in $TC(D) \cap TC(E)$, at which the paths starting from D and E combine. This Ibranch case, shown in figure 4-15, requires two cut-arcs to split paths onto separate processors. After coding the arcs with triads, the algorithm

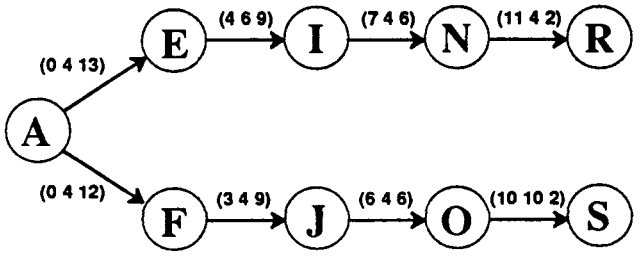


Figure 4-14. The parallelism instance with coded arcs

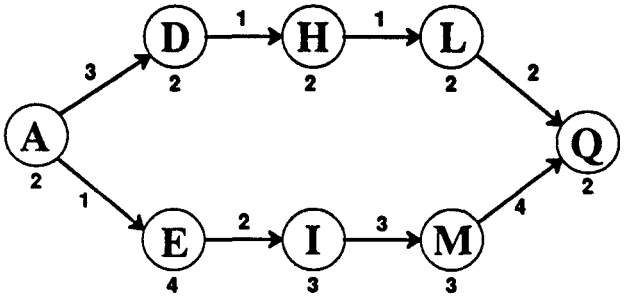


Figure 4-15. An intersecting branch case

first calculates the makespans for cutting pairs of arcs in the four "corner" cases (AD,LQ), (AD,MQ), (AE,LQ), and (AE,MQ), which are pairs of arcs directly connected to the branch node or merge node. The procedure uses the shortest makespan obtained over the corner cases as a bound to quickly eliminate other eligible pairs of cut-arcs. Due to the excessive communication costs in this case, the minimum makespan solution is to leave all arcs uncut; that is, schedule the entire structure on a single processor. In such cases when no path is cut, the algorithm deletes the successor with smaller static level from $IS(A)$ to prevent the same two paths from being analyzed again, and moves on to examine the next two nodes in $IS(A)$. If there is only one successor remaining, as in this case, the routine passes to the next branch node.

When all the branch nodes have been exhausted, the algorithm connects a dummy start node N_s to all the initial nodes in the graph, and computes static levels and transitive closures for each node in the reverse direction (right to left), denoting these quantities as $SLR(N_i)$ and $TCR(N_i)$ respectively. The procedure identifies all the merge nodes in the graph and repeats the procedure applied to the branch nodes, except that path analysis proceeds in the opposite direction. In addition to treating cases in which the parallelism is initially extant, this consideration of merge nodes

handles situations in which two paths combine in several places. Since the parallelism detection strategy only considers the first merge node (with largest static level) when isolating an Ibranch case, the parallel paths combining at the later merge nodes would be ignored if all the merge nodes were not identified and analyzed.

Continuing our example, the algorithm examines merge node F and its two immediate predecessors A and B. After classifying this instance into the nonintersecting merge (Nmerge) case and coding arcs, the procedure determines that the best solution is not to cut any arcs. When considering merge node Q, the algorithm recognizes that this Imerge case has already been examined as an Ibranch case for node A and skips on to consider merge node S and its two immediate predecessors O and P. Since $TCR(O) \cap TCR(P) = \phi$, the algorithm classifies this instance into the Nmerge case, traces the two longest paths to the dummy start node as illustrated in figure 4-16, and isolates KP as the optimal cut-arc.

After all the cut-arcs have been determined, the algorithm temporarily removes them from the graph and invokes a depth-first search to isolate the connected components remaining in the graph. Each remaining component is designated as an elementary cluster. After cutting arcs IN, AF, and KP, the elementary clusters in our example are shown below in figure 4-17. For comparison, the linear clustering algorithm produces

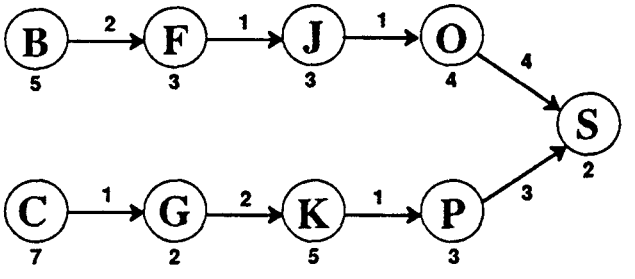


Figure 4-16. A nonintersecting merge case

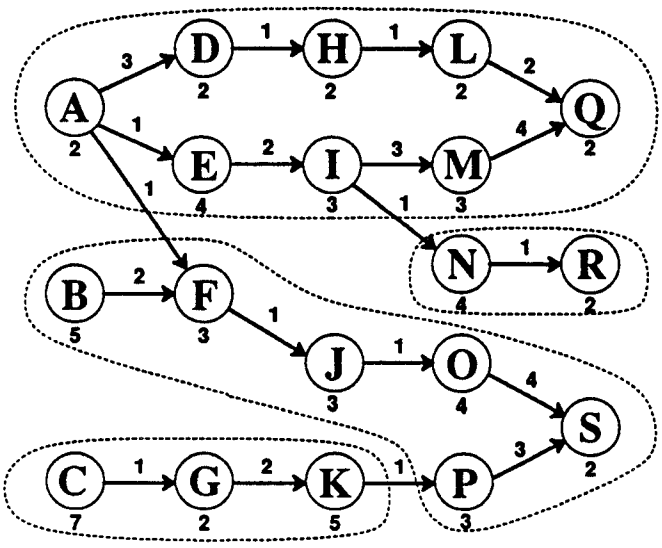


Figure 4-17. The elementary clusters in the graph

the set of clusters shown in figure 4-18, and the internalization approach produces the clusters shown in figure 4-19. It is important to realize that selecting an arc to be a cut-arc does not force the nodes at each end onto separate processors. Rather, these

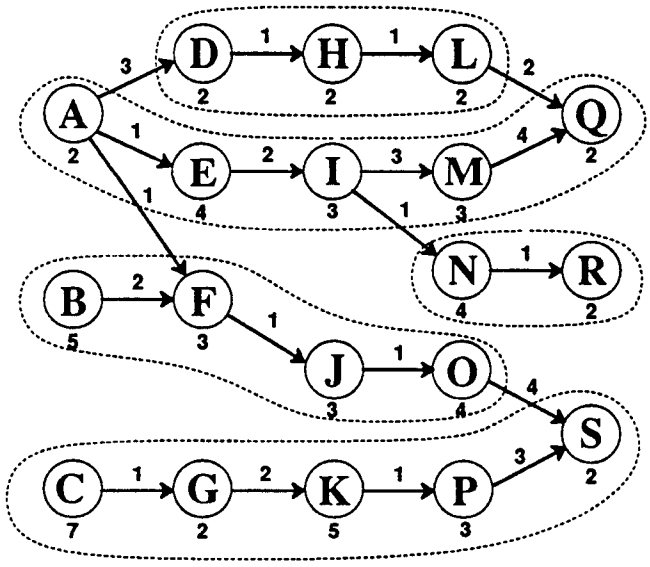


Figure 4-18. The linear clustering clusters

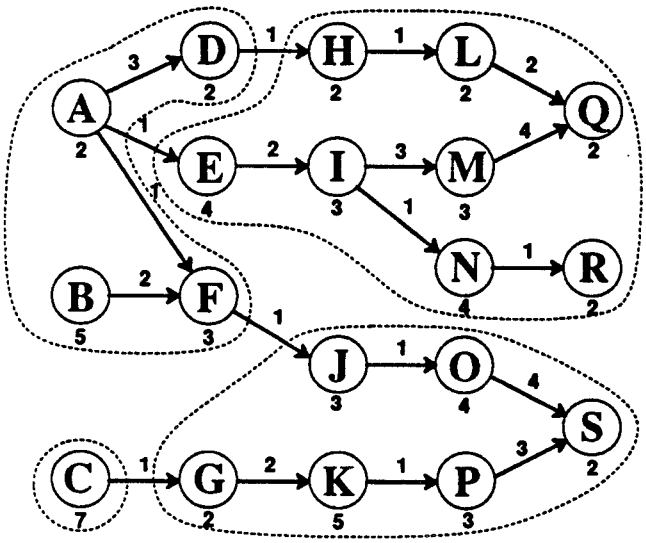


Figure 4-19. The internalization clusters

arcs represent promising locations for splitting paths onto different processors. The mapping decisions are made in later phases of the algorithm.

4.2. HIERARCHICAL CLUSTER GROUPING

The **hierarchical cluster grouping** stage combines existing clusters in a pairwise fashion until a single cluster remains which contains every node in the graph. This procedure establishes a parallelism hierarchy, effectively sorting the graph parallelism by importance in preparation for declustering.

This procedure initially sorts the elementary clusters by the sum of the execution times of the nodes they contain, to allow the smallest cluster, say C_1, to be considered for merging at each step. The algorithm determines which cluster communicates most heavily with C_1 by examining the intercluster cut-arcs. Ties are broken by arbitrarily selecting the cluster with smaller total execution time. The algorithm proceeds to merge C_1 and its selected cluster into a new larger cluster, and records this cluster

combination step (e.g. $[C_1 + C_2 = C_3]$) in order of execution for future reference. To account for nodes with hardware constraints, each cluster maintains a list of processors which are eligible to execute every node in the cluster. Two clusters will only be combined if there is at least one processor which can execute every node in both clusters. After a cluster combination step is invoked, the algorithm removes the two component clusters from the list of current clusters, adds the new larger cluster to the list in sorted order, and considers the next smallest cluster for merging. This procedure is repeated until only a single cluster remains, or the process has progressed as far as the hardware constraints permit. The procedure finishes by scheduling the remaining cluster(s) onto a processor which satisfies its hardware constraints to obtain an initial makespan.

The order in which the clusters are combined is important. The procedure selects the smallest cluster for combination at each step because combining this cluster with another does not suppress very much parallelism. While this technique incorporates interprocessor communication considerations, it builds clusters in such a fashion to maintain as much parallelism as possible for as long as possible. The end result is that the combination steps near the end merge clusters which have the largest amounts of parallelism between them, subject to the communication pattern in the graph. This effectively imposes a ranking of the instances of parallelism present in the graph, where the combination steps near the beginning suppress the less important parallelism instances, and the combination steps near the end suppress the most prominent parallelism instances.

4.3. CLUSTER HIERARCHY DECOMPOSITION

The **cluster hierarchy decomposition** phase begins the declustering process, in which the parallelism hierarchy constructed in the first two stages is decomposed into successively smaller levels. To avoid the inflexibility induced by the clustering process, the algorithm traverses cluster granularity levels from large to small to find the level which is roughly consistent with the characteristics of the target architecture.

This procedure examines the list of cluster combination steps in reverse order, so that the last consolidation step (say $[C_{18} + C_{19} = C_{20}]$) is considered first. It then invokes a decomposition step on C_{20} by shifting either subcluster C_{18} or C_{19} onto a selected group of candidate processors, which are selected by considering the cut arcs of the chosen subcluster, the finishing times of the processors, and any pertinent hardware constraints. The algorithm shifts the chosen subcluster onto each of the candidate processors in turn, and list schedules the graph to determine the makespan for each of the subcluster placements.

4.3.1. List Scheduling Method

As in the dynamic-level scheduling approach, the list scheduling method used here schedules all communications as well as all computations, employing a routing algorithm tailored for the particular architecture in the topology-dependent section of the scheduler. Likewise, there is no global timeclock to distinguish which nodes are runnable and which processors are available. A node is runnable if all its immediate predecessors have already been scheduled, and the processor assignments for each node are already fixed by the declustering algorithm before scheduling is invoked.

Static levels are used to determine the ordering of nodes on each processor. Dynamic levels are not necessary in this case because of the fixed processor assignments. After scheduling, a node reordering procedure is invoked which analyzes the schedule and identifies instances where a shift of node priorities permits a more effective schedule. We illustrate this idea using two examples.

In the first example, shown in figure 4-20, node B has static level 9 while node C has static level 8. This causes node B to be scheduled before node C, leading to the schedule with makespan 23 shown in the top chart of figure 4-21. The node reordering phase sees the idle time before the receive node and realizes that the communication can be executed earlier if a node scheduled before C can be moved behind it. By searching the transitive closure of each node scheduled before C on processor 0, it discovers that node B can be executed after C. The routine then changes the priority of node B below that of node C, resulting in the schedule shown in the lower chart in figure 4-21, which has makespan 18.

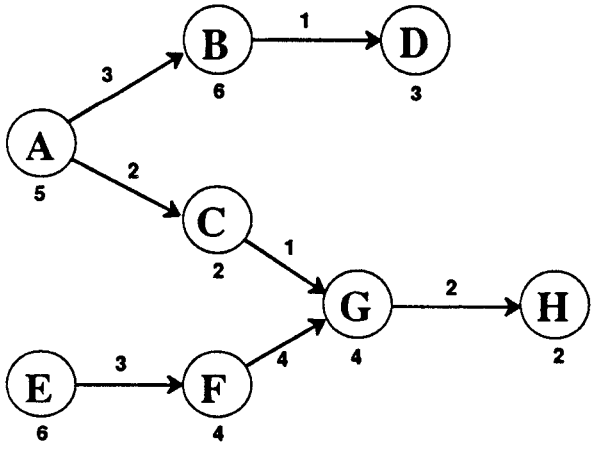


Figure 4-20. An example APEG

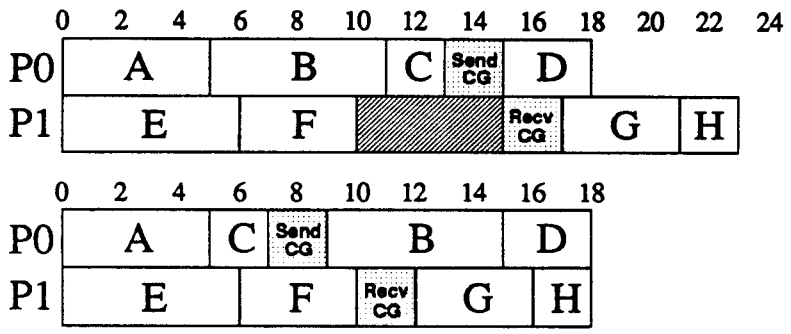


Figure 4-21. The reordering shifts node B in back of node C

In the second example, shown in figure 4-22, the the static level of node G exceeds the static level of node F. This can cause the schedule shown at the top of figure 4-23 which has makespan 22. The node reordering procedure again notices the idle time before the receive node. Since the communication cannot be moved forward any further, the technique next tries to shift nodes into the gap by searching for nodes scheduled after node G which are not contained in the transitive closure of G. Upon discovering that nodes F and G have no precedence relationships between them, the routine shifts the priority of node F above that of node G, resulting in the schedule shown at the bottom of figure 4-23.

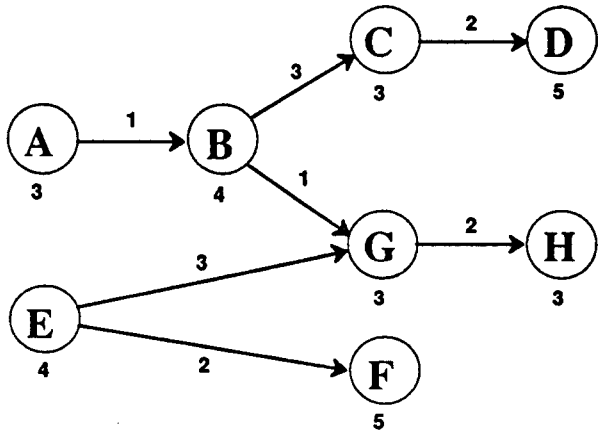


Figure 4-22. An example APEG

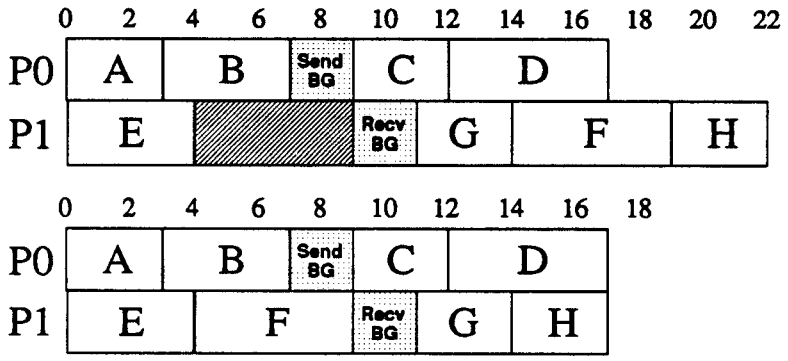


Figure 4-23. The reordering shifts node F in front of node G

After decomposing a cluster by splitting one of its component clusters onto another processor, the algorithm accepts the decomposition step if a faster schedule is obtained. If a tie in makespan occurs, the first tiebreak criterion selects the schedule with less IPC, and the second criterion selects the schedule with a smaller sum of processor finishing times. If the step is accepted, the procedure saves the new schedule and records a new processor location for the shifted subcluster. Otherwise, it ignores the step and considers the next cluster combination step (in reverse order) for decomposition. This process is repeated until all the cluster combination steps have been considered. This procedure constrains the number of processor placement permutations which are examined, because each cluster remains in its default position if its decomposition step is discarded.

When the list of cluster combination steps is exhausted, the algorithm invokes two ad-hoc cluster shifting techniques called *communication reduction* and *load shift*. These routines search for better processor placements at the current level of cluster granularity, and are repeated on successively lower levels of granularity until reaching the elementary clusters. This strategy is consistent with the overall approach in examining placements at the higher levels of cluster granularity before examining place-

ments at the lower granularity levels. This top-down strategy helps avoid overlooking the situation in which shifting two lower granularity component clusters individually does not produce a better result, but shifting these components together results in a better placement.

Rather than examine alternative placements in a haphazard fashion, the cluster shifting techniques attack the **schedule limiting progression** (SLP), the progression of nodes and communications which inhibits attainment of a faster schedule. While this progression may span several processors, it cannot contain any idle time. The SLP is a function of the particular schedule, and depends on the effectiveness of the scheduling procedure as well as the characteristics of the targeted architecture. In contrast with the critical path, the SLP is sensitive to changes in the number of processing and communication resources. An example schedule is shown below in figure 4-24, with the SLP marked by the sequence of arrows.

The *communication reduction* routine, which attempts to remove instances of interprocessor communication from the SLP, first isolates the SLP clusters at the current level of granularity. The routine traces the intercluster cut-arcs stemming from each SLP cluster and evaluates the difference between the number of data units passed to clusters on a different processor and the number of data units passed to other clusters on the same processor. Expressing this mathematically, we define $D\,[C_i, P\,(C_i)]$ to be

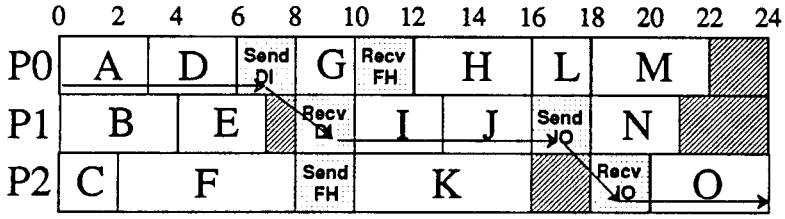


Figure 4-24. An example schedule with the SLP marked using arrows

the number of data units passed from cluster C_i to other clusters on the same processor $P(C_i)$, and $D[C_i, \overline{P(C_i)}]$ to be the number of data units passed from C_i to clusters on any other processor. We then define

$$\Delta(C_i) = D[C_i, \overline{P(C_i)}] - D[C_i, P(C_i)] \tag{4.1}$$

as the difference between the number of outgoing and ingoing data units for processor $P(C_i)$ attributable to cluster C_i. We arbitrarily designate the three SLP clusters with the highest values of $\Delta(C_i)$ as cluster switch candidates, because there is a good possibility that switching these clusters onto a different processor can lessen the amount of IPC incurred. For each cluster switch candidate, the procedure uses the $\Delta(C_i)$ criterion to identify several other clusters as candidates for exchanging processor locations with. The procedure invokes each cluster location switch individually and executes the switch which provides the greatest improvement in makespan.

The *load shift* routine moves clusters onto different processors to balance processor loads more effectively. It first categorizes each processor as being heavily loaded or lightly loaded. We distinguish these categories using a load threshold which is initialized to half of the maximum sum of the computation and communication costs on any processor. If all processors are heavily-loaded, the threshold is multiplied by 1.5 and this process is repeated until some processors fit into the lightly loaded category. The procedure invokes a sequence of cluster shifts from heavily loaded processors to lightly loaded processors and implements the shift which provides the greatest improvement in makespan.

While it may seem unusual to build a parallelism hierarchy, only to tear it down, this procedure provides several advantages. The sorting of graph parallelism which occurs during hierarchy construction allows effective use of the available processors. As

each new processor is pressed into service, the algorithm assigns it the largest section of unrealized parallelism remaining in the graph which has the smallest interprocessor communication cost. Faced with the inevitable problem of having to exploit some parallelism instances at the expense of others, the declustering algorithm insures that the most significant sections of graph parallelism are assigned first. The less important parallelism instances will be exposed later if there are sufficient resources remaining to warrant exploitation. By sweeping through a range of cluster granularities during cluster decomposition, this approach also performs some natural partitioning which automatically adapts to the grain size of the nodes. When the cluster granularity reaches the point where IPC costs negate any gains realized in exploiting parallelism, the algorithm discards any further decomposition steps. So in cases where the nodes are too fine-grained for the architecture to effectively use the internode parallelism, the process keeps the hierarchy at a higher level which is conducive to parallelism exploitation. If the desired granularity is smaller than the level of the elementary clusters, these clusters can be broken further in the next phase of the algorithm.

4.4. CLUSTER BREAKDOWN

Whereas the decomposition phase cannot manipulate subgraph segments smaller than the elementary clusters, the **cluster breakdown** phase has the ability to break down the elementary clusters, if additional freedom is required to achieve a better placement. This stage supplies the capability necessary for effective load balancing when the appropriate cluster granularity is smaller than the level of the elementary clusters.

The procedure first identifies the SLP nodes which are connected to at most one other SLP node on the same processor. Each of these SLP edge nodes denotes a starting

point for a *breakpath*, where a breakpath refers to a portion of the SLP which is a good candidate for being shifted onto another processor. Starting from an edge node, the procedure extends a path by one node further into the SLP on the same processor. The current path of nodes is considered a breakpath if the sum of execution times in the path falls between lower and upper bounds derived for this case. The lower bound insures that any gain created by shifting nodes onto a different processor exceeds the net IPC cost. The upper bound, set by considering the processor loads, insures that potential shifts have a possibility of obtaining a faster schedule. For each edge node, the algorithm finds the sequences of nodes which satisfy the bounding requirements. It then individually switches each breakpath onto another processor to obtain the makespan, and executes the split which produces the fastest schedule.

To illustrate this procedure, consider once again the APEG in figure 4-1, which is split into three elementary clusters. While this example maps onto three processors effectively by scheduling one cluster on each processor, the fastest two-processor schedule obtained after cluster hierarchy decomposition has makespan 30, as shown in figure 4-25. The load imbalance occurs because the granularity of the elementary clusters is too large for the two processor case. A smaller cluster granularity is required to obtain effective processor utilization. The SLP, which lies entirely on processor 0, consists of nodes {A SendAF J B K C L D E M}. The algorithm analyzes the two

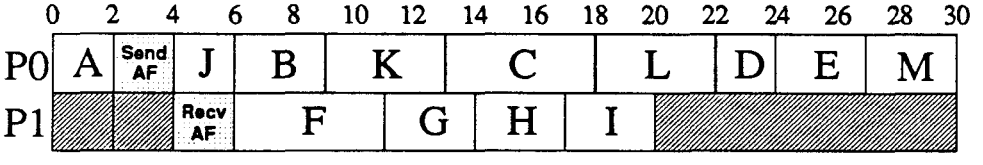


Figure 4-25. The fastest schedule obtained after cluster shifting

edge nodes in this group, E and M, and uses the bounding techniques to obtain the possible breakpaths {E, DE, CDE, M, LM, and KLM}. After shifting processor assignments and list scheduling the graph in each of these cases, the procedure determines that splitting DE onto processor 1 gives the best schedule, which is shown in figure 4-26. This ability to break down cluster granularity beyond the elementary level is essential, because as illustrated in the preceding example, the most effective cluster granularity is determined by the characteristics of the architecture.

While this cluster breakdown technique is intrinsic to the declustering process, it can be applied successfully to the other clustering methods as well. After using a clustering technique to address interprocessor communication considerations, it is useful to apply a declustering procedure to gain flexibility for effective load balancing.

4.5. SCHEDULING RESULTS

To test the scheduling effectiveness of the declustering technique, we scheduled several digital signal processing algorithms onto a shared bus multiprocessor containing four DSP56001 processors. We invoked the declustering algorithm, Sarkar's internalization algorithm, and a modified version of Kim and Browne's linear clustering algorithm on each of these DSP applications to compare scheduling performance. The modification to the linear clustering algorithm was necessary because the method

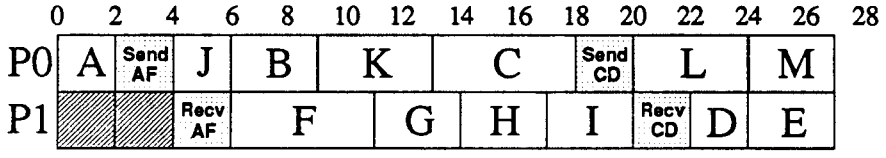


Figure 4-26. The schedule after cluster breakdown

used to assign clusters to processors was not described clearly enough for implementation. Our modified version, which we will refer to as the critical-path clustering algorithm, uses the linear clustering method to determine a set of clusters, but substitutes phases 2 and 3 of the declustering algorithm for processor assignment. We tested these scheduling techniques using four different signal processing algorithms: two sound synthesis programs, a telephone channel simulator, and a 16-QAM (quadrature amplitude modulation) transmitter. The schedule makespans in processor cycles are shown below in table 4-1 for each of these cases. The declustering technique produced the best result (shortest schedule length) in each instance.

Upon comparison of the times required to construct each schedule, as shown in table 4-2, it is readily seen that the critical-path clustering technique is the quickest and the internalization approach the slowest of these algorithms. If n represents the number of nodes and p represents the number of processors, the critical path algorithm has complexity $O[n^3p]$, while the declustering and internalization algorithms have complexity $O[n^3(n+p)]$.

These signal processing algorithms were all homogeneous graphs, meaning that each arc passes the same number of data units. For such graphs, the declustering algorithm

SCHEDULE LENGTHS IN PROCESSOR CYCLES			
DSP Algorithm(#nodes)	Declustering	C-P Clustering	Internalization
Sound Synthesis I (26)	173	179	218
Sound Synthesis II (27)	170	214	210
Telephone Channel Simulator (67)	297	297	488
QAM Transmitter (411)	4661	4881	5024

Table 4-1. Schedule lengths in processor cycles for 4 DSP algorithms

TIME REQUIRED FOR SCHEDULING IN SECONDS			
DSP Algorithm(#nodes)	Declustering	C-P Clustering	Internalization
Sound Synthesis I (26)	5.65	2.35	5.60
Sound Synthesis II (27)	3.16	1.72	4.34
Telephone Channel Simulator (67)	34.16	19.50	48.32
QAM Transmitter (411)	1523.20	265.72	5024.46

Table 4-2. Time required to construct a schedule (in seconds) for each scheduling technique. The algorithms were programmed in Lisp and run on a Sun3-60.

usually produces the same set of clusters as the critical-path clustering algorithm. Since the communication penalty in breaking any arc in the graph is exactly the same, the graph will almost always be broken in locations which expose the greatest amount of parallelism, namely the arcs output from a branch node or input to a merge node. To investigate the more interesting nonhomogeneous case, we randomly generated 100 APEGs containing between 70 and 140 nodes, where the node execution times were uniformly distributed over [4,20], and the number of data units assigned to each arc were uniformly distributed over [1,5]. These graphs were scheduled onto shared-bus architectures containing 4, 5, 6, 7, and 8 processors respectively, where the same send/receive communication protocol used in this paper was assumed. The declustering algorithm again demonstrated the best scheduling performance, and the average percentage speedup improvements (analogous to average percentage improvements in makespan) of the declustering algorithm over the two clustering techniques are shown in figure 4-27.

One reason that the declustering method performs better than the critical-path clustering algorithm is its superior clustering approach. The graph parallelism analysis techniques allow the declustering algorithm to break arcs not directly connected to branch or merge nodes, a capability which critical-path clustering lacks. The declustering

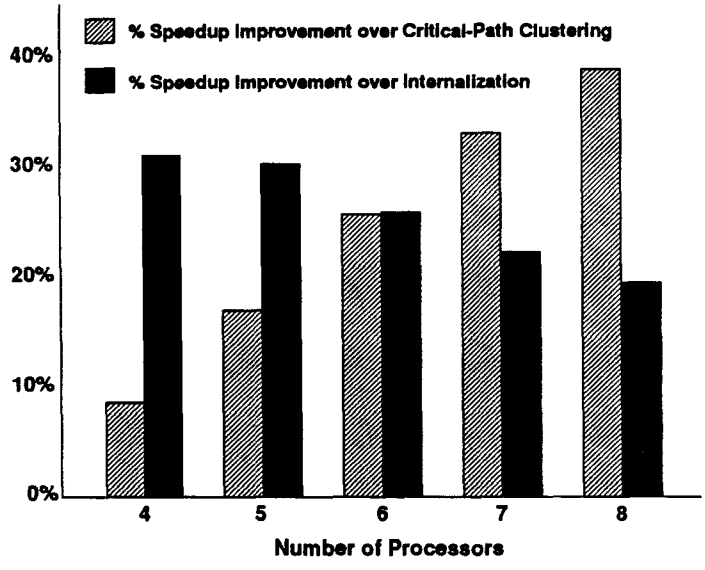


Figure 4-27. Percentage improvement in speedup of declustering over critical-path clustering and internalization

algorithm gains additional improvement by directly attacking the scheduling limiting progression rather than the critical path, because the two are often very different. The increasing speedup improvement as additional processors are added is primarily due to the enhanced load-balancing capability obtained through declustering. The graph-analysis clustering technique also surpasses the clustering performance of the internalization method. While the internalization procedure is more flexible than the critical-path clustering technique, it occasionally "overclusters" the graph by combining nodes which should remain separate. This effect is caused by the algorithm's inherent ambiguity in deciding the order in which arcs that pass the same number of data units should be considered. Lacking a global view of the graph, the internalization procedure often merges arcs in a suboptimal order.

To illustrate why the parallelism-analysis clustering technique in the first stage of the declustering algorithm outperforms the critical-path clustering and internalization

approaches, consider the graph example shown below in figure 4-28, which will be scheduled onto a two-processor shared-bus configuration. The declustering algorithm invokes the graph analysis techniques presented in section 3. After identifying nodes B and F as the immediate successors of branch node A, the procedure finds that cut-arc AB gives the minimum makespan solution for this Nbranch case. The algorithm proceeds to merge node I, identifies its immediate predecessors as nodes F and H, and determines that cutting arc HI yields the optimum solution in this Nmerge case. The elementary clusters obtained after removing these two cut-arcs and invoking a connectivity search are shown in figure 4-29. The resulting schedule with makespan 24 is shown in figure 4-30.

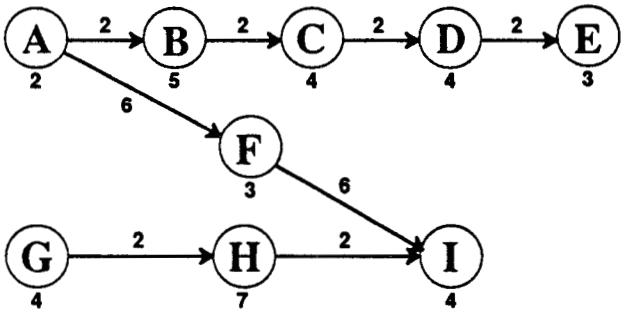


Figure 4-28. An example graph

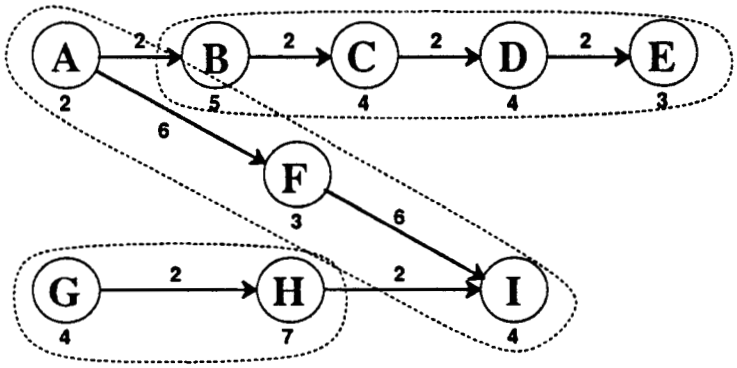


Figure 4-29. The declustering algorithm's elementary clusters

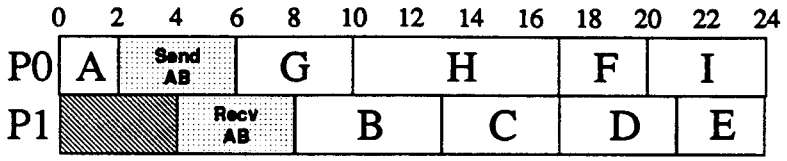


Figure 4-30. The declustering schedule

The critical-path clustering algorithm first identifies path {A-B-C-D-E} as the critical path with (computation + communication) length 42. After coalescing this path into a linear cluster and removing these nodes from the graph, it determines that path {G-H-I} is the critical path in the remaining portion of the graph. After placing node F into its own cluster, the final set of clusters is shown below in figure 4-31. The best placement of these clusters results in the schedule with makespan 30 shown in figure 4-32. It is immediately apparent that clustering the critical path can force a large communication to occur. Regardless of whether cluster F is grouped with cluster ABCDE or

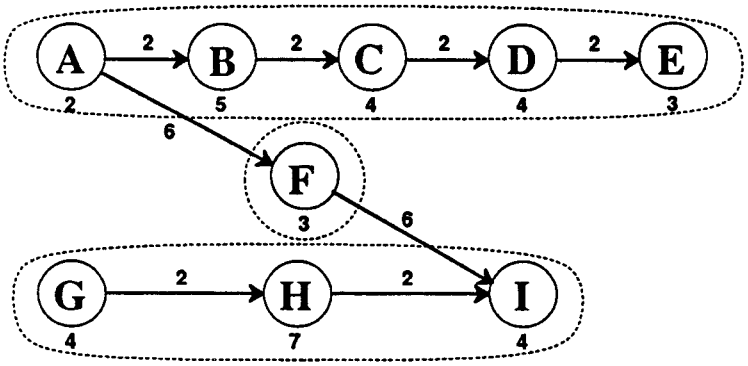


Figure 4-31. The clusters obtained through critical-path clustering

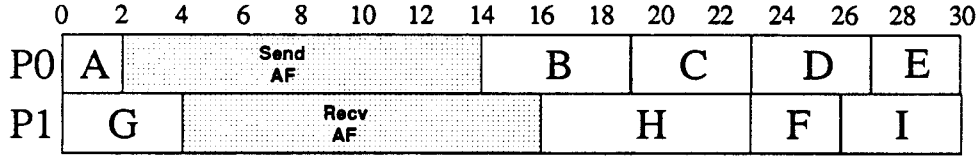


Figure 4-32. The critical-path clustering schedule

cluster GHI, this cluster composition forces a communication of 6 data units either between nodes A and F or nodes F and I. This phenomenon occurs because the algorithm treats each path as a complete entity, ignoring the fact that communication should weigh more heavily than computation for clustering purposes. In addition, clustering the entire critical path together may be unnecessary. Since the critical path is often different from the schedule limiting path, clustering only part of the critical path (e.g. BCDE) may lead to a more effective solution.

The internalization algorithm sidesteps this difficulty by considering merges at a lower level. It initially places each node in a separate cluster and sorts the arcs in decreasing order by the amount of data transferred. Since arcs AF and FI pass the greatest number of data units, the clustering steps [A + F = AF] and [AF + I = AFI] are immediately invoked, which avoids the data transfers of six data units. It then executes [AFI + B = ABFI], [ABFI + C = ABCFI], [ABCFI + D = ABCDFI], [ABCDFI + E = ABCDEFI], [G + H = GH], [ABFI + H = ABFHI], and [CDE + G = CDEG], to produce the clusters shown in figure 4-33. Using its modified list scheduling procedure for cluster placement, the internalization algorithm obtains the schedule

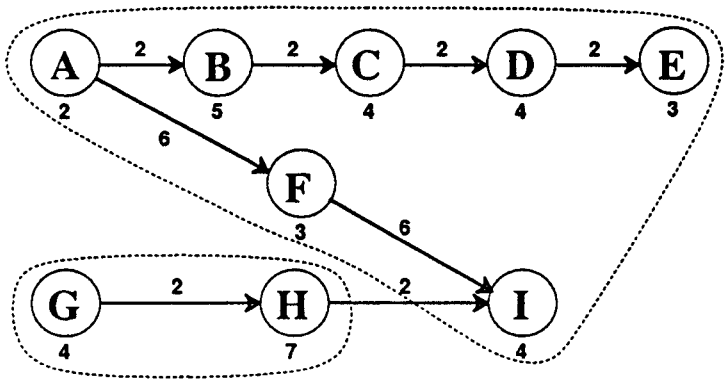


Figure 4-33. The clusters obtained through internalization

shown below in figure 4-34 which has makespan 29. Recalling that the algorithm accepts any cluster merging step which does not increase the parallel execution time estimate, we can immediately see that the ordering of arcs plays a major role in determining the clusters. In this example, there are 6 arcs which pass 2 data units each, and the order in which these arcs are considered wholly determines the cluster composition. The ordering {DE CD BC GH HI AB} produces the same clusters as the declustering technique, while the ordering {AB HI BC GH CD DE} produces clusters ABCFGHI and DE, which leads to a schedule with length 33. This ordering dependence reflects the local view exhibited by the internalization algorithm. By considering each arc for internalization individually, the algorithm loses its global perspective as to how each merge affects the total cluster structure.

A quick comparison of these schedules illustrates the global scheduling perspective taken by the declustering algorithm. Instead of immediately invoking the two initially executable nodes A and G on processors P0 and P1 respectively, the declustering algorithm maps both nodes onto P0 and idles P1 for four time units to obtain the optimal schedule.

Next, consider the graph shown in figure 4-35, which will be scheduled onto a 3 processor shared-bus topology. The declustering algorithm identifies nodes C and D as the immediate successors of branch node A and determines that arc AC is the optimal

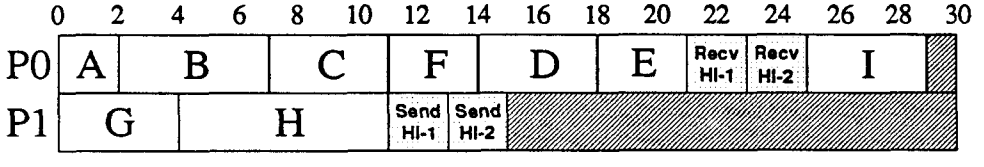


Figure 4-34. The internalization schedule

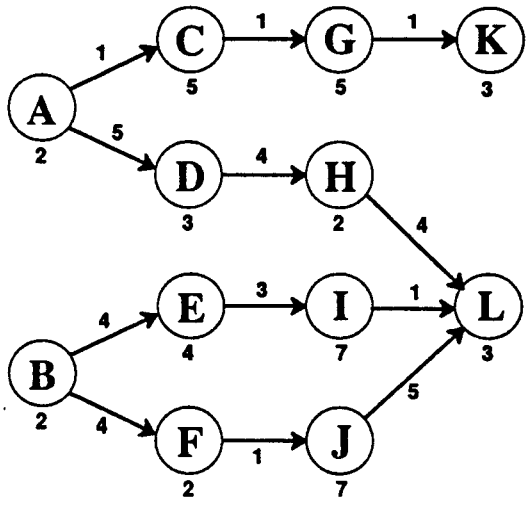


Figure 4-35. Another example graph

cut-arc for this Nbranch case. Arcs FJ and IL are subsequently identified as the two best cut-arcs for the Ibranch case involving nodes B, E, and F. Having exhausted the branch nodes, the algorithm turns to merge node L, which has three immediate predecessors H, I, and J. Since I and J have the highest values of $SLR(N_i)$, the procedure initially considers the Imerge parallelism instance involving nodes I, J, and L. Recognizing that this Imerge case was already examined when considering branch node B, the algorithm deletes node I from the list of immediate predecessors and proceeds to the Nmerge case involving nodes H, J, and L. This instance gives cut-arc FJ, which was already selected earlier. The elementary clusters are shown below in figure 4-36. Notice the irregular structure of clusters BEFI and ADHJL. To avoid excessive IPC cost, the algorithm suppresses some of the available parallelism in the graph by clustering together nodes which can be executed in parallel. This group of clusters produces the schedule shown in figure 4-37, which has makespan 24.

The critical-path clustering method iteratively clusters paths {A-D-H-L}, {B-E-I}, {C-G-K}, and {F-J} to obtain the set of clusters shown below in figure 4-38. The

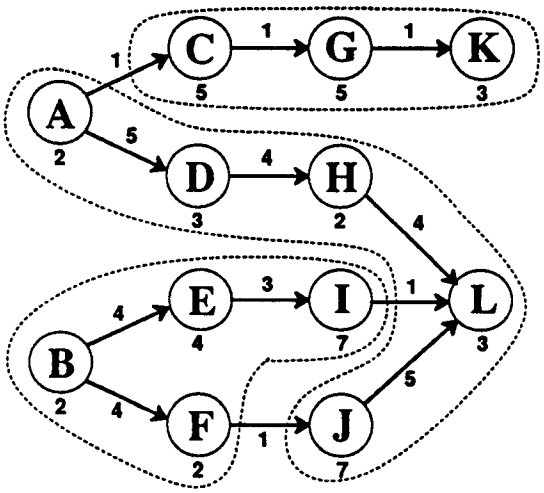


Figure 4-36. The declustering clusters

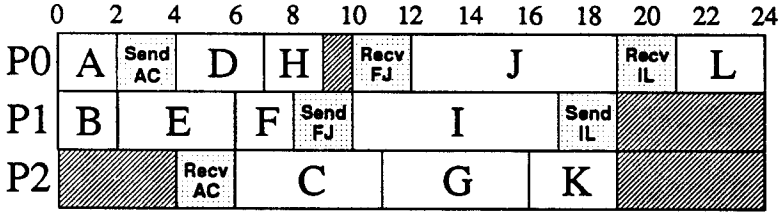


Figure 4-37. The declustering schedule

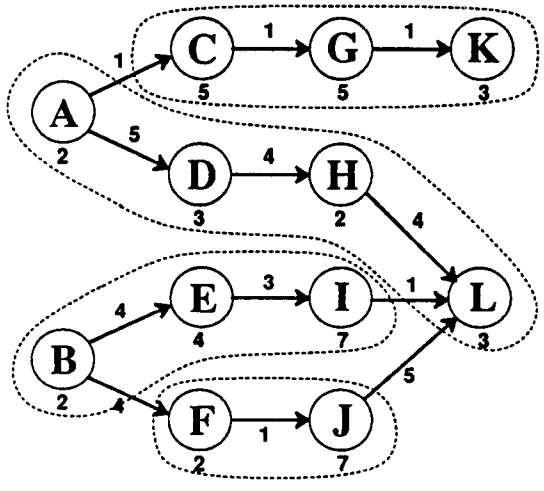


Figure 4-38. The critical-path clustering clusters

critical-path clustering schedule, shown in figure 4-39, has makespan 32. Unlike the

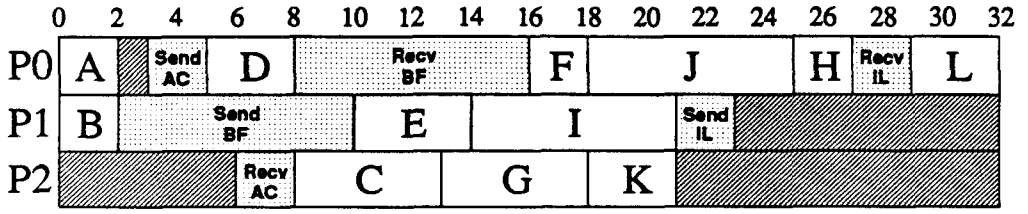


Figure 4-39. The critical-path clustering schedule

declustering algorithm, the technique of iteratively clustering linear paths cannot break arc FJ because it is not directly connected to a branch or merge node. It is therefore forced to break arcs BF and JL which transfer four and five units of data respectively. This inability to break arcs in the middle of a sequential string can cause large scheduling inefficiencies for nonhomogeneous graphs.

The internalization algorithm constructs the set of clusters shown below in figure 4-40, which produces the schedule shown in figure 4-41 with makespan 28. Although the internalization technique can construct irregular cluster shapes, it again illustrates a lack of global scheduling perspective by merging nodes A and C into the same clus-

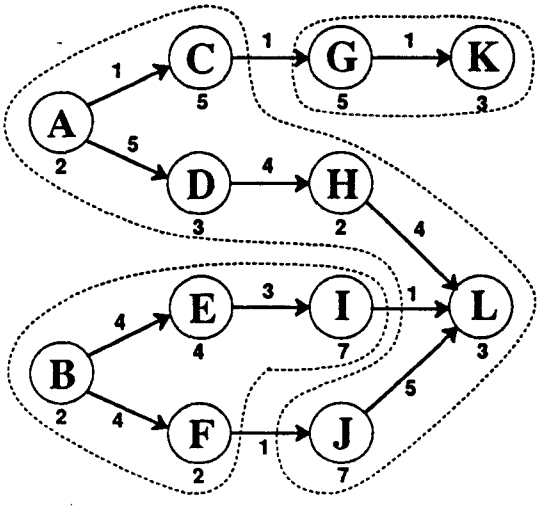


Figure 4-40. The internalization clusters

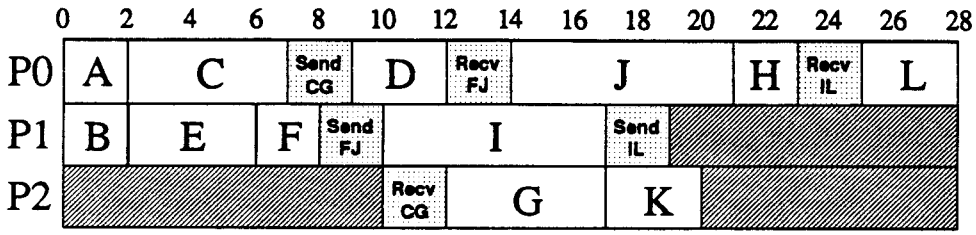


Figure 4-41. The internalization schedule

ter. Lacking the parallelism analysis techniques of the declustering method, the internalization algorithm does not see that merging arc CG instead of AC will expose a greater amount of graph parallelism.

4.6. SUMMARY AND CONCLUSIONS

We have introduced a new compile-time scheduling heuristic called declustering, which accounts for interprocessor communication costs and interconnection constraints within multiprocessor architectures. The first stage of this algorithm consists of a new clustering strategy which outperforms traditional clustering schemes by using novel parallelism analysis techniques to explicitly compare the tradeoff between parallelism exploitation and IPC cost. By systematically establishing and then decomposing a parallelism hierarchy, declustering exposes graph parallelism instances in order of importance and adjusts the level of cluster granularity to suit the characteristics of the specified architecture. So while it retains the ability to account for IPC, it also displays the flexibility necessary for effective load balancing and ensures efficient use of the available processors. The algorithm gains additional performance improvement by methodically attacking the schedule limiting progression, rather than the critical path. Due to the hierarchical nature of this approach, the algorithm scales well to

handle larger problems, especially for architectures which themselves possess a hierarchical structure.

While declustering is intended to target multiprocessors, we anticipate that message-passing multicomputers are also valid targets. The major difference is that whereas processors in a shared memory architecture are essentially equidistant for purposes of IPC, the distance between processors in a multicomputer varies according to the number of hops between processor locations. Although techniques such as virtual cut-through or wormhole routing lessen the importance of this distinction, the declustering algorithm can no doubt benefit from a technique which reassigns placements according to processor proximities.

5

FURTHER WORK

Research is the process of going up alleys to see if they are blind

— Marston Bates

The problem of scheduling with IPC overheads contains many issues which remain to be explored. This chapter touches on a few of these ideas, leaving them as suggestions for further research. The first section shows that several APEG realizations can be derived from the same data flow algorithmic description, and raises the question of finding which derivations are most suitable for certain scheduling methods. The second section discusses other approaches to the scheduling problem and poses other scheduling topics which need to be addressed in the same context. The third section proposes the subject of scheduling/routing interaction as an additional research topic.

5.1. APEG DERIVATION

Thus far, the scheduling process has started from the acyclic precedence expansion graph (APEG) description, with the implicit assumptions that this graph is immutable and scheduled only once. In actuality, most signal processing algorithms are run repeatedly on an infinite stream of data. We will assume that the schedules are blocked with blocking factor J, meaning that each node in the APEG must be scheduled exactly J times in the current cycle before scheduling any nodes in the next cycle. The *schedule period* $S_J(\phi)$ is defined to be the amount of time a schedule ϕ requires to complete a single cycle, where the cycle time includes the time required to execute each node exactly J times plus any time required for communication to begin the next cycle. The *iteration period* $T_J(\phi)$ for schedule ϕ is then defined

$$T_J(\phi) = \frac{S_J(\phi)}{J} \tag{5.1}$$

This normalized quantity, which indicates the average amount of time taken to execute each cycle, allows comparison of schedules with different blocking factors. In this new context, where the algorithm is run over and over, the scheduling goal is to minimize the iteration period.

We have been expressing DSP algorithms using synchronous data flow graphs, where the APEG is derived from the SDF graph using the algorithm given in Appendix I. These SDF graphs can contain directed loops if each loop contains enough logical delays (z^{-1} in signal processing) on its arcs to ensure deadlock avoidance. Again, each logical delay on an arc from N_i to N_j corresponds to an initial token in the input buffer of N_j coming from N_i. An interesting property of SDF graphs is that there may be several SDF representations which describe the same DSP algorithm. These

equivalent realizations can be obtained by shifting delays around (retiming), or adding delays to arcs which are not in any directed loops (pipelining). We can also increase the number of iterations of the graph in one periodic cycle (increasing blocking factor). Each of these representations generates a different APEG, some of which may be better suited for particular scheduling schemes than others. By using these techniques, we may be able to find an APEG which leads to a better scheduling result.

5.1.1. Increasing Blocking Factor

One technique which may reduce iteration period is increasing the blocking factor, that is, increasing the number of times each node in the APEG is executed in each cycle. Consider the data flow graph in figure 5-1, in which nodes A, B, and C are connected in a directed loop. The 6D on the arc from C to A indicates the presence of 6 logical delays, corresponding to 6 initial tokens in A's input buffer. Node A is thus initially executable, as is node C. Using the blocking factor J = 1 results in the schedule shown in the top chart in figure 5-2, where the arrows indicate communications required to start the next cycle. The schedule period $S_1(\phi)$ and iteration period $T_1(\phi)$ are both 12. The speedup is 1, indicating that the second processor is being wasted. If the blocking factor J is increased to 2, the schedule shown at the bottom of

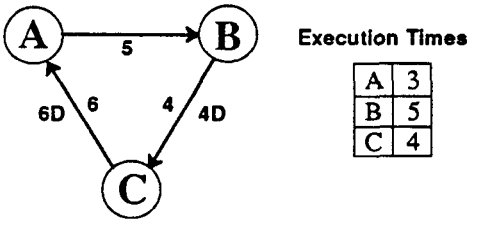


Figure 5-1. An example graph

figure 5-2 results. Here, the schedule period $S_2(\phi)$ is 12, the iteration period $T_2(\phi)$ is 6, and the speedup is 2, indicating an optimal schedule. Here we have assumed that the nodes have no state, so that successive invocations of the same node can be invoked without transmitting any data between them. The costs incurred by an increased blocking factor are that more memory is required in each processor to store the longer schedule, and that scheduling will take longer because more nodes are present in the APEG. To our knowledge, the problem of finding the optimal blocking factor is still open. One possible approach is to step through different blocking factors to see if a reduced iteration period is feasible.

An alternative to using blocked schedules is to employ schedules which overlap successive periods, such as the cyclo-static schedules of Schwartz [Swr85].

5.1.2. Retiming

Retiming was originally developed to minimize the clock period in synchronous circuits by changing the locations of registers [Lei83]. In the present context, we can use it to change the initially executable nodes by altering placement of the logical delays.

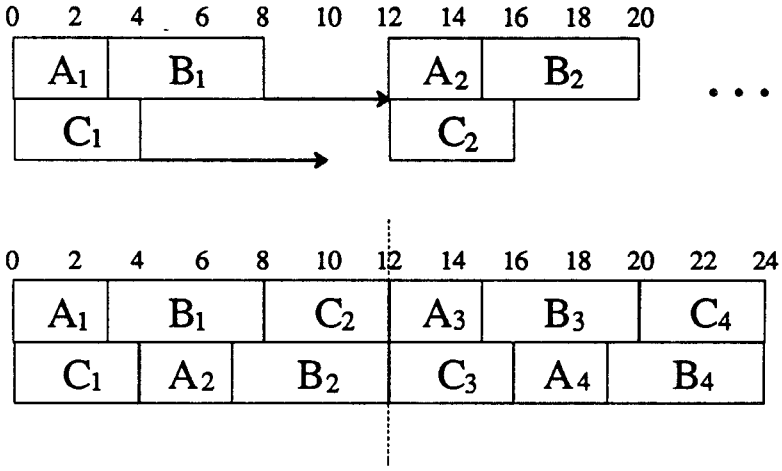


Figure 5-2. Schedules for blocking factor 1 and 2

Retiming can be thought of as a freedom to choose the initial condition, that is, permitting the choice of where time zero should be designated. By simply invoking a few nodes ahead of time before starting the scheduling process, an improved iteration period in the steady state may result. Consider the data flow graph G1 shown at the left of figure 5-3. The precedence graph, shown at the right of figure 5-3, indicates that nodes A and B are initially executable. Scheduling of this precedence expansion results in the disastrous schedule with iteration period 25 shown in figure 5-4. This poor schedule is a result of the extreme load imbalance, coupled with the communication of 8 time units needed to begin the next cycle. If nodes B and C are initially invoked before starting the scheduling process, the retimed graph G2 shown at the left of figure 5-5 is obtained, with its corresponding APEG shown at the right. Scheduling this graph leads to an iteration period of 18 as shown in figure 5-6. While the load is

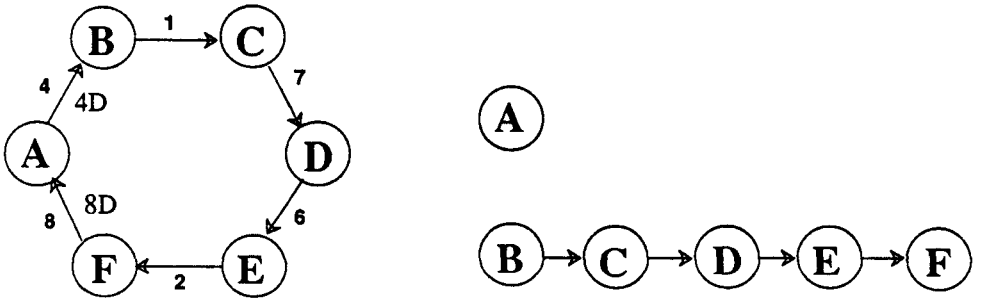


Figure 5-3. Data flow graph G1 and its precedence expansion

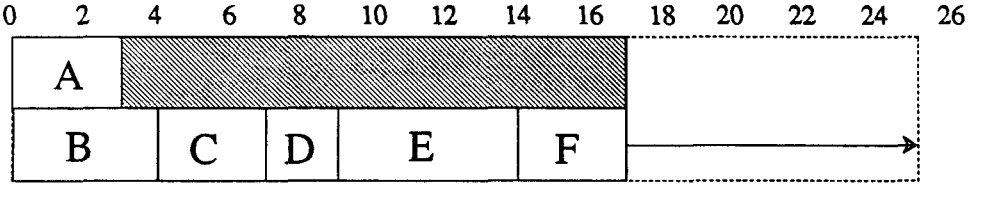


Figure 5-4. One iteration of the schedule for G1

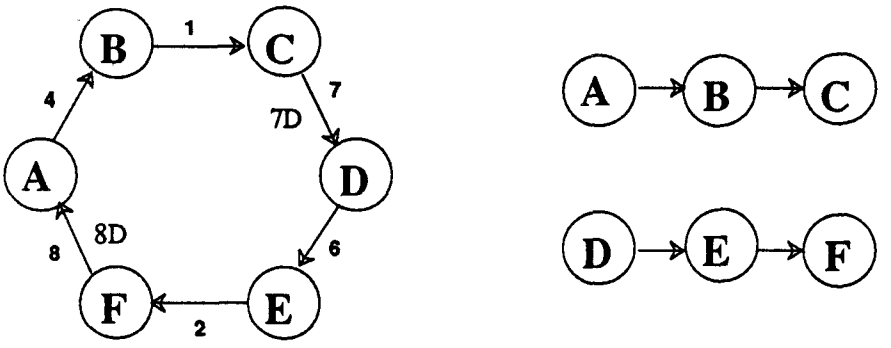


Figure 5-5. Data flow graph G2 and its precedence expansion

more evenly distributed, the communication penalty of 8 time units to begin the next iteration cycle is still present.

By initially invoking additional nodes, the retimed graph G3 and its derived APEG shown in figure 5-7 can be obtained. The schedule for this representation, shown in figure 5-8, has iteration period 12, because only 2 time units are required for

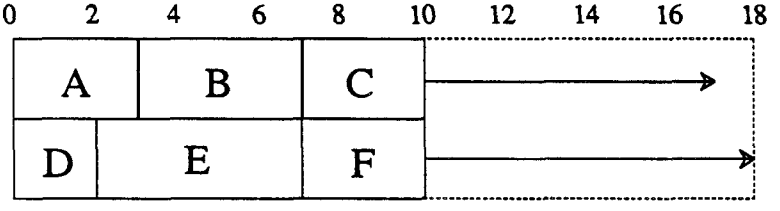


Figure 5-6. One iteration of the schedule for G2

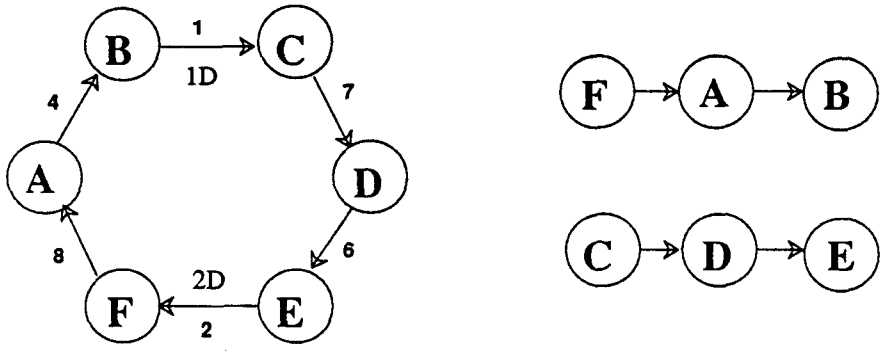


Figure 5-7. Data flow graph G3 and its precedence expansion

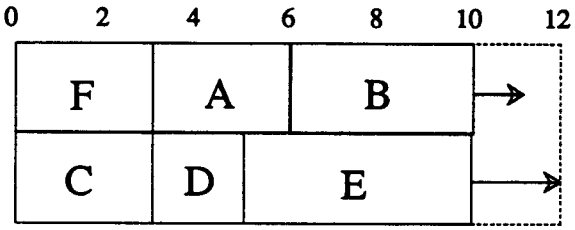
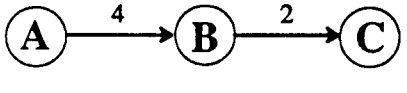


Figure 5-8. One iteration of the schedule for G3

communication to begin the next iteration cycle. In addition, by coupling this retimed graph with an increased blocking factor, one can spread this communication time over a greater number of cycles, making its effective contribution to the iteration period arbitrarily small. Since IPC costs are dependent on the architecture, a retiming algorithm will not be solely dependent on just the graph, but rather will vary with the architecture being targeted. Retiming could be used initially to minimize the critical path in the graph, or used in a schedule-analysis feedback routine to reduce the length of the schedule limiting progression.

5.1.3. Pipelining

Pipelining is a means of increasing throughput by exploitation of temporal parallelism. Similar to an assembly line, a task is split into subtasks, each of which is assigned dedicated hardware to form a pipeline stage. Each pipeline stage operates concurrently with other stages in the pipe, so that the execution of successive tasks is overlapped at the subtask level. Consider the simple linear graph shown in figure 5-9.



Execution Times

A	4
B	6
C	5

Figure 5-9. An example graph

In a classical pipelining approach, pipelining latches are placed between each pair of nodes, separating the string of nodes into 3 pipeline stages, as shown in figure 5-10. Different processors are then allocated to execute each pipeline stage, resulting in the schedule shown in figure 5-11. The iteration period of this schedule is 8, which gives a speedup of 15/8 = 1.875.

Under the synchronous data flow methodology, pipelining is performed by placing delays on each arc which is not part of a directed loop. For the linear graph example, the pipelined graph is shown in figure 5-12. This approach allows more freedom than the classical pipelining technique, because processors are not constrained to execute certain nodes. In the classical pipelining example, an increase in blocking factor doesn't accomplish anything, because node A is only executed on processor P_0, node

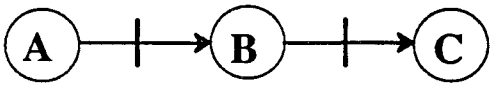


Figure 5-10. Classical pipelining approach

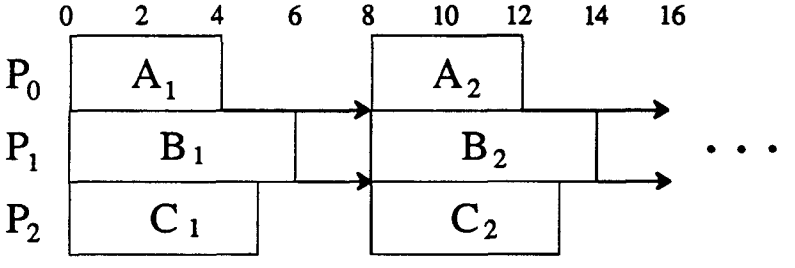


Figure 5-11. Classical pipeline schedule

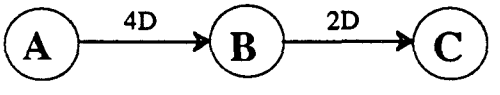


Figure 5-12. Pipelining by adding logical delays

B is only executed on processor P_1, and nodeC is only executed on processor P_2. However, the dynamic level scheduler allows a node to be naturally assigned to whichever processor can execute it most effectively, on the basis of dynamically changing priorities. Increasing the blocking factor to 3 results in the schedule shown in figure 5-13, which has a schedule period $S_3(\phi)$ of 15 and an iteration period $T_3(\phi)$ of 5 in the steady state (the pipeline is full). The associated speedup is 45/15 = 3, indicating an optimal schedule.

5.2. SCHEDULING PROBLEMS

There are a number of scheduling issues left to consider. The most pressing need is for an algorithm which combines the techniques of retiming, changing blocking factor, and pipelining into the current scheduling framework. The ability to modify the APEG being scheduled by changing the data flow description is powerful. It supplies an additional method of overcoming scheduling difficulties which would be especially effective when applied within an iterative scheduling approach.

While this thesis has presented a few schedule analysis techniques, there is still a great need to develop heuristic methods of analyzing a schedule to provide clues as to what

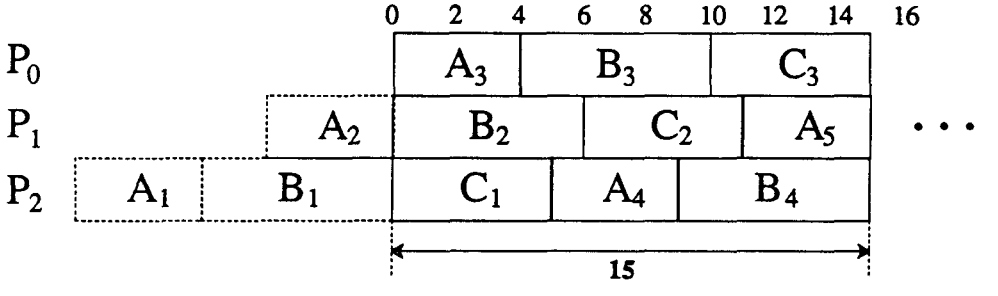


Figure 5-13. Dynamic level pipeline schedule

scheduling changes should be made in the next iteration. In addition to providing performance improvement, such techniques contribute to scheduling efficiency by identifying subsets of the search space of possible schedules which have a high probability of containing a faster schedule. An important question regarding iterative schemes is: "To what degree should worse solutions be accepted?" As mentioned in chapter 2, iterative schemes which only accept better solutions risk becoming trapped in local minima. On the other hand, the simulated annealing process is too slow to be used in many practical applications.

5.2.1. Other Scheduling Problems

There are a number of other interesting scheduling problems to consider.

Scheduling with Real-Time Constraints

Scheduling in the presence of real-time constraints is a slightly different problem than scheduling to minimize makespan. Once the real-time constraint is met, it may not be useful to schedule the graph any faster; a criterion to minimize memory buffer sizes or hardware cost might be more useful to pursue in this instance. When attempting to minimize hardware cost, one possible approach is to find a schedule that meets the real-time constraint and successively delete hardware components as long as the constraint is satisfied.

Scheduling for Massively Parallel Systems

Massively parallel systems using thousands or tens of thousands of processors such as the Connection Machine or MasPar MP-1 are gaining momentum in parallel computation. Whereas such machines were traditionally used for their cost/performance ratio,

they are now starting to provide the fastest performance for several computation intensive applications. Daniel Hillis, founder of Thinking Machines Corporation, estimates that by 1995, massively parallel machines will be about 100 times faster than conventional supercomputers [Hil90].

The problem of programming massively parallel machines from fine-grained precedence graphs is correspondingly gaining importance. These graphs may represent an algorithm broken into its elementary computations, or may be generated from a parallelizing compiler. In either case, the graph may contain tens or hundreds of thousands of nodes, which creates a formidable scheduling difficulty when one recalls the complexity of the scheduling problem. A hierarchical clustering scheme would be a reasonable first approach.

5.2.2. A Smart Scheduling System

Since it is highly doubtful that any single scheduling strategy can cope with the diversity of scheduling instances that might be encountered, it would be useful to collect a library of scheduling strategies, each of which performs effectively for a certain class of applications. The existence of such a library raises the possibility of using a "smart" supervisor to oversee their use. When confronted with a scheduling instance, this scheduling manager would invoke several information-gathering routines on the precedence graph and specified architecture to isolate characteristics to help determine the appropriate scheduling technique or subset of techniques. A block diagram of such a system is shown below in figure 5-14. The user first inputs the precedence graph describing the desired algorithm and the processor architecture specification. The scheduling system extracts several parameters from the graph which capture its

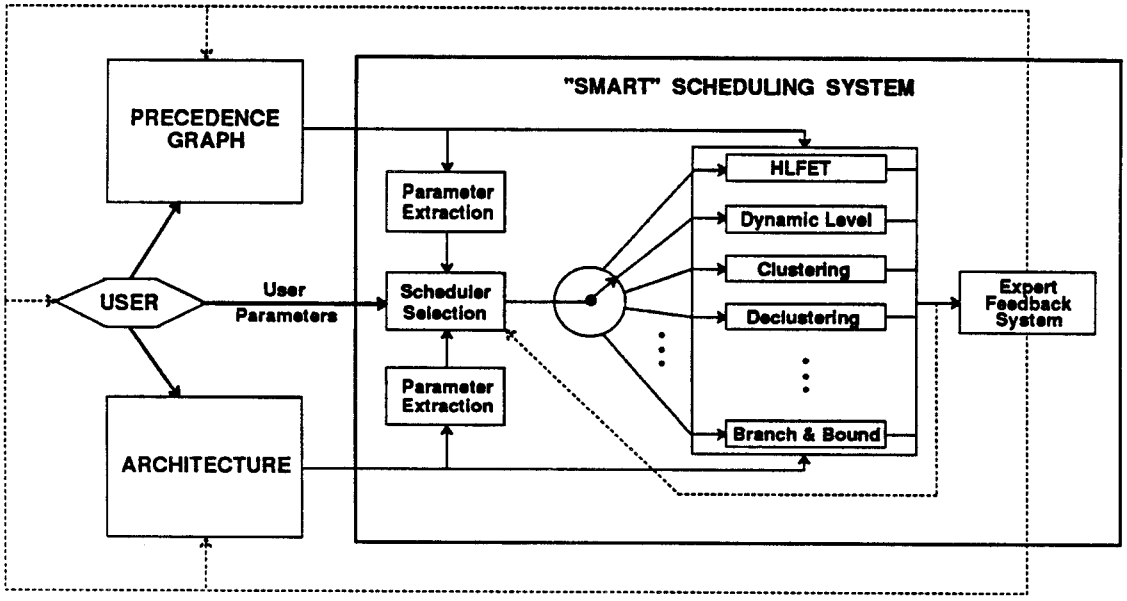


Figure 5-14. Diagram of a "smart" scheduler

characteristics such as its size (number of nodes), its density (ratio of number or arcs to number of nodes), its shape (strongly connected components), the mean and variance of the node execution times, and perhaps the average indegree and outdegree of the nodes. The system also extracts architectural parameters such as the number of processors, the interprocessor communication bandwidth, the number of nearest neighbors for each processor, and the routing scheme being employed. The grain size of the nodes in the graph (relative size of node execution times to interprocessor communication times) is another important consideration. The user can also enter parameters, such as a real-time constraint on the DSP algorithm, or a timeframe in which feedback information is desired. Using these input parameters, a scheduling selector chooses an appropriate technique or group of techniques for scheduling.

After scheduling the precedence graph, the system can feed back the standard scheduling results (makespan, processor efficiency, time spent in IPC) to the scheduler selec-

tor to help modify parameters of the current scheduling scheme, or select another technique altogether. An "expert" (rule-based) system could also supply feedback information to the user, suggesting possible changes in the graph and/or the architecture. For example, it might indicate possible retimings of the graph, or point out that pipelining certain feedforward arcs would be useful in creating more parallelism. If the graph description itself is hierarchical, it might suggest that certain large-grain nodes be broken down one level of granularity to create additional flexibility for load-balancing. On the architecture side, the system might indicate that there are too many or too few processors, or it might determine that resource contention is the main problem and suggest an architecture that allows more simultaneous communication between processors.

5.3. SCHEDULING-ROUTING INTERACTION

Another possible avenue of investigation concerns the relationship between the intertwined scheduling and routing functions. Since previous communication resource reservations may block a node from being scheduled on a certain processor, the rerouting of data transfer paths may facilitate a better node to processor mapping. While many researchers have acknowledged the importance of this topic, it has been neglected in the literature. The relative importance of the mapping functions and the communication scheduling routines is still unknown. This thesis concentrates on the scheduling functions, using very simple routing schemes. However, for some scheduling instances, the communication routing may have more effect on performance, so that incorporating adaptive routing techniques may be worthwhile.

REFERENCES

[Ada74].

T.L. Adam, K.M. Chandy, and J.R. Dickson, "A Comparison of List Schedules for Parallel Processing Systems," *Communications of the ACM* 17(12) pp. 685-690 (December 1974).

[Adam87].

G.B. Adams III, D.P. Agrawal, and H.J. Siegel, "A Survey and Comparison of Fault-Tolerant Multistage Interconnection Networks," *Computer*, pp. 15-27 (June, 1987).

[Alm89].

G.S. Almasi and A. Gottlieb, *Highly Parallel Computing*, Benjamin/Cummings Publishing Company, Redwood City, CA (1989).

[Amd67].

G.M. Amdahl, "Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities," *AFIPS Conference Proceedings* 30 pp. 483-485 (1967).

[And85].

F. Andre, D. Herman, and J.P. Verjus, *Synchronization of Parallel Programs*, MIT Press, Cambridge, Massachusetts (1985).

[Arl88].

R. Arlauskas, "iPSC/2 System: A Second Generation Hypercube," *The Third Conference on Hypercube Concurrent Computers and Applications* 1(January, 1988).

[Bac78].

J. Backus, "Can Programming Be Liberated from the von Neumann Style? A Functional Style and Its Algebra of Programs," *Communications of the ACM* 21(8) pp. 613-641 (August, 1978).

[Bat68].

K.E. Batcher, "Sorting Networks and Their Applications," *Proceedings 1968 Spring JCC* 32 pp. 307-314 (1968).

[Bat80].

K.E. Batcher, "Design of a Massively Parallel Processor," *IEEE Transactions on Computers* C-29(9) pp. 836-840 (September, 1980).

[Ben62].

V.E. Benes, "On Rearrangeable Three-Stage Connecting Networks," *Bell System Technical Journal* 41(5) pp. 1481-1492 (September, 1962).

[Ben65].

V.E. Benes, *Mathematical Theory of Connecting Networks*, Academic Press, New York (1965).

[Bhu89].

L.N. Bhuyan, Q. Yang, and D.P. Agrawal, "Performance of Multiprocessor Interconnection Networks," *Computer*, pp. 25-37 (February, 1989).

[Bia87].

R.P. Bianchini Jr. and J.P. Shen, "Interprocessor Traffic Scheduling Algorithm for Multiple-Processor Networks," *IEEE Transactions on Computers* C-36(4) pp. 396-409 (April 1987).

[Bie90].

J.C. Bier, E.E. Goei, W.H. Ho, P.D. Lapsley, M.P. O'Reilly, G.C. Sih, and E.A. Lee, "Gabriel: A Design Environment for DSP," *IEEE Micro* 10(5) pp. 28-45 (October, 1990).

[Bir90].

J.C. Bier, S. Sriram, and E.A. Lee, "A Class of Multiprocessor Architectures for Real-Time DSP," in *VLSI DSP IV*, IEEE Press, New York (1990).

[Bok81].

S.H. Bokhari, "A Shortest Tree Algorithm for Optimal Assignments Across Space and Time in a Distributed Processor System," *IEEE Transactions on Computers* **SE-7**(6) pp. 583-589 (November, 1981).

[Bol88].

S.W. Bollinger and S.F. Midkiff, "Processor and Link Assignment in Multicomputers using simulated annealing," *International Conference on Parallel Processing* **1** pp. 1-7 (August, 1988).

[Bou72].

W.J. Bouknight, S.A. Denenberg, D.E. McIntyre, J.M. Randall, A.H. Sameh, and D.L. Slotnick, "The Illiac IV System," *Proceedings of the IEEE* **60**(4) pp. 369-379 (April, 1972).

[Bur46].

A.W. Burks, H.H. Goldstine, and J. von Neumann, "Preliminary Discussion of the Logical Design of an Electronic Computing Instrument," *IAS Report*, (June, 1946).

[Bus74].

B. Bussell, E. Fernandez, and O. Levy, "Optimal Scheduling for Homogeneous Multiprocessors," *Proceedings IFIP Congress*, pp. 286-290 (1974).

[Che81].

P.Y. Chen, D.H. Lawrie, P.C. Yew, and D.A. Padua, "Interconnection Networks Using Shuffles," *Computer*, pp. 55-64 (December, 1981).

[Chr41].

A. Church, *The Calculi of Lambda-Conversion*, Princeton Univ. Press, Princeton, NJ (1941).

[Chu80].

W.W. Chu, L.J. Holloway, M.T. Lan, and K. Efe, "Task Allocation in Distributed Data Processing," *Computer*, pp. 57-69 (November 1980).

[Chu87].

W.W. Chu and L.M.T. Lan, "Task Allocation and Precedence Relations for Distributed Real-Time Systems," *IEEE Transactions on Computers* C-36(6) pp. 667-679 (June 1987).

[Clo53].

C. Clos, "A Study of Non-Blocking Switching Networks," *Bell System Technical Journal* 32 pp. 406-424 (March, 1953).

[Cof73].

E.G. Coffman, Jr. and P.J. Denning, *Operating Systems Theory*, Prentice Hall, Englewood Cliffs, NJ (1973).

[Cof76].

E.G. Coffman Jr., Editor, *Computer and Job Shop Scheduling Theory*, John Wiley and Sons, New York, NY (1976).

[Con67].

R.W. Conway, W.L. Maxwell, and L.W. Miller, *Theory of Scheduling*, Addison-Wesley, Reading, Mass. (1967).

[Dal86].

W.J. Dally and C.L. Seitz, "The Torus Routing Chip," *Journal of Distributed Computing* 1(3)(1986).

[Dal87].

W.J. Dally, "Wire-Efficient VLSI Multiprocessor Communication Networks," *Proceedings 1987 Stanford Conference on VLSI*, pp. 391-415 (1987).

[Dij68].

E.W. Dijkstra, "Co-operating Sequential Processes," pp. 43-112 in *Programming Languages, F. Genuys ed.*, Academic Press, New York (1968).

[Edm72].

J. Edmonds and R.M. Karp, "Theoretical Improvements in Algorithm Efficiency for Network Flow Problems," *Journal of the ACM* 19 pp. 248-264 (April, 1972).

[Fen81].

T.Y. Feng, "A Survey of Interconnection Networks," *Computer*, pp. 12-27 (1981).

[Fly72].

M.J. Flynn, "Some Computer Organizations and their Effectiveness," *IEEE Transactions on Computers* C-21(9) pp. 948-960 (Sept. 1972).

[Fre82].

S. French, *Sequencing and Scheduling: An Introduction to the Mathematics of the Job-Shop*, Ellis Horwood, New York (1982).

[Gar78].

M.R. Garey, R.L. Graham, and D.S. Johnson, "Performance Guarantees for Scheduling Algorithms," *Operations Research* 26(1) pp. 3-21 (January 1978).

[Gar79].

M.R. Garey and D.S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*, W.H. Freeman and Co., New York, NY (1979).

[Gok73].

L.R. Goke and G.J. Lipovski, "Banyan Networks for Partitioning Multiprocessor Systems," *Proceedings of the First Annual Symposium on Computer Architecture*, pp. 21-28 (1973).

[Gon77].

M.J. Gonzalez, Jr., "Deterministic Processor Scheduling," *Computing Surveys* 9(3)(September, 1977).

[Goo88].

J.R. Goodman and P.J. Woest, "The Wisconsin Multicube: A New Large-Scale Cache-Coherent Multiprocessor," *International Symposium on Computer Archtecture*, pp. 422-431 (June, 1988).

[Gra69].

R.L. Graham, "Bounds on Multiprocessing Timing Anomalies," *SIAM Journal on Applied Mathematics* 17(2) pp. 416-429 (March, 1969).

[Gre87].

B. Greenblatt and C.J. Linn, "Branch and Bound Algorithms for Scheduling Communicating Tasks in a Distributed System," *Compcon*, pp. 12-16 (1987).

[Gup89].

R. Gupta, "The Fuzzy Barrier: A Mechanism for High Speed Synchronization of Processors," *ASPLOS-III Proceedings*, pp. 54-63 (April, 1989).

[Gyl76].

V.B. Gylys and J.A. Edwards, "Optimal Partitioning of Workload for Distributed Systems," *COMPCON*, pp. 353-357 (Fall, 1976).

[Haj85].

B. Hajek, "A Tutorial Survey of Theory and Applications of Simulated Annealing," *Proceedings of the 24th Conference on Decision and Control*, pp. 755-760 (December 1985).

[Hil90].

W.D. Hillis, "Keynote Address," *Supercomputing '90*, (December 1990).

[Hor80].

J. Horowitz, *Critical Path Scheduling : Management Control Through CPM and PERT*, R.E. Krieger, Huntington, NY (1980).

[Hu61].

T.C. Hu, "Parallel Sequencing and Assembly Line Problems," *Operations Research* 9(6) pp. 841-848 (November 1961).

[Kar88].

R.M. Karp and V. Ramachandran, "A Survey of Parallel Algorithms for Shared-Memory Machines," *Report No. UCB/CSD 88/408*, (March, 1988).

[Ker79].

P. Kermani and L. Kleinrock, "Virtual Cut-Through: A New Computer Communication Switching Technique," *Computer Networks* 3 pp. 267-286 (September, 1979).

[Kim88].

S.J. Kim and J.C. Browne, "A General Approach to Mapping of Parallel Computations upon Multiprocessor Architectures," *Int. Conf. on Parallel Processing* 3 pp. 1-8 (August, 1988).

[Kir83].

S. Kirkpatrick, C.D. Gelatt, Jr., and M.P. Vecchi, "Optimization by Simulated Annealing," *Science* 220(4598) pp. 671-680 (May 1983).

[Koh75].

W.H. Kohler, "A Preliminary Evaluation of the Critical Path Method for Scheduling Tasks on Multiprocessor Systems," *IEEE Transactions on Computers*, pp. 1235-1238 (December, 1975).

[Kru83].

C.P. Kruskal and M. Snir, "The Performance of Multistage Interconnection Networks for Multiprocessors," *IEEE Transactions on Computers* C-32(12) pp. 1091-1098 (December, 1983).

[Kuc81].

D.J. Kuck, R.H. Kuhn, D.A. Padua, B. Leasure, and M. Wolfe, "Dependence Graphs and Compiler Optimization," *Symp. on Principles of Programming Languages*, pp. 207-218 (Jan. 1981).

[Kum87].

V.P. Kumar and S.M. Reddy, "Augmented Shuffle-Exchange Multistage Interconnection Networks," *Computer*, pp. 30-40 (June, 1987).

[Law75].

D.H. Lawrie, "Access and Alignment of Data in an Array Processor," *IEEE Transactions on Computers* C-24(12) pp. 1145-1155 (December, 1975).

[Lee87].

E.A. Lee and D.G. Messerschmitt, "Static Scheduling of Synchronous Data Flow Programs for Digital Signal Processing," *IEEE Transactions on Computers* C-36(2)(January, 1987).

[Lee89].

E.A. Lee and S. Ha, "Scheduling Strategies for Multiprocessor Real-Time DSP," *Globecom*, (November, 1989).

[Lei83].

F.M. Rose and J.B. Saxe, "Optimizing Synchronous Circuitry by Retiming," *Proceedings of the 3rd Caltech Conference on VLSI*, (March, 1983).

[Lo88].

V.M. Lo, "Heuristic Algorithms for Task Assignment in Distributed Systems," *IEEE Transactions on Computers* 37(11) pp. 1384-1397 (November, 1988).

[Mud87].

T.N. Mudge, J.P. Hayes, and D.C. Winsor, "Multiple Bus Architectures," *Computer*, pp. 42-48 (June, 1987).

[Nug88].

S.F. Nugent, "The iPSC/s Direct-Connect Communications Technology," *The Third Conference on Hypercube Concurrent Computers and Applications* 1(January, 1988).

[Pat81].

J.H. Patel, "Performance of Processor-Memory Interconnections for Multiprocessors," *IEEE Transactions on Computers* C-30(10) pp. 771-780 (October, 1981).

[Pfi85].

G.F. Pfister and V.A. Norton, " "Hot-spot" Contention and Combining in Multistage Interconnection Networks," *IEEE Transactions on Computers* C-34 pp. 943-948 (October, 1985).

[Ram72].

C. V. Ramamoorthy, K.M. Chandy, and M.J. Gonzalez, "Optimal Scheduling Strategies in a Multiprocessor System," *IEEE Transactions on Computers* C-21(2) pp. 137-146 (February 1972).

[Ray86].

M. Raynal, *Algorithms for Mutual Exclusion*, MIT Press, Cambridge, MA (1986).

[Ree87].

D.A. Reed and D.C. Grunwald, "The Performance of Multicomputer Interconnection Networks," *Computer*, pp. 63-73 (June, 1987).

[Ros82].

J.B. Rosser, "Highlights of the History of the Lambda-Calculus," *Proceedings of the ACM Symposium on LISP and Functional Programming*, pp. 216-225 (August, 1982).

[Sar89].

V. Sarkar, *Partitioning and Scheduling Parallel Programs for Multiprocessors*, MIT Press, Cambridge, MA (1989).

[Sch80].

J.T. Schwartz, "Ultracomputers," *ACM Transactions on Programming Languages and Systems* 2(4) pp. 484-521 (October, 1980).

[Sco89].

S.L. Scott and G.S. Sohl, "Using Feedback to Control Tree Saturation in Multistage Interconnection Networks," *IEEE 16th Annual Symposium on Computer Architecture*, pp. 167-176 (1989).

[Sto71].

H.S. Stone, "Parallel Processing with the Perfect Shuffle," *IEEE Transactions on Computers* C-20(2) pp. 153-161 (February, 1971).

[Sto77].

H.S. Stone, "Multiprocessor Scheduling with the Aid of Network Flow Algorithms," *IEEE Transactions on Computers* SE-3(1) pp. 85-93 (January, 1977).

[Swr85].

D.A. Schwartz, "Synchronous Multiprocessor Realizations of Shift-Invariant Flow Graphs," *Ph.D. Thesis, Georgia Institute of Technology*, (June, 1985).

[Ull73].

J.D. Ullman, "Polynomial Complete Scheduling Problems," *4th Symposium on Operating System Principles*, pp. 96-101 (1973).

[Win88].

D.C. Winsor and T.N. Mudge, "Analysis of Bus Hierarchies for Multiprocessors," *IEEE Symposium on Computer Architecture*, pp. 100-107 (1988).

[Wul72].

W.A. Wulf and C.G. Bell, "C.mmp--A Multimicroprocessor," *AFIPS Conference Proceedings* **41** pp. 765-777 (1972).

[Yew87].

P.C. Yew, N.F. Tzeng, and D.H. Lawrie, "Distributing Hot-Spot Addressing in Large-Scale Multiprocessors," *IEEE Transactions on Computers* C-36(4)(April, 1987).

[Yu84].

W.H. Yu, "LU Decomposition on a Multiprocessing System with Communication Delay," *Ph.D. Thesis, UC-Berkeley*, (1984).

APPENDIX I
SDF TO APEG GRAPH EXPANSION

This appendix describes the algorithm for expanding a synchronous data flow (SDF) graph into an acyclic precedence expansion graph (APEG). We first introduce some formalism which is described in greater detail in [Lee87]. A synchronous data flow graph can be represented by a **topology matrix** Γ, which represents its structure. This matrix is constructed by assigning a column to each node and a row to each arc, so that the (i,j)th entry represents the amount of data injected into the FIFO buffer on arc i by node j on each invocation. This number is negative if node j consumes data from arc i. For example, the SDF graph shown in figure A-1 has topology matrix

$$\Gamma = \begin{bmatrix} 2 & -3 & 0 \\ 0 & 1 & -2 \\ -1 & 0 & 3 \end{bmatrix}$$

where the arcs labeled 1, 2, and 3 correspond with the rows of the matrix going from top to bottom, and nodes A, B, and C correspond to the columns going from left to right respectfully.

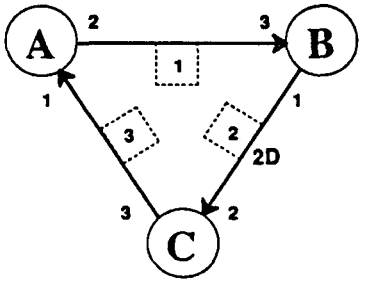


Figure A-1. An example SDF graph

A SDF graph with s nodes which has consistent sample rates is guaranteed to have $rank(\Gamma) = s - 1$, which ensures that the topology matrix has a nullspace [Lee87]. This fact is important because the smallest integer vector q in the nullspace of Γ ($\Gamma q = \phi$), indicates the number of invocations of each node in each complete iteration of the graph. The topology matrix above has integer nullspace vector

$$q = \begin{bmatrix} 3 \\ 2 \\ 1 \end{bmatrix}$$

indicating that A is invoked three times, B twice, and C once in each invocation.

For simplicity, we first concentrate on expanding the single arc from A to B in figure A-1. Vector q indicates that there are three invocations of A (A_1, A_2, A_3) and two invocations of B (B_1, B_2) in the expansion, which is shown in figure A-2. The SDF arc from A to B has no delays on it, indicating that the arc buffer is initially empty. The first data sample produced by A_1 is therefore the first sample consumed by B_1, and the second sample produced by A_1 is the second sample consumed by B_1. However, the third sample consumed by B_1 is the first sample produced by A_2, because each invocation of A only produces two samples. The rest of the expansion follows in a straightforward fashion.

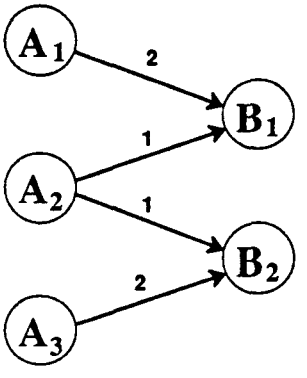


Figure A-2. The expansion for the arc between nodes A and B

The situation becomes more complicated if the SDF arc has delays on it, as shown in figure A-3. If the arc has n logical delays, the n data samples initially present in the arc's FIFO data buffer are the first data units to be consumed by the instances of B. For example, in part a) of figure A-3, the SDF arc has a single delay, so the first sample consumed by B_1 is this initial data sample, and the first data sample produced by A_1 is the *second* data unit consumed by B_2. The other data assignments follow

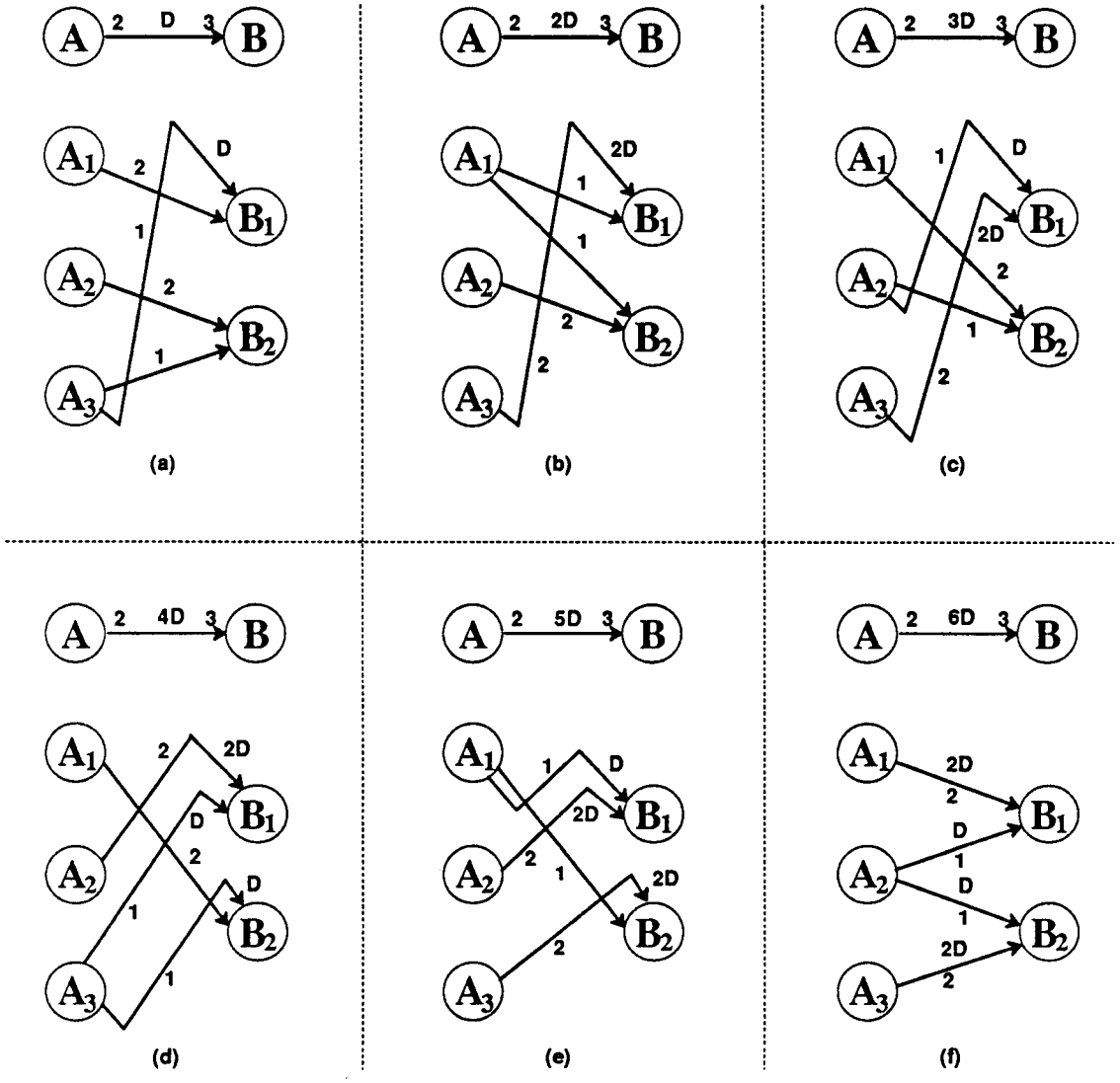


Figure A-3. Graph expansions for different numbers of delays

accordingly by stepping through instances of A and B in order, switching invocations as necessary. Similarly, the two delays on the SDF arc in part b) indicate that the first sample produced by A_1 is the third sample consumed by B_1, and the three delays on the arc in part c) indicate that the first sample produced by A_1 is the first sample consumed by B_2. As we increase the number of delays, this process continues until the situation in part f), where the data assignments are the same as in figure A-2, except that each of the data units consumed by B_1 and B_2 come from the FIFO buffer.

To formalize this arc expansion, consider the single arc between nodes A and B shown in figure A-4, which we assume to be part of a larger SDF graph. The notation A_o represents the number of data samples that node A produces on the arc in each invocation, and B_i is the number of data units that node B consumes from the arc on each invocation. The notation nD indicates the presence of n logical delays on the arc. We let I_A and I_B represent the number of invocations of nodes A and B respectively in each iteration of the graph, as given by the smallest integer vector q in the nullspace of the topology matrix. The first sample produced by node A_1 goes to sample number B_{sample} of instance number $B_{instance}$ of B, where

$$B_{sample} = [(n \ \ mod \ \ I_b B_i) \ \ mod \ \ B_i] + 1,$$

and

$$B_{instance} = [(n \ \ mod \ \ I_b B_i) \ \ div \ \ B_i] + 1.$$

The other data assignments are made by systematically stepping through samples of

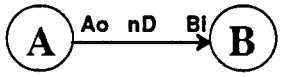


Figure A-4. An example SDF arc from tailnode A to headnode B

instances of A and B, switching invocations as necessary. The **expanded graph** is obtained by repeating this procedure for each arc in the graph. To derive the precedence graph from the expanded graph, we must identify the **initially runnable** nodes in the expanded graph. We say a node is initially runnable, if for each arc entering the node, the number of initial data units on the arc (number of logical delays on the arc) equals or exceeds the number of data units required to invoke the node. The APEG is then derived by breaking the arcs entering the initially runnable nodes.

The algorithm to derive an APEG from a SDF graph can be stated as follows:

1) Construct the topology matrix Γ.
2) Find the smallest integer vector q in the nullspace of Γ
3) Expand each arc using the procedure described above.
4) Identify the initially runnable nodes.
5) Break the arcs heading into initially runnable nodes.

APPENDIX II
RANDOM GRAPH GENERATION

The algorithm used to generate random graphs is shown below in a pascal-like syntax.

```
global variable : endnodes

Function Random_Graph (number_start_nodes graph_length)
 begin
        (* Make a list of initially runnable nodes *)
        For i:= 1 to number_start_nodes do
          begin
                newnode := Create_node(exec_mean, exec_var)
                Add newnode to endnodes list
          end

        iterations := number_start_nodes x graph_length
        For i:= 1 to iterations do
          begin
                (* Choose a random integer in [0, 100] to select an action *)
                action_number := Random_Integer(100)

                (* Connect a newly-created node to one of the endnodes *)
                if {action_number < 40} then
                        Extend_Node { Pick_Randomly(endnodes 1) 1 }

                (* Cause some endnodes to all converge to a single node *)
                if {40 < action_number <= 60} then
                        conv_number := Random_Integer(k)
                        conv_nodes := Pick_Randomly(endnodes conv_number)
                        Converge(conv_nodes)

                (* Have a randomly chosen endnode diverge out to several newly-created nodes *)
                if {60 < action_number <= 80} then
                        Diverge { Pick_Randomly(endnodes 1), Random_Integer(m) }

                (* Attach a structure which diverges and then converges to an endnode *)
                if {80 < action_number <= 90} then
                        Diverge_Converge{ Pick_Randomly(endnodes 1) }

                (* Make a random connection *)
                if {action_number > 90} then
                        Random_Connection()
          end
 end

Function Extend_Node (old_endnode extension_length)
 begin
        For i := 1 to extension_length
          begin
                new_endnode := Create_Node(exec_mean exec_var)
                Connect_Nodes(old_endnode new_endnode data_mean data_var)
                Delete old_endnode from endnodes list
                Add new_endnode to endnodes list
                old_endnode := new_endnode
          end
        Return new_endnode
 end
```

```
Function Converge (list_of_nodes_to_converge)
 begin
        new_endnode := Create_Node(exec_mean exec_var)

        For each endnode in list_of_nodes_to_converge do
          begin
                Delete endnode from endnodes list
                Connect_Nodes (endnode new_endnode data_mean data_var)
          end
        Add new_endnode to endnodes list
 end

Function Diverge (old_endnode number_to_diverge)
 begin
        For i := 1 to number_to_diverge
          begin
                new_node := Create_Node (exec_mean exec_var)
                Connect_Nodes (old_endnode new_node data_mean data_var)
                Add new_node to endnodes list
                Add new_node to list_of_diverged_nodes
          end
        Delete old_endnode from endnodes list
        Return list_of_diverged_nodes
 end

Function Diverge_Converge (old_endnode)
 begin
        number_divcon := Random_Integer (n)
        diverged_nodes := Diverge(old_endnode, number_divcon)
        numb_extend := Random_Integer (m)
        For each diverge_node in diverged_nodes do
          begin
                endpath_node := Extend(diverge_node numb_extend)
                add endpath_node to endpath_nodes_list
          end
        Converge(endpath_nodes_list)
 end

Function Random_Connection ()
 begin
  Try_again
        head := Pick_Randomly (all_nodes 1)
        tail := Pick_Randomly (all_nodes 1)
        if no loop created, then Connect_Nodes (head tail data_mean data_variance)
        else goto Try_again
 end

Function Pick_Randomly (list_of_nodes number_to_pick)
 begin
        if {number_to_pick = 1} then
                Pick and return a random node out of list_of_nodes
        else
                Pick and return a list of nodes out of list_of_nodes
 end
```

Function **Random_Integer** (*range*) picks and returns a random integer in [0,*range*]

Function **Connect_Nodes** (*tailnode headnode data_mean data_var*)
connects tailnode to headnode, assigning the arc a number of data
units chosen from a uniform distribution with given mean and variance.

Function **Create_Node** (*mean variance*) creates and returns a new node with execution time chosen from a uniform distribution with given mean and variance.