PARTITIONING AND ALLOCATION OF FUNCTIONAL PROGRAMS FOR DATA FLOW PROCESSORS

Thirumalai Muppur Ravi

April 1986
CSD-860063

UNIVERSITY OF CALIFORNIA

Los Angeles

Partitioning and Allocation of Functional Programs for Data Flow Processors

A thesis submitted in partial satisfaction of the requirement for the degree Master of Science in Computer Science

by

Thirumalai Muppur Ravi

1986

To my parents

TABLE OF CONTENTS

page

ACKNOWLEDGEMENTS

I am deeply grateful to my advisor Professor Milos Ercegovac for his unwavering patience, encouragement and support. His ideas on computer architecture have been very stimulating and farsighted, and have given me a solid foundation in this area.

Pak Chan who worked with me on several projects has helped me in numerous ways and I would like to thank him particularly for taking the time when I first joined the project to show me the ropes.

Doris Sublette helped me in tracking references and publications and has been a good friend throughout my stay here.

Partial support from NASA Lewis Research Center grant NAG 3-132 and Sandia National Laboratories Contract No. 25-3074 is gratefully acknowledged.

ABSTRACT OF THE THESIS

Partitioning and Allocation of Functional Programs
for Data Flow Processors

by

Thirumalai Muppur Ravi
Master of Science in Computer Science
University of California, Los Angeles, 1986
Professor Milos D. Ercegovac, Chair

This thesis seeks to demonstrate the viability of High Level Data Flow Architectures which utilize a number of high performance uniprocessors. Data flow techniques are applied at an optimal level of granularity instead of the fine grain granularity of conventional data flow machines.

Programs written in the Funtional Programming Language (FPL) are analyzed to extract a program graph with fine grain parallelism. By symbolic interpretation and resolution of functional forms we obtain a data flow graph. A criterion is established for the reduction of the fine grain graph to obtain a task graph which exploits the appropriate degree of parallelism. The reduced graphs are partitioned based on critical paths, which are the longest paths in the graph. Partitioning according to critical paths allows us to give priority to tasks on the critical path.

An algorithm for the allocation of partitions to the given number of processors has been developed. The allocation algorithm is a heuristic algorithm for static allocation. Finally we study the performance of the allocation algorithm and observe the speed-up obtained by increasing the number of processors. We also observed an improvement in performance when only appropriate degrees of parallelism are extracted instead of complete parallelism.

CHAPTER I

Introduction

1.1 Architectural Issues for High Speed Computation

The drive towards high speed computation has lead to considerable investigation into the exploitation of the von-Neumann model of computation for high performance. While the advances in technology lead to a significant reduction of cost and resulted in orders of magnitude of improvement in processor efficiency and memory access time, it became clear that VLSI technology was not being effectively utilized and the machine organization had become a bottleneck in achieving higher speeds. Early high performance processors use instruction pipelining [KOGG 81] where each instruction is subdivided into successive operations, and several instructions at different stages of operation are overlapped. A significant speed up is achieved by pipelined processors which closely adheres to the von-Neumann model which is simple and whose implementation is well understood. The limit on the number of operations the instruction can be divided into, besides the starvation of the pipe due to branching and dependencies restricts the concurrency that can be exploited.

Current vector processing architectures ([HWAN 81] & [ERCE 86]) exploit the presence of vector operations in programs and set up operations for the entire array just by a single instruction decode. Computers like Cray-1, based on this approach have demonstrated very high performances for vector operations. The efficiency of the vectorization by the compiler depends on the vector features present in the language and the extent of variable scoping and assignments in the program. The amount of vectorization possible will depend ultimately on the available concurrency in the program. To achieve speeds greater than those delivered by vector processors requires abandoning the single instruction stream and recognizing that control has to be distributed. The exploitation of parallelism present in a program which is not vectorizable or even parallel streams of vectorizable code, is necessary to further speed up program execution.

Multiprocessor systems ([ENSL 77] & [STON 80]) can exploit the process level parallelism that exists in programs. The principal problem to be solved in multiprocessors is the problem of programmability - mapping the program onto the processing elements. Partitioning, allocation and scheduling are the steps by which a computation can be divided, assigned to and executed in a certain order in processing resources. The size of partitions is determined by the tradeoff between the gain by exploiting parallelism and the overhead due to interprocessor communication. Extensions of the von-Neumann model have to deal with the problem of sharing common memory which requires a protocol for correct memory access. In multiprocessors, instruction level partitioning has proved to be unfeasible [GEHR 82] due to the large amount of overhead involved in the synchronization and message passing required for each instruction, and as a result only a very small degree of parallelism, which is at the procedure level is exploited.

The data flow approach [DENN 80], which is radically different from the von-Neumann approach, discards the program counter and distributes control of the program. A node in the program is enabled when all its operands are available. Nodes or program segments communicate by passing data values and do not have a concept of a globally shared updatable memory. The data flow concept in recent years has attracted much attention, and several research efforts are underway to develop high-speed data flow architectures. But, despite the attractiveness of the data flow concept, initial results of data flow designs have not been very encouraging. The exploitation of maximum parallelism, though conceptually simple, has several problems associated with it. Data flow architectures, like multiprocessors, execute a single instruction in a processor and rely on run-time methods for activating as many instructions in parallel as possible. Unfortunately, this simplistic view of exploiting maximum parallelism whenever possible suffers because no attention is paid to the critical paths in the program [GAJS 82]. Having no scheduling strategy is particularly detrimental when there are limited resources, with the degree of parallelism exceeding the number of resources, as computations are carried out much before they are actually needed and at the cost of critical paths in the program. While instruction level parallelism avoids the partitioning process altogether, saving compile time analysis of programs, the reliance on a greedy approach for scheduling can result in an overall nonoptimal utilization of resources due to the lack of global view of the computation.

Another serious problem is the large overhead incurred for the activation of each data flow node. Data flow computers tend to have long pipelines [GAJS 82], with waiting matching units for activation, instruction fetch units, structure management units, functional units and token labelling and routing units. By having a single instruction execute in the pipeline the fraction of time spent in actual arithmetic operations is only a fraction of the time spent in the entire pipeline. An additional overhead is the time spent in the communication network between the processors. The net effect of this is that the critical paths in the program are executed inefficiently and, whatever the parallelism present in the program, the response time will be hurt. In the data flow machines proposed, the critical path cannot be pipelined as each instruction on the critical path depends on the previous instruction. It is clear that the response times achievable from conventional data flow are not the best, and may not be even competitive with high performance vector machines when the parallelism is regular.

We intend to investigate the applicability of the data flow approach at different levels of granularities of subcomputations. In our model the program is partitioned at compile time into tasks which may contain several instructions. Each task after partitioning is considered to be a node in the data flow graph. At the task level the model is strictly data flow, tasks being activated when all the arguments of the task have arrived. The tasks communicate with each other by passing arguments between different tasks. The size of the task is based on a tradeoff between exploiting parallelism and the overhead due to communication and activation. Other work on variable resolution graphs include [GAUD 82], [GAUD 84], [BABB 84] & [RAVI

83].

Each task by itself is executed on a high performance von-Neumann processor, because of the high efficiency of the von-Neumann processor in executing sequential code. We exploit features like pipelining, fast accumulators and registers, and instruction caches for the execution of a task. The execution of the task is transparent to the top data flow level without any side effects. We thus seek to exploit an appropriate level of parallelism for which data flow gives the best performance and execute each of the concurrent processes efficiently by von-Neumann processors.

The difference between the high level data flow approach and the general multiprocessor approach is in the activation of tasks and the communication between processes. A data flow task is activated by the arrival of all its arguments and hence can adjust to runtime-dependent delays in the system. Data flow tasks moreover have predecessor-successor relationships with communication solely by arguments and results of tasks, thus avoiding the idle time and overhead due to synchronization primitives.

1.2 Language Issues

We now concentrate on the recognition and extraction of parallelism in programs written in high level languages. Imperative languages like Fortran, PL/1 etc. were designed for execution on a von-Neumann uniprocessor model and contain control dependencies for sequencing the program and explicitly use primary memory. But the execution of an algorithm is constrained only by its data dependencies and hence the removal of control dependencies would reveal the true parallelism (i.e., maximum parallelism) present and would enable us to obtain a fine grain data flow graph. The detection of statement level parallelism has been studied ([BERN 66], [KUCK 81] & [ALLAN 81]) and emphasize transformations to remove control dependencies.

Partitioning and vectorization compilers which transform programs statically have been reported ([KUCK 81] & [EL-DESS 81]). The analysis is very involved because of the complex scope of variables utilized by programmers who try to optimize memory and exploit the control sequencing for this purpose. Multiple nesting of control structures and the use of undisciplined control structures (ex. go-to) can make the analysis all the more difficult. Despite the difficulty in writing optimizing compilers, data flow graphs with some lingering influence of control structures can be obtained. Taking into account the existing software environment, compatibility and user familiarity with established programming languages, the additional time for compilers to obtain data flow graphs (which may not be very fine grain or flexible) may be justified. The effectiveness of the vectorization approach is discussed in [ARNO 82].

An alternative is to provide extensions to imperative languages which would reduce the complexity of analysis while making the programmer more responsible for expressing parallelism explicitly. Fork & join, test & set, semaphores, monitors etc. are explicit high level constructs for the synchronization of concurrent processes. Constructs like fork & join rely on the programmer for the detection of all parallelism which is burdensome and results in the exploitation of block or procedure level concurrency only.

Applicative languages (functional languages) ([BACKUS 78] & [ACKE 82]) are based on a programming style which is more naturally suited for concurrency exploitation. These languages enforce the single assignment rule where the variables can be assigned only once. The partitioning and vectorization compiler approach ([KUCK 81] & [EL-DESS 81]) achieves the same property by renaming. Freedom from side effects ensures that the data dependencies are the same as the sequencing constraints. Another feature associated with functional languages is the locality of effect and definite scope of variables. These properties, which are features of applicative languages, are the goal of the partitioning compilers for imperative languages. Partitioning compilers achieve the same functional semantics after complex and time consuming transformations.

Functional languages facilitate modular construction of programs allowing us to form tasks consisting of a function of the appropriate size. The partitions formed can be verified for correctness because of the mathematical properties applicable to functions. For our purpose we have chosen a functional language (FPL) [LAHTI 81] based on Backus FP [BACKUS 78], from which a direct translation to a directed dependence graph can be obtained.

The limitation in the use of functional languages is due to its being relatively new. Simple constructs in imperative languages are sometimes cumbersome to express in functional languages, and generally functional languages have been found to be verbose. In addition problems of structure representation and the representation of explicit parallelism is still open.

1.3 Thesis Overview

This work seeks to demonstrate that data flow principles can be applied at any level and that it is desirable to exploit an appropriate level of parallelism and not the lowest level of parallelism using data flow. An approach for the partitioning and allocation of data flow graphs to execute on a general data flow machine is proposed.

Chapter 2 introduces the programming environment, the model for partitioning and allocation and the proposed organization of a data flow computer based on the principles discussed above. The implementation of the system developed for automatic translation, partitioning and allocation is discussed. In Chapter 3 the translation of functional programs to data flow graphs is described. The criterion for reducing parallelism and exploiting the appropriate levels of parallelism is established. We then present a methodology for partitioning data flow graphs. Chapter 4 is concerned with the algorithm for allocating the partitions to processors along with a scheme to force an execution order by giving priority to critical paths in the program. Chapter 5 illustrates our scheme with an example along with the evaluation of its performance. Finally, Chapter 6 concludes with a summary of our work and proposes some problems to be investigated.

CHAPTER II

Introduction to the Model

2.1 Assumptions

Our objective in distributing a program over several resources, is to achieve better performance, i.e., a shorter response time for program execution. The response time is the time taken to complete execution of a program, and consists of the processing time, communication time and the task activation overhead. We assume a deterministic model, where the execution time of instructions, the time taken to communicate data between two processors and the overhead in the activation of a task, i.e., the time spent in the pipeline in noncomputational activities, is estimated apriori. Our approach is in contrast to attempts in reducing response time indirectly by identifying a dominating parameter influencing response time, such as interprocessor communication, data structure access or load balancing, and optimizing this parameter ([CHOU 82], [IRANI 82] & [HAES 80]).

The partitioning and allocation of tasks is done at compile time. Assumptions are made on the size of the input data structure, run-time dependent structure sizes and the number of iterations for run time dependent loops. It is our intention to demonstrate that static partitioning and allocation algorithms are feasible. A static analysis would also reveal information on data access patterns, which will facilitate storage of data structures and memory management. Compile time allocation would make it possible to predict which task is likely to be executed next on a processor, making code and instruction cache techniques very effective.

Initially a fine grain data flow graph is obtained by a direct translation of functional programs. We assume the computation graph to have multiple entry nodes and a single result node. Graphs with multiple result nodes can be converted to this form by adding a dummy node with arguments from each of the result nodes. The fine grain data flow graph is reduced by applying a set of rules, to obtain a task graph. A task consists of a single node which replaces a subgraph in the initial data flow graph. A task is a complete self contained portion of the computation, which is started only when all its inputs have arrived. The task is not made to wait for additional inputs once it has been initiated. Tasks are formed by combining sequential instructions and by reducing parallelism which cannot be efficiently exploited. The reduction of parallelism converts the fine grain data flow graph into a large grain data flow graph. Parallelism which exists within a task is not exploited. The task graph is a large grain data flow graph with each node representing task computation, and the arcs between nodes representing communication between tasks.

After obtaining a high level data flow graph, the next step is to map it onto the data flow computer. The partitioning process groups together tasks, decreasing the number of items to be allocated, thus reducing the complexity of the allocation. Partitioning generaly involves a logical division of the program depending on the structure of the computation. An important criterion for partitioning is the reduction of communication between partitions to avoid congestion and loading of the intercommunication network. Partitions are indivisible over processing elements and are the basic unit of allocation. Therefore tasks which communicate can be in the same partition, but tasks which can be executed concurrently should be in different partitions. Finally, we do not permit overlapping partitions, i.e., a task can belong to only one partition in order to avoid execution anomalies and code redundancy.

The allocation process involves the dedication of partitions to limited resources under constraints imposed by the architecture, in order to minimize the response time. We do not consider enumerative and optimal allocation algorithms due to the computational complexity in terms of the space and time as the program size and the number of processors increases [GAREY 79]. We have developed heuristic algorithms for obtaining minimum execution times for a given number of processors.

A task is not interruptable and once the processor starts executing it, it has to be completed; but intermediate results can be sent out as they are computed. A data driven mode of activation is assumed, i.e., a task is executed when all its operands have arrived. While the data flow mechanism is a control mechanism for sequencing, it by itself may not lead to optimal execution. It is not always a good policy to activate tasks as soon as possible, and it is sometimes better to keep a processor idle even when there are tasks ready [RAMA 72]. This is because a task on a critical path with much higher priority may become available and will not be able to execute if a long low priority task is occupying the processor. Further when several tasks are ready for execution on a processor; tasks on the critical path should be given priority in execution. Hence for minimum response time allocation, a mechanism for the control of task activation is introduced to favor critical paths in the program. Unlike the multiprocessor machines, no central control exists for scheduling, and therefore this control has to be incorporated within the data flow sequencing mechanism. A poor partitioning and allocation for data flow, unlike von-Neumann multiprocessors, does not compromise the correctness of the computation, but at most results in larger response times.

To summarize the assumptions :

1. We assume that the program is given as a set of tasks in a directed, acyclic precedence graph with no backward arcs. Task durations and communication times are known apriori and can be unequal.

2. A task can be activated only when all its inputs have arrived, but results can be sent out as they are computed. Tasks are not interruptible, and once the execution of a task begins, it proceeds till completion.

3. In a data flow model if more than one task has been activated in a processor then any one of the activated tasks can be executed next. There is no mechanism like the program counter which orders task execution.

4. The partitioning and allocation is done at compile time (static allocation) for a system with n identical processors. The partitioning step groups together tasks into partitions. Partitions are indivisible over processors and are the basic unit of allocation.

5. The performance criterion which is optimized is the minimization of response time (completion time) while keeping the number of processors required minimum.

2.2 Architecture Model

The targetted architecture consists of a number of identical processing elements, which communicate with every other processing element (PE) through an asynchronous communication network. Each PE in the system communicates with every other PE, and the time taken to communicate a token between one processing element and the input buffer of another processing element is the same. A token is a packet carrying data and the destination task address. Each processing element contains a high performance von-Neumann processor and local storage where tasks are preloaded at compile time. The von-Neumann processor in the system is capable of executing a task independently, and uses its accumulator, registers and local memory to store intermediate results. Our partitioning and allocation model utilizes timing parameters from the architecture, and is applicable irrespective of the organization of the processing elements. Figure 2.1 illustrates the data flow multiprocessor architecture to the detail necessary for our model.

We now describe an architecture of a dataflow multiprocessor. The partitioning and allocation algorithms are not limited to this architecture but can be applied to any architecture which has processors capable of executing tasks and which enforces the data flow activation mechanism.

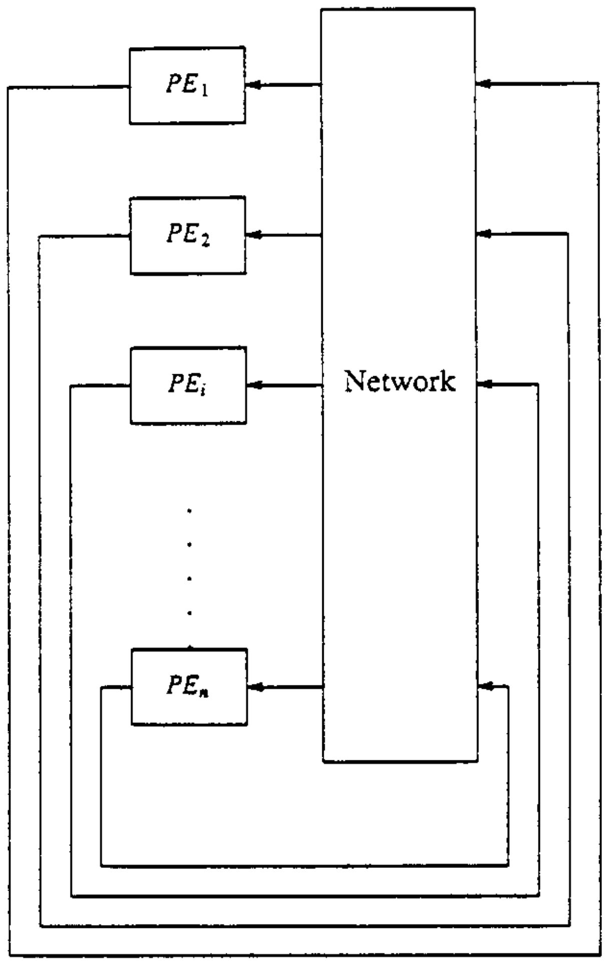


Figure 2.1 : General Data Flow Multiprocessor

The architecture consists of a number of high performance processing elements (PE's), each connected to the others by a broadcast bus. A token from a PE, which contains the result of the execution of a task is broadcast to all the other PE's (Figure 2.2) and selectively accepted by the destination PE. The communication mechanism between processing elements (PE's) is chosen so as to make the architecture easily expandable, i.e., the addition of a PE does not involve any change in the existing communication buses but just the addition of a new broadcast bus. Data between PE's is communicated through tokens. Each token additionally carries a unique identifier of the destination task. A task is enabled when all its input tokens have arrived. The task is executed and produces result tokens which are broadcast on the output bus and are utilized by the appropriate PE. The PE which houses the task specified by the destination task address in the token will pick up the token and process it.

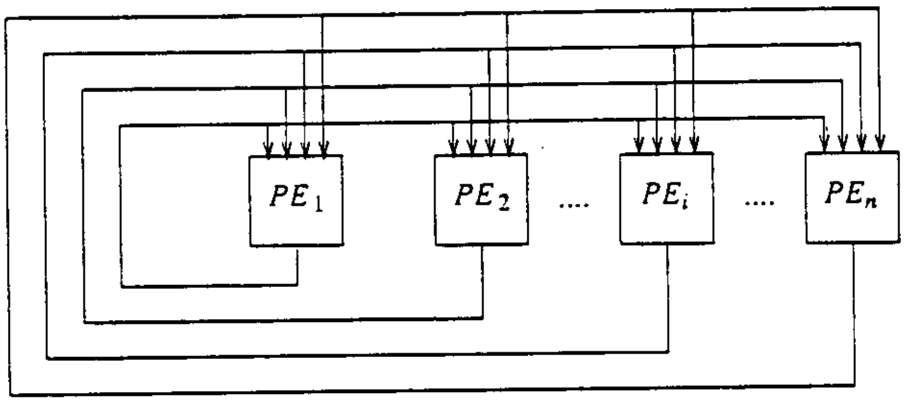


Figure 2.2 : Processing Elements Connected by Broadcast Buses

A functional description of a processing element consisting of a high performance von-Neumann processor is shown in Figure 2.3. The filter section services the input queues, and accepts or rejects a token by matching the task id. on the token and the tasks present in that PE. The data value of the accepted token is stored in a data memory. The data store is also responsible for accessing arrays and structures from a structure memory when the address is passed in the token. The task identifier from the accepted token also is sent to a waiting matching section which keeps track of the number of arguments arrived, and compares it with the number of arguments necessary for the activation of that particular task. If a task is ready for activation then the task id is passed to the task fetch unit and the execution unit. The execution unit fetches the data into local registers, and executes the task code using local registers and memory. The output section tags each of the results with the appropriate result task identifier number, and broadcasts them on the bus. Overlap between the stages of the pipeline, and the use of multiple functional units for each stage will speed up the PE.

A detailed description of a multiprocessor data flow architecture discussed above is given in ([ERCE 84] & [CHAN 84]).

2.3 Programming Environment

Programs are written in Functional Programming Language (FPL), which is suitable for the exploitation of low level concurrency. FPL is based on the functional language (FP) proposed by Backus [BACKUS 78]. The syntax used is slightly different, and some commonly used functions have been implemented as primitive functions. More details on the environment can be found in [BADE 83] and [LAHTI 81].

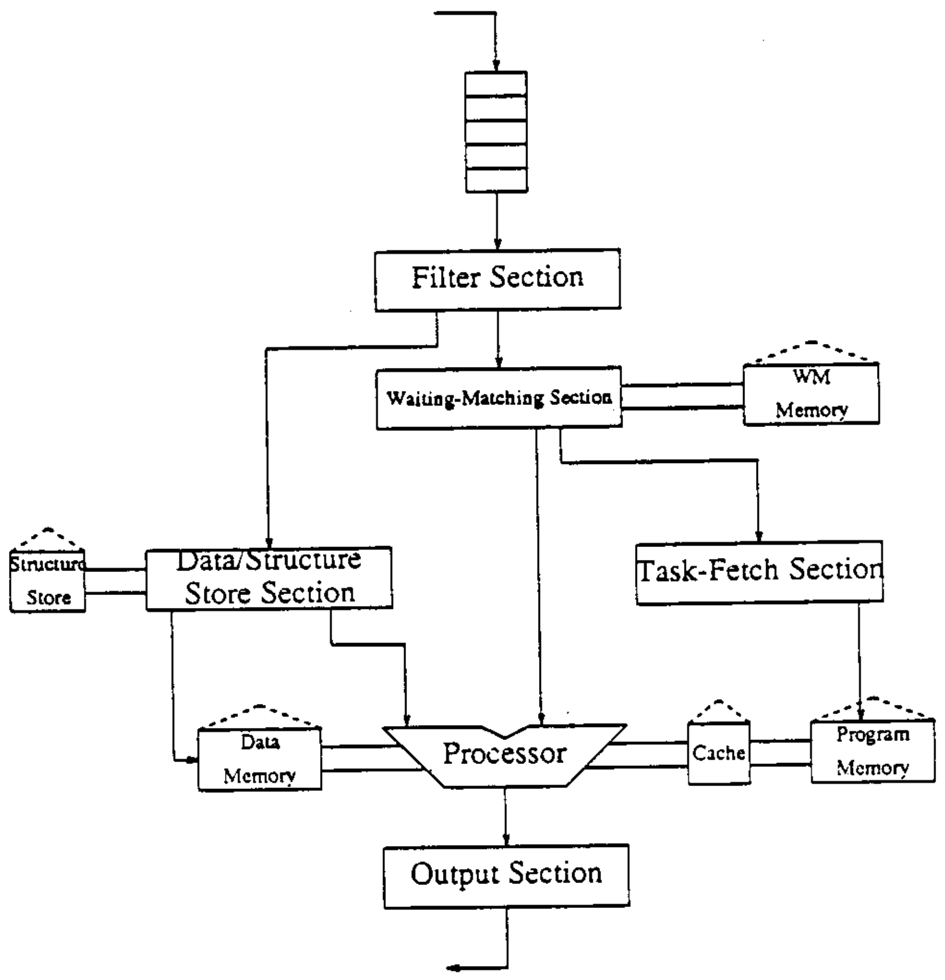


Figure 2.3 : Individual Processing Element

The Functional Programming Language (FPL) consists of objects, primitive functions, functional forms, user defined functions and an operator which is *application (:)*. Objects can be a single atom such as a character, integer or null object; a list of objects or an error symbol - *bottom (?)*. Functions are applied to objects to produce result objects. A simple description of primitive functions is given in Table 2.1. A functional form is an expression with other functions and objects as parameters. Functional forms combine functions to form new functions. The following are the functional forms in FPL.

1. Composition

   CM(f,g): x ≡ f: <g: x>

2. Construction

   [f,g,h]: x ≡ <f: x, g: x, h: x>

3. Condition

   IF (f,g,h):x ≡

   g: x (if f: x is True)

   h: x (if f: x is False)

4. Constant

   Ky: x ≡ y

5. Insert

   IN f: $<x_1, x_2, x_3, \ldots x_n>$ ≡ f: $<x_1$, IN f: $<x_2, x_3 \cdots x_n>$

6. Associative Insert

   AI f: x ≡

   $$f: <AI\ f: <x_1, x_2 \cdots x_{\lceil \frac{n}{2} \rceil}>, AI\ f: <x_{\lceil \frac{n}{2} \rceil + 1}, \cdots x_n>>$$

7. Apply-to-all

$\text{AP } f: x \equiv \langle f{:}x_1, f{:}x_2, \ldots\ldots f{:}x_n \rangle$

A definition is used to name large functions formed by combining smaller functions. Thus an entire program in FPL is just a single function built from primitive, user defined functions and functional forms.

Table 2.1: Primitives Functions of FPL

Function	Description
SLk	Selects the k'th element of a vector
PK	Picks the k'th element of a vector
ID	Identity
TL	Tail of a vector
FR	Front of a vector
LA	Last object of a vector
IX	Integers from 1 to n
AT	True if object is an atom
NL	True if null sequence
LN	Length of vector
+	Integer add
-	Integer subtract
*	Integer multiply
/	Integer divide
#+	Floating Point add
#-	Floating Point subtract
#*	Floating Point multiply
#/	Floating Point divide
#SIN	Sin(y)
#COS	Cos(y)
#TAN	Tan(y)
#EXP	Exponential
#POW	Power
#LOG	Logarithm
#SQRT	Square root
&,|,!	Logical And, Or and Not
TR	Transpose
EQ	True if equal
GT	True if x > y
LT	True if x < y
DL	Distribute left
DR	Distribute right
AL	Append left
AR	Append right
CT	Concatenate vectors to form one vector
SP	Splits vectors into halves
PR	Forms vector into vector of pairs
CL,CR	Circular rotation to left and right
RV	Reverses order of elements in a vector

CHAPTER III

Program Decomposition and Partitioning

3.1 Overview

In this chapter we discuss how data flow graphs are obtained from high level language programs and then partitioned. Functional programs are initially decomposed into a structured program graph containing primitives, user defined functions and functional forms. The functional forms are resolved by replacing them by an appropriate execution model. By symbolic interpretation the structured program graph is transformed to a fine grain data flow graph, with user defined functions and functional forms replaced by primitives. The fine grain data flow graph is reduced to a task level data flow graph based on processing time and communication time criterion for achieving better response times, assuming we have a system with limited resources. The data flow graph is grouped together into partitions based on the critical paths in the program. The timing and associated parameters of the partitions capture the necessary information for a combinatorial solution of the allocation problem.

3.2 Graph Generation

The FPL program is a function formed by combining smaller functions using functional forms. It is initially parsed into an intermediate level representation [FELL 81] - Complete Decomposition Form (CDF) [ERCE 83, LU 84]. This is done by scanning the program for the *Compose (CM)* functional form which reflects the sequential dependencies between functions. This information is used to decompose the program into an instruction (FPL primitive, functional form or user defined function) level representation with dependency information. Some preliminary transformations are performed on the CDF representation to make program execution more efficient [LU 84].

The data driven CDF code is converted to a directed program graph where the nodes represent high level (FPL) program constructs and the arcs represent the dependencies. The target machine language is an instruction set consisting of primitive functions of FPL along with some additional instructions necessary to support the execution. Additional instructions like *list* and *unlist* to be discussed in Sec. 3.6 are provided for structure handling. The high level operations are directly supported by the architecture and are implemented as instructions or subroutine calls. The high level constructs of the machine language differ from the FPL constructs in that *application (:)* is not strict. A function does not have to be applied to a single object as in FPL but can have more than one input. Non-arithmetic operations like *construct (CN), select (SL), pick (PK), tail (TL)* etc. which manipulate data structures are retained during the partitioning stage and can be resolved if necessary after the partitions have been formed. Structure operations are necessary if the vector nature of functional forms like *apply-to-all* is to be exploited by a pipelined processor, or if we seek to unfold the functional forms to a limited extent for execution efficiency.

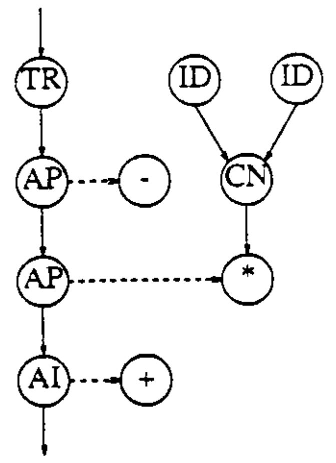
An alternate approach [CHAN 84, ERCE 84a] is to resolve all (except *pick*, and conditionals depending on the value of an object) the routing and control constructs by symbolic interpretation. The effect of resolving all the non-arithmetic functions like *select, construct* etc. is to decompose the vectors and treat each element of the vector as a scalar. Resolving the functional forms causes a large amount of replication of code, because the number of copies of code will be equal to the number of elements in the object to which the functional form is applied. By resolving all the functional forms and structure manipulation functions, much of the information required to systematically group together code (when resources are restricted and maximum parallelism cannot be exploited) is lost.

We choose not to unravel functional forms initially to prevent code replication, facilitate the formation of partitions, and for data locality. The structured program graph obtained is hierarchical and consists of primitive functions, functional forms and user defined functions. Functional forms like *associative insert (AI), insert (IN) and apply-to-all (AP)* are represented as macronodes with a pointer to the graph of the function being applied or inserted. As functional forms can be used for combining other functional forms, the entire program graph can contain several nested subgraphs which can be accessed by traversing the pointers. Figure 3.1 illustrates the graph of the function to calculate the sum of the square of the deviances.

$SumDevSq = \sum (y_i - y_j)^2$

The FPL function for the graph is:

CM(AI+, APCM(, [ID, ID]), AP-, TR)*



Main Graph Auxiliary Graphs

Figure 3.1 : Hierarchical Program Graph

Example

The translation of a functional program into a program graph is illustrated by an example. The function $f(x) = x^3 + 4x^2 + x + 2$ can be written in FPL as:

```
/*CONSTANT*/
B = K4.0;
D = K2.0;
```

```
/*FUNCTION*/
CubX = CM(*, [SqX, ID]);
SqX = CM(#POW, [ID, K2]);

/*MAIN*/
main = CM(#+, [CM(#+, [CubX, CM(#*, [B, SqX])]), CM(#+, [ID, D])]);
;
```

The FPL program is parsed into the corresponding CDF form. The order of the functions in the CDF code is strictly according to precedence, i.e., the inputs to a function at any line are evaluated at previous lines. The corresponding CDF code in data driven form is given below. The primitives are explained in Table 2.1.

```
f11 : ID .
f12 : K2
f10 : CN f11 f12
f8 : #POW f10
f9 : ID .
f7 : CN f8 f9
f5 : * f7
f14 : K4.0
f13 : CN f14 f8
f6 : #* f13
f4 : CN f5 f6
f2 : #+ f4
f17 : ID .
f18 : K2.0
f16 : CN f17 f18
f3 : #+ f16
f1 : CN f2 f3
f0 : #+ f1
```

The graph corresponding to this program is shown in Figure 3.2. The primitives used in the nodes of the graph correspond to the FPL functions in the program.

As functional forms manipulate functions rather than values, an execution model for each functional form has to be specified. In Section 3.6 the resolution of

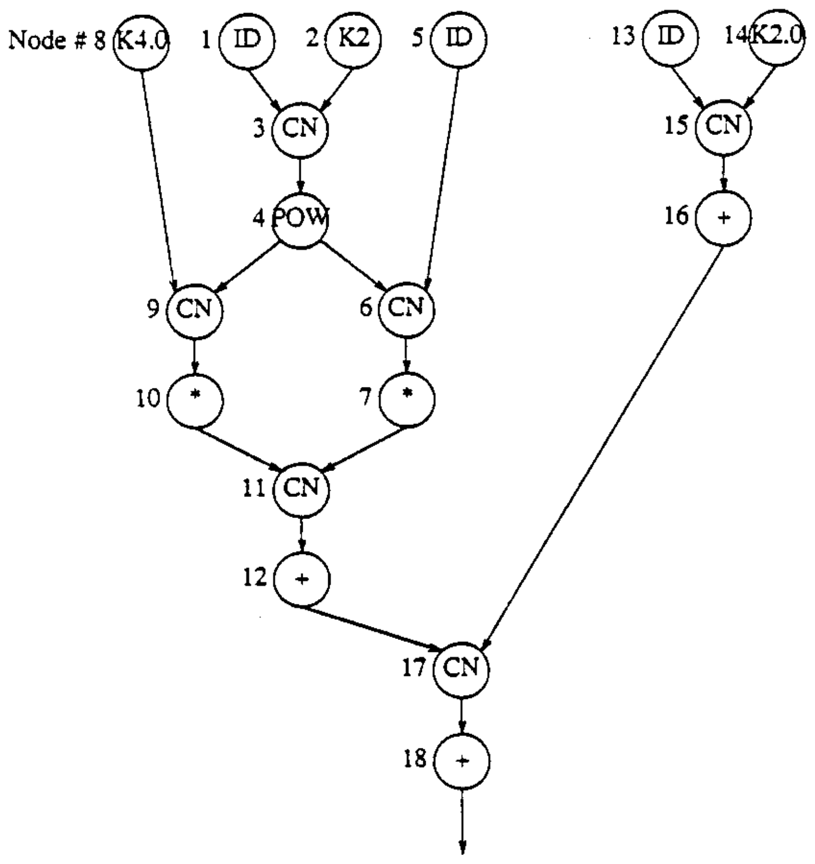


Figure 3.2 : Graph for Polynomial Function

functional forms *IF, AI, IN, AP* are discussed. In this study we assume that the data flow graphs have no loops or branches.

After obtaining an intermediate level code (CDF) from the FPL program it is translated into a graph with multiple levels of nesting corresponding to functional forms. Symbolic interpretation (Section 3.4) determines the size of data paths in the

```
begin
   1. revcdf {obtain data driven cdf code}
   2. hierarchical-graph-generation(graph,nesting)
   3. symbolic-interpretation
   4. for i:=1 to nesting do assigntime(graph)
   5. for i:=1 to nesting do graph-reduction
   6. Resolve-IN
   7. Resolve-AI
   8. Resolve-AP
   9. graph-reduction
end.
```

Figure 3.3 : Graph Generation

program. It also provides information for the resolution of functional forms. Now we are ready to assign processing times and communication times to each of the nodes in the main graph and the nested graphs. Before the resolution of functional forms, we reduce each of the nested graphs. Graph reduction is discussed in Section 3.5. The nested graphs are reduced in order to get an accurate estimate of the critical paths of the nested functions, which will determine what degree of parallelism can be extracted efficiently. The functional forms are now resolved (Section 3.6) and a single directed graph is obtained. This graph is reduced to obtain the final data flow graph.

The steps involved in the generation of a large grain data flow graph are shown in Figure 3.3.

3.3 Assignment of Time & Longest Path Calculation

Each node in the program graph contains information on the processing time (t_p) of the node and the communication time (t_c) of its results. t_p is the time to to execute an instruction and is expressed in number of cycles. The instructions are assigned different processing times based on an estimate of their execution times. Instructions like *front (FR), last (LA), pick (PA), construct (CN), select (SL), identity (ID) and tail (TL)* are estimated to take only a small fraction of the time taken by other instructions.

The communication time (t_c) is the sum of the time taken to communicate a token between two processing elements and the activation overhead of the processing element. We assume that a token can be communicated to another PE in a fixed number of cycles, and is independent of the position or distance from other processors. The activation overhead is the time taken by the non-computational stages of the pipeline, like the filter section, waiting-matching section, task fetch section and the output section. Tokens may carry a data value or a structure address. The communication time depends on the the number of data values being sent. We model the communication time as

$t_c = \alpha n + \beta$

where

α represents the overhead factor due to the size of the data structure

n is an indication of the no. of elements in the object

β is a constant factor depending on the architecture.

Associated with each node (i) is the longest path (lp_i), which is the total time of the longest path to the top of the graph. The longest path of a node is the minimum possible time at which the results from that node will be available. When a node has several results then we associate a single communication time t_c with it.

The algorithm to calculate the longest path of a node is:

1. *If the node (i) is on the top of the graph, i.e., it has no arguments then* $lp_i = t_{p_i} + t_{c_i}$

2. *Else* lp_i *= Max (longest path of argument nodes)* $+t_{p_i}+t_{c_i}$.

3.4 Symbolic Interpretation

The number of elements of data sent from a node, is a factor in the calculation of the communication time (t_c). Symbolic interpretation of the input data structure is used to obtain the structure and number of elements in the data object corresponding to each arc in the graph. The number of elements in the data object may be run-time dependent where worst case estimates of the size are made.

The input data structure is assumed to be symbolic, where the structure of the input data is accurately captured, but the actual values are represented by data. Corresponding to each FPL primitive, a definition relates the structure of the input to the output structure. The graph is traversed node by node, with the nodes having no arguments taking the program's symbolic input as their input structure. (The *Constant (K)* is an exception.) From the symbolic function definition of each node the output structure and hence the number of elements in an object data (n) is obtained.

The symbolic interpretation of the program to evaluate the size of the structures makes a conservative estimate when run time dependencies determine structure sizes. Worst case estimates are made for the *conditional (IF)* forms. When functional forms like *insert (IN) and apply-to-all (AP)* are encountered then each of the elements of the input structure is examined. If they are uniform, i.e., each element has the same structure, then the nested graph (the function being applied or inserted) has to be applied to only one of the elements of the input to the functional form. The output structure of the functional form can be constructed by replicating the output from a single element of the input. If the input structure to the functional form is not uniform i.e. the elements of the input do not have a similar structure, then the nested graph has to be repeatedly applied to each element of the input structure.

Example

Consider the function: *SumDevSq* $=\sum(y_i - y_j)^2$ whose data flow graph is illustrated in Figure 3.1.

The FPL function for the graph is:

CM(AI+, APCM(, [ID, ID]), AP-, TR)*

The symbolic input to the program is assumed to be:

((x, x, x, x), (x, x, x, x))

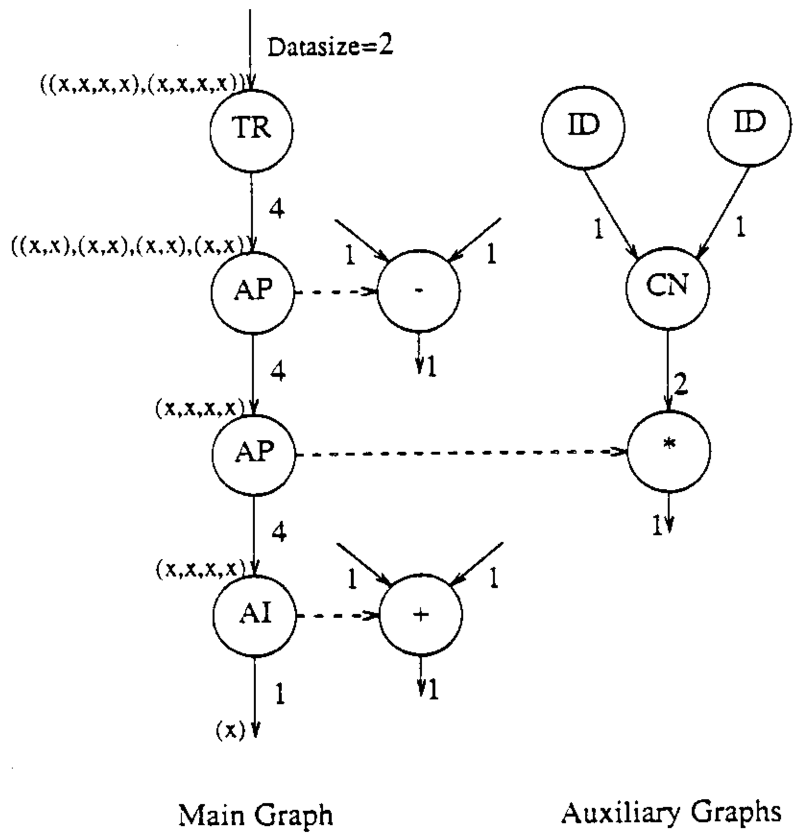


Figure 3.4 : Symbolic Interpretation of Hierarchical Graph

1. The application of *Transpose (TR)* to the input gives:

   ((x, x), (x, x), (x, x), (x, x))

2. *Apply-to-all - (AP-)* gives: (x, x, x, x)

   This is constructed by obtaining x as the output to (x, x) and replicating it four times.

3. *Apply-to-all CM(*, [ID, ID])* gives:

   (x, x, x, x)

   This is obtained by taking a single element x from the input to this function and applying the nested graph to this input.

   CM(*, [ID, ID]):x

   = * : [x, x]

   = * : (x, x)

   = x

4. Finally *AI+* of the input (x, x, x, x) gives:

   x as the result.

Figure 3.4 shows the graph of Figure 3.1 with the arcs marked by the no. of elements in the object being communicated. The no. of elements in the object is considered to be the number of elements in the first level of the structure.

The graph of Figure 3.2 is shown in Figure 3.5 with processing and communication times of each node listed in the node and on the arcs respectively. For the calculating the communication time, we take $\alpha = 0.2$ and $\beta = 1$. The no. of elements in the data object (n) for each arc is determined from the symbolic interpretation of the input (in this case (x)). Each node is placed on a level, with the nodes without arguments on level 0, and the root node on the highest level. The level of a node is one more than the maximum level of its arguments.

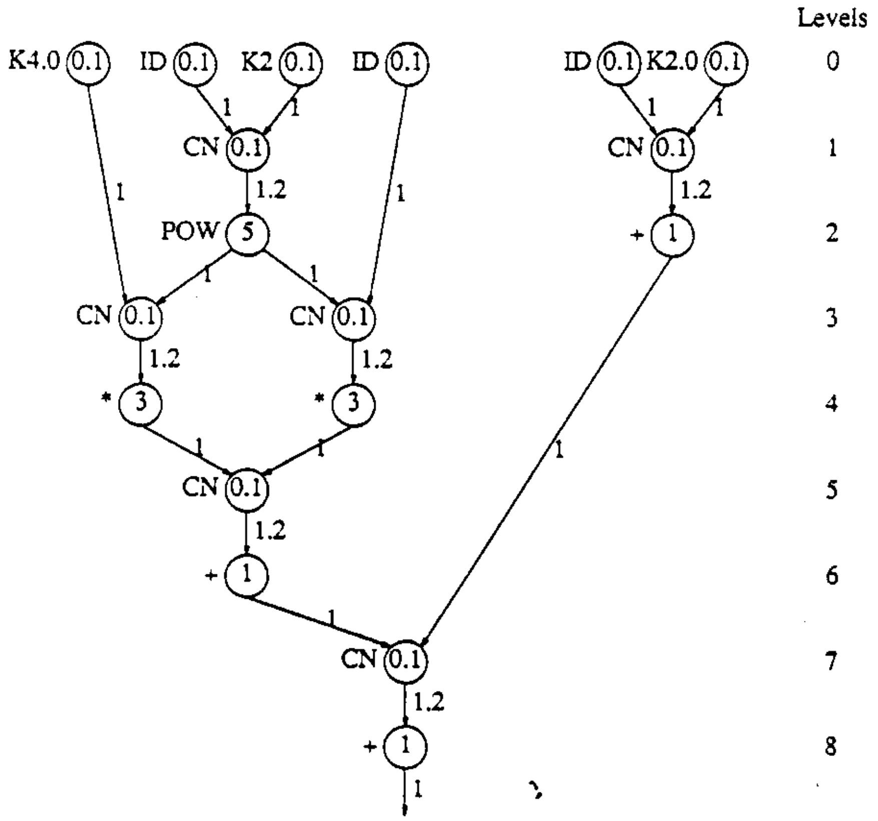


Figure 3.5 : Polynomial Function Graph with Timing Parameters

3.5 Graph Reduction

In this section the algorithm for graph reduction is presented, and its application to the program graph is described in the next section.

The fine grain data flow graph is reduced to a task graph, in order to efficiently utilize the parallelism present. By applying a set of rules, subgraphs in the data flow graph are replaced by a single node. The principal criterion for lumping together instructions into a single task is the minimization of the response time for the corresponding subgraph.

The graph reduction algorithm consists of two steps - in the first step sequential nodes are combined together, and in the second step parallelism which cannot be efficiently exploited is reduced.

3.5.1 Combination of Sequential Nodes

When each sequential instruction is activated and executed separately in a processor, each of them will have to bear the activation overhead of going through the data flow mechanisms like filter section, waiting-matching section, task (instruction) fetch section etc. which account for a large fraction of the communication time. This will be the case even if the instructions are allocated to the same processor. This overhead present in fine grain data flow execution is unnecessary, as no time can be gained by the activation and execution of these sequential instructions one at a time.

Sequentially ordered nodes which have a single argument arc (which may be a single value or a structure), and a single result arc, can be combined together into a single node, saving the communication time between them.

Figure 3.6a shows a subgraph which can be reduced to the graph of Figure 3.6b by the combination of sequential nodes. In Figure 3.7 when nodes 1 and 2 are

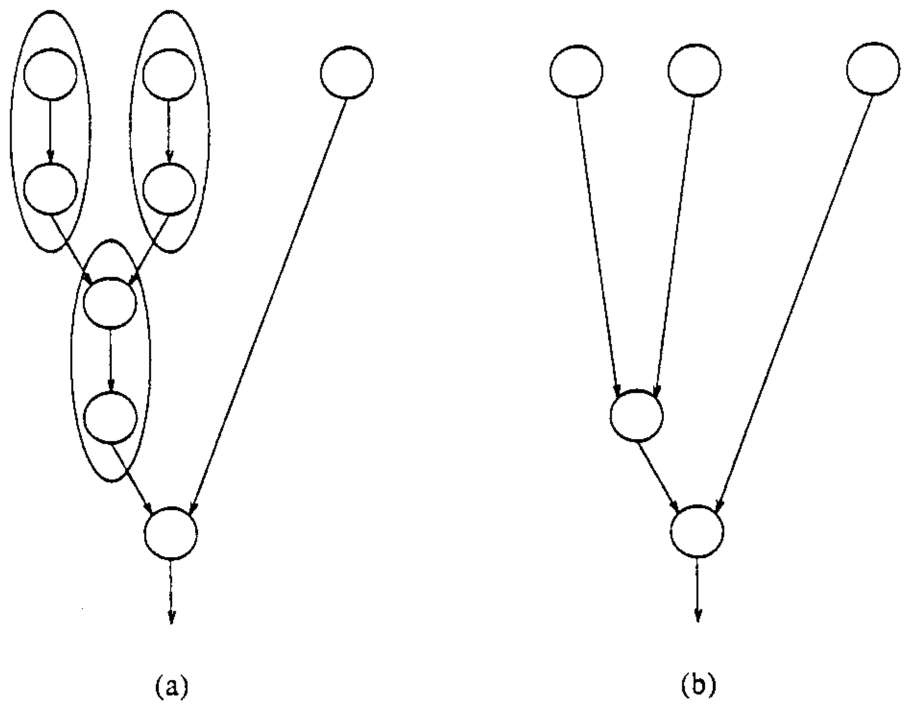


Figure 3.6: Sequential Reduction

combined to 1', the resulting processing time of 1' is the sum of the processing times of nodes 1 and 2, i.e.,

$$t_{p_1'}=t_{p_1}+t_{p_2}$$

The communication time of node 1' ($t_{c_1'}$) is the same as the communication time of node 2 (t_{c_2}). The result of instruction 1 is used immediately as the input for instruction 2 in node 1', saving the communication time between 1 and 2. By reducing the sequential nodes in the graph (Figure 3.7) the response time reduces from 12 to 10 cycles.

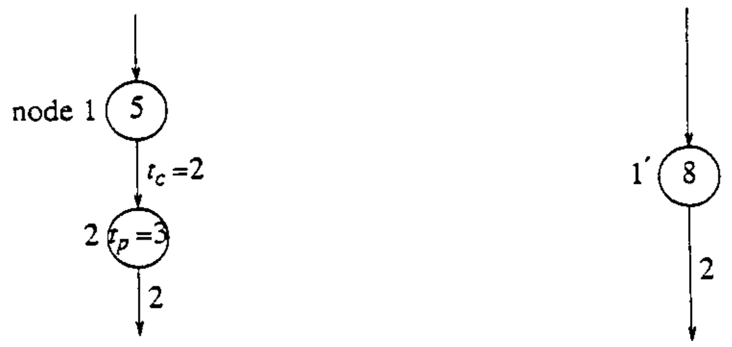


Figure 3.7 : Combination of Nodes 1 & 2

When sequential nodes are lumped together, a single activation results in the execution of all the nodes contained in the new node, avoiding the overhead of individual activation for each of the nodes.

3.5.2 Reduction of Parallelism

When the delay incurred due to interprocessor communication and activation overhead exceeds the gain in time due to concurrent execution, it is no longer justifiable to distribute the nodes over several processors. That is, when the response time of a subgraph executed sequentially in a single processor is less than or equal to the response time when executed concurrently, then the nodes are reduced and executed sequentially.

The condition for combining a node with its predecessor nodes is:

$$\sum_i t_{parg_i} \leq \max_i (t_{parg_i} + t_{carg_i})$$

where t_{parg} is the processing time of a predecessor node

t_{carg} is the communication time of a predecessor node and

i ranges from 1 to the number of predecessors (narg) of the node under consideration.

When this condition is satisfied then the node and its predecessor nodes are lumped together into a single node.

This step is illustrated in Figure 3.8. Figure 3.8a is a subgraph where the nodes are separated in order to take advantage of the parallelism, while in Figure 3.8b the nodes 1, 2 and 3 have been lumped together into $1'$. The processing time of the node $1'$ is equal to the sum of the processing time of the node under consideration and its predecessor nodes. In Figure 3.8b program code of nodes 1, 2 and 3 are executed sequentially. Therefore,

$$t_{p_1'} = t_{p_1} + t_{p_2} + t_{p_3}$$

The predecessors of the new node consists of all the predecessors of the predecessor nodes of the node being considered.

In the subgraph of Figure 3.8a, node 1 can execute only after the results from node 2 and 3 have arrived. If nodes 2 and 3 are activated at the same time then the result from node 2 will arrive after 12 cycles and the result from node 3 will arrive after 14 cycles. Node 1 fires at 14 units of time and the result from node 1 is ready for transmission after 24 cycles. In the sequential case the result is available after 21 cycles although the parallelism of the graph is reduced.

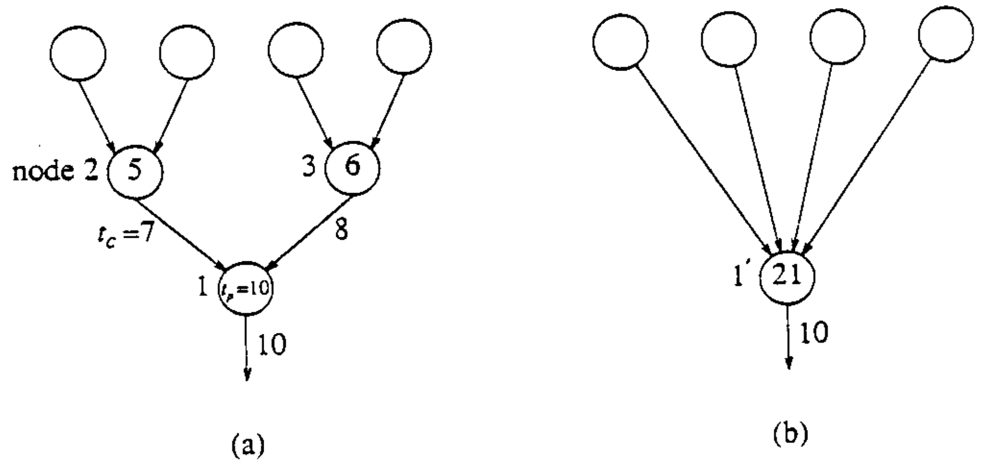


Figure 3.8 : Reduction of Parallelism

An algorithm to obtain variable resolution data flow graphs depending on the processing time and communication times of nodes is shown in Figure 3.9.

The systematic reduction of parallelism in order to efficiently utilize the parallelism, is based on local decisions at each node. The parallelism reduction criterion lumps together a node and its arguments without consideration of actually when the arguments are activated. By doing so the critical path of the program may after graph reduction contain some lumped nodes thus extending the longest path. Hence the parallelism reduction criterion is only a heuristic.

Under limited resource constraints, much of the overhead caused in managing an unexploitable degree of fine grain parallelism, is avoided by graph reduction. Further, allocation which is a polynomial time process takes far less time as smaller

```
procedure compress (i:typenode);

{This recursive procedure lumps a node and its argument node
 together, depending on the processing time and communication
 time. Each node has the fields; argument (arg), no. of
 arguments (narg), funct (function code, processing time (proctime)
 and communication time (commtime).}

begin
  with node[i] do
    if narg > 0 then
    begin
      {test condition}
      seqtime:=0; partime:=0;
      for k:=1 to narg
        begin
          seqtime:=seqtime + node[arg[i]].proctime;
          if (node[arg[k]].proctime + node[arg[k]].commtime)
                > partime then
            partime := node[arg[k]].proctime + node[arg[i]].commtime;
        end;
      if partime -seqtime ≤ 0 then
        {condition for reduction of parallelism is not true}
        for k:=1 to narg do compress(k);
      else begin
        {condition is true}
        copy the code in each of the arguments to node[i].funct
        node[i].proctime := seqtime;
        node[i].narg := sum of the narg of each of the arguments
                  of node[i]
        arguments of new node := arguments of all nodes combined
                with node[i]
        remove the old argument nodes from graph
        compress(i);
      end;
  end; {compress}

  procedure reduction (G:typegraph);
  {This procedure increases the grain size of the data flow graph (G) so
   that it can be efficiently executed. Starting at the root node, nodes
   are combined with its arguments.}

  begin
    compress(Root(G));
  end; {reduction}
```

Figure 3.9 : Variable Resolution Graph Reduction

number of nodes need to be allocated.

3.6 Resolution of Functional Forms

We now explore the application of the algorithm discussed in the previous section to the structured program graph obtained in Section 3.2. The nested program graph obtained from translating the FPL program consists of a main graph and auxiliary graphs representing functions combined by functional forms. The nodes representing forms like *AI, IN and AP* are similar to subroutines in imperative languages. Initially the graph reduction algorithm is applied to the main graph and each of the auxiliary graphs linked to it. The nodes representing functional forms are prevented from being lumped with other nodes because we intend to substitute graphs for the nodes at a later stage. The auxiliary graphs which are nested in the main graph are also reduced before substitution in order to get actual estimates of the time taken by the nested subgraph.

We unfold the functional form and specify a model for the efficient execution of the functional forms. A vector object consists of several elements to which the same function is applied. Our analysis assumes that scalar processors are used in the architecture, and hence elements of the vector have to be separated into individual scalar objects or else the vector has to be executed sequentially element by element.

The *conditional (IF)* functional form in FPL consists of the application of one of two functions on the data object depending on the boolean result of the condition tested. That is, *IF (p,f,g) : x*

returns f:x if p:x is T;

returns g:x if p:x is F;

else returns ?

We can implement this functional form with a data flow graph in two possible ways. In the first model (Figure 3.10a) the *IF* is merely a selector. Both branches of the conditional are evaluated as soon as possible and the *IF* acts like the merge actor. In this model the condition function, as well as the true and false branch functions are evaluated concurrently.

The model performs poorly when the true and false paths differ very much in length, and the shorter path is selected by the condition function. The *IF* node will have to wait, till the longer path is evaluated. A solution is to alter the firing rule for the *IF* node. The *IF* node can be made to fire if the boolean token and the result token from the branch corresponding to the condition have arrived; without having to wait for the token which is not going to be selected. The other disadvantage of this model is that a large amount of unnecessary computation will be performed, and may also result in the propagation of "*bottoms*" (undefined tokens). Propagation of "*bottoms*" occurs when one of the functions (f or g) when applied to the data object, causes a bottom to be generated because it is not the function selected. An important difference in the semantics of the high level machine instructions and the the FPL functions is that the high level constructs are not bottom preserving [BACKUS 78], i.e., sequences are permitted to have *bottoms (?)* as elements. Hence the semantics of our model permits *bottoms* as elements in order to permit the two branches as well as

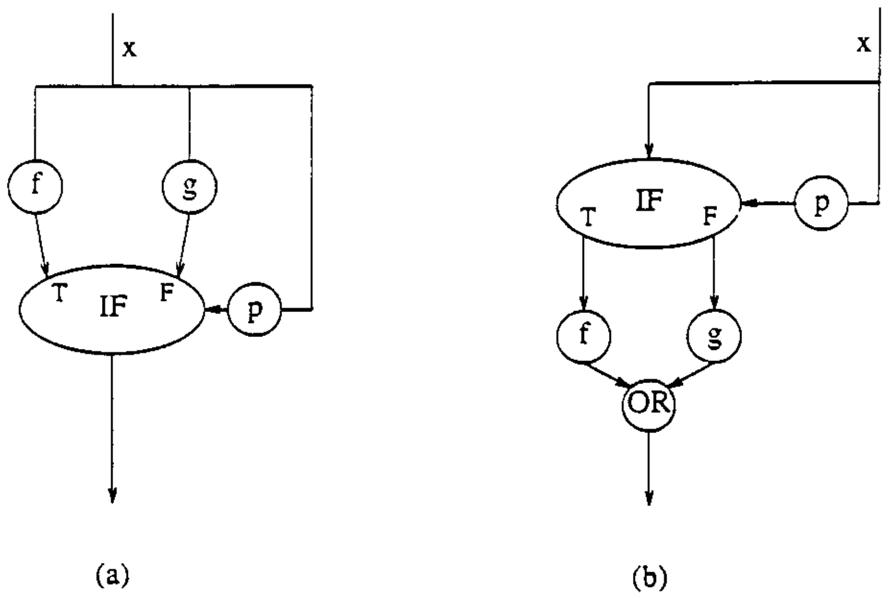


Figure 3.10 : Models for Conditionals (IF)

the boolean condition to be evaluated concurrently.

The alternate choice for implementing the conditional functional form is a demand driven model (Figure 3.10b) which executes the chosen branch after the condition testing function has been evaluated. While restricting the parallelism that can be exploited this model avoids unnecessary computation and the propagation of "*bottoms*" (undefined tokens). In our implementation we chose the first approach (Figure 3.10a) because the data driven model is consistent with our overall approach.

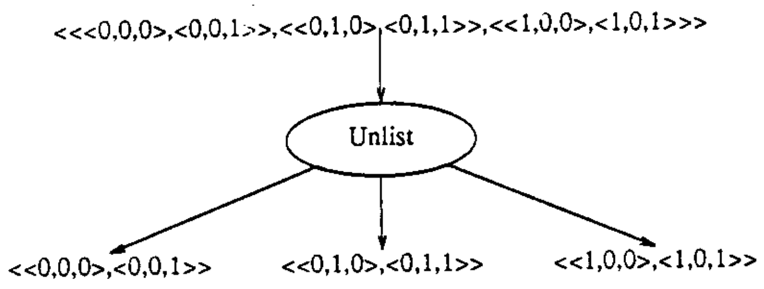


Figure 3.11 : The Unlist Function

Two new instructions are introduced for building and decomposing structures. The *unlist* instruction (Figure 3.11) separates the elements of a vector. The *list* function which is similar to the implementation of *concatenate*, builds a single vector from the argument elements (Figure 3.12).

Initially the *insert (IN)* and *associative insert (AI)* nodes are replaced by the corresponding subgraphs in the main graph and the auxiliary graphs. The *associative insert (AI)* is implemented as a binary tree (Figure 3.13). The number of data elements *(n)* in the structure to which the *AI* is applied to, is known beforehand from the analysis of the data sizes. The graph pointed to by the node *AI* is replicated n-1 times, assuming two input nodes, and is then connected to a binary structure. The inputs to the n-1 copies come from the *unlist* function, which separates the structures so that they can be concurrently operated upon. The macronode *AI* is substituted by the subgraph formed from the *unlist* function and the (n-1) copies of the graph pointed to by the *AI* node.

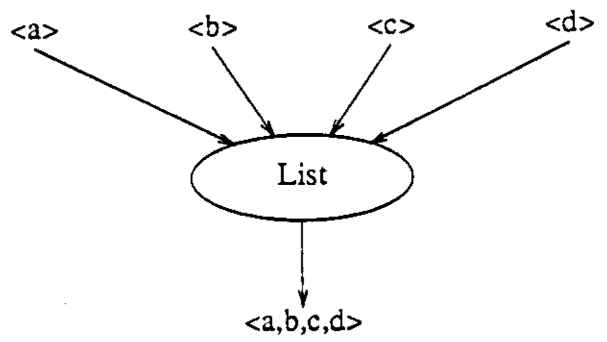


Figure 3.12 : The List Function

The *insert (IN)* functional form recursively applies the function to the input sequence. Figure 3.14 shows how the functional form is executed. The node *IN* in the graph is replaced by a function *IN+* which applies + to the input structure sequentially. The processing time of the new node is (n-1) times the processing time of the function applied, where n is the number of elements in the input structure.

Finally we resolve the *apply-to-all* form which applies the function to every element of the input structure. This functional form is frequently used to express parallelism in functional programs and similar to the unfoldable loops of imperative programs. Our first step is to apply high level transformations on the program graph. Optimization based on the algebra of programs [ISLAM 81, WADL 81, KIEB 81] can provide gains in execution time at the cost of the clarity of the program. We apply the transformation [LU 84]:

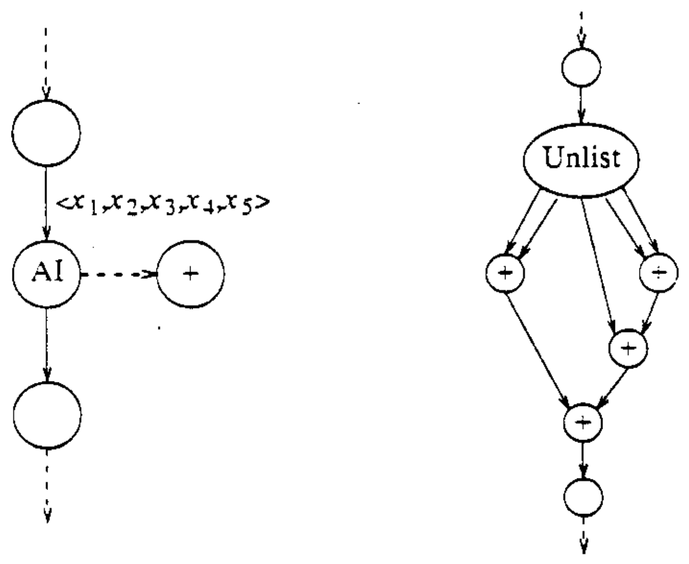


Figure 3.13 : Resolution of Associative Insert (AI)

AP $[f_1, f_2, \ldots f_n]$ = CM(TR,[APf_1,APf_2, ... APf_n])

This has the effect of replacing a single graph for the function being applied with several simpler graphs (Figure 3.15).

The resolution of the *apply-to-all* involves a tradeoff between data locality and concurrent execution. Figure 3.16 presents two models for the execution of the *apply-to-all*. In Figure 3.16a the function f_1 is applied to the structure $<x_1, x_2, \ldots x_n>$ sequentially. If there are n elements in the structure, and each function evaluation takes a single cycle then the response time will be n cycles. There will be no replication of code in this case and the structure does not have to be decomposed.

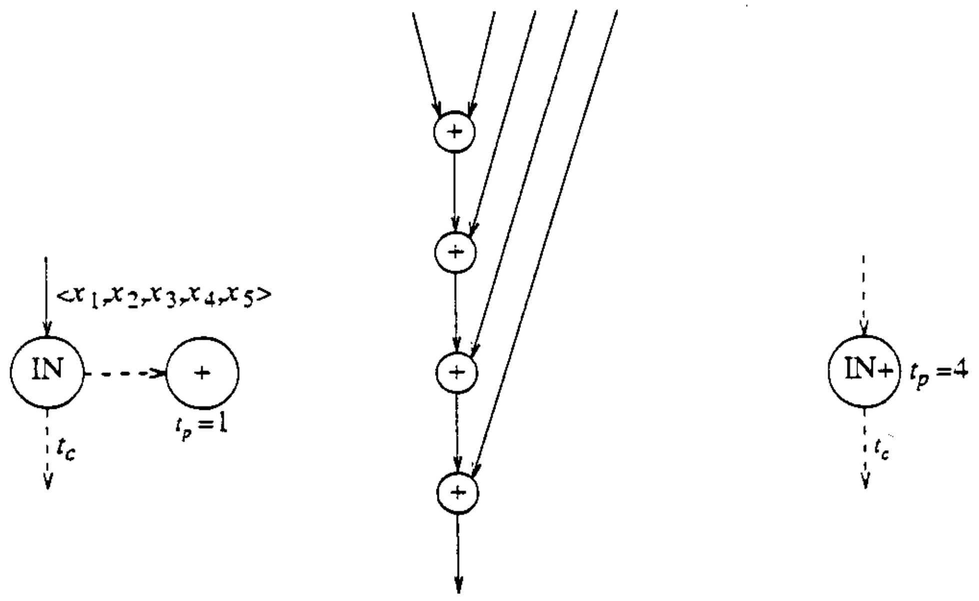


Figure 3.14 : Resolution of Insert (IN)

In Figure 3.16b the functional form is completely unfolded, with as many copies of the function as there are elements in the structure. For example, assuming $t_c = 5$ and $t_p = 1$ cycle for the functions in the graph, the best case response time for the functional form is 13 cycles.

When the functional form is completely unfolded it is clear that if the number of processors available is smaller than the number of data elements in the structure then the nodes will not be executed concurrently and the optimal response time (13 cycles in our example) will not be achieved. Therefore, when the number of elements in the structure exceeds the number of processing elements available, it is no longer

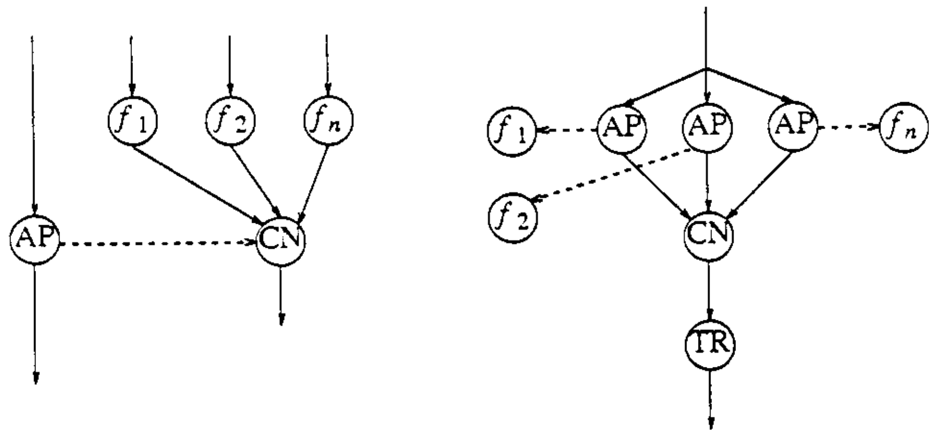


Figure 3.15 : Functional Transformation

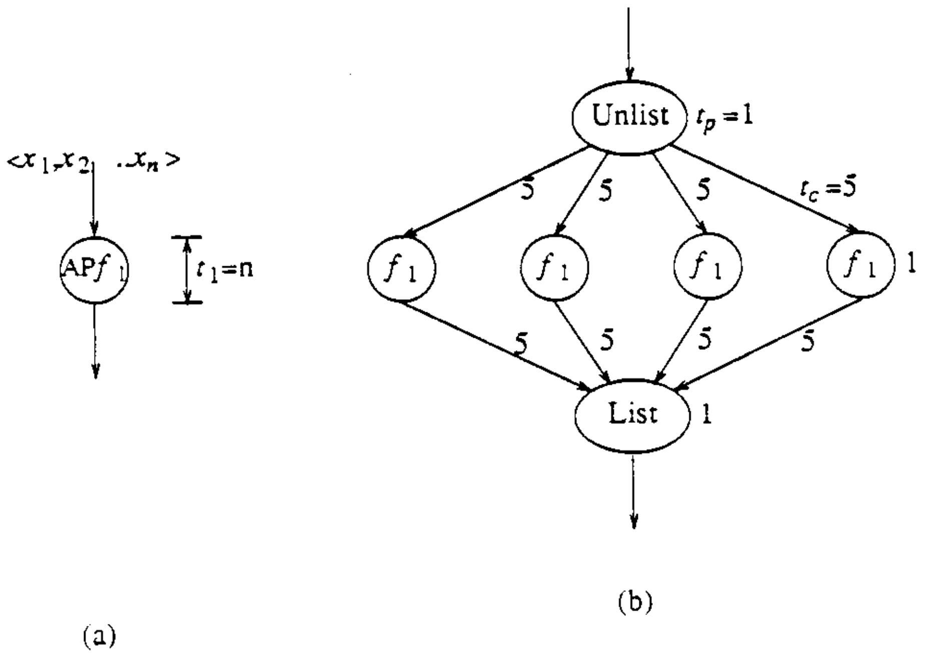


Figure 3.16 : Models for the Execution of Apply-to-All (AP)

worthwhile to expand the *AP* completely. Instead we unfold the functional form to the extent to which it can be efficiently executed. An additional parameter which we utilize is the degree of concurrency (k) exploited from apply-to-all's. The structure is decomposed into k parts and each part is executed sequentially (Figure 3.17). For example, if α=0.1 and β=5 (where $t_c=\alpha n+\beta$) then t_{c_1} in Figure 3.17 will be $(5+\frac{n}{4}*0.1)$ and the execution time for the functional form will be $2(5+\frac{n}{4}*0.1)+\frac{n}{4}+2$ cycles.

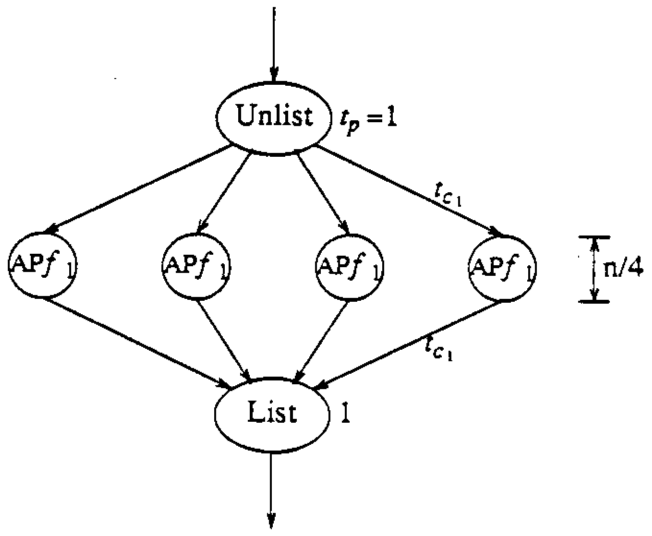


Figure 3.17 : Limited Expansion of Apply-to-All (AP)

In our implementation we compare the execution time of the sequential model with the restricted unfolding model and replace the functional form in the graph with the more efficient subgraph. As an example, when there are thirty elements in the structure (n=30) and the maximum degree of concurrency is four (k=4) then the execution time achieved from each of the two models (Figure 3.16a and 3.17) will be 30 and 21 cycles respectively. In this case the model exploiting limited concurrency is used. But with ten elements in the structure and the same maximum degree of concurrency, the execution times are 10 and 15 cycles respectively. Hence the subgraph to replace the macronode will depend on the maximum degree of concurrency to be exploited, the processing time of functions, and the communication time between instructions.

The functional forms are resolved hierarchically - the nested forms first and then the functional forms in the main graph. After each macronode is substituted by a subgraph the link to the auxiliary function graph is removed. The longest path (lp_i) of the nodes (i) in a graph are recomputed after any modification to that graph. The longest path of the root node is the processing time of the function represented by that graph.

When all the functional forms are resolved we obtain a single, directed data flow graph. Each node is a task containing a high level FPL function which is to be sequentially executed; and is self contained and complete. The arguments and results of the nodes specify the linking information between the tasks.

3.7 Partitioning

The partitioning problem we consider here is to group together tasks in order to reduce the complexity of allocation [MAR 79]. By introducing the partitioning step the number of groups (which will now be partitions instead of tasks) that have to be allocated, will decrease. An important criterion for the partitioning algorithm is that grouping together tasks should not adversely affect the performance or the allocation.

The partitioning algorithm is based on the critical paths in the data flow graphs. The critical path of a node (i) is the longest path (lp_i) from the node to the entry nodes on the top of the graph. The tasks on the longest paths are kept in one partition so that critical paths can be given priority during allocation. The longest path (lp_i) of each node (i) is the earliest possible time at which the result from that node will be available. The longest path of the root node is hence the best response time of the program. Associated with each node (i) in the graph is a field *longest path* (lp_i), and the critical path from a node is traversed by comparing the *longest paths* of the predecessor nodes, and going to the node which has the maximum *longest path.*

Starting at the root node the critical path of the directed acyclic graph is traversed and each node that is visited is placed in the first partition. If there is more than one critical path, then one of them is chosen arbitrarily. The nodes placed in the partition are marked as assigned. Now we consider the latest node assigned to a partition and examine its other paths along different edges. We traverse the next longest path along a different edge from the assigned node under consideration and

```
procedure traverse (i:typenode, p:integer);

{This recursive procedure traverses the longest path ($lp_i$) to
 the top of the graph from node[i] and places all the
 nodes traversed but not yet assigned, to a new partition}

begin
 partition(p) ← i;
 if node[i] has predecessors then
 begin
  find the predecessor node (k) with the maximum longest path
  traverse(k,p);
  for every other predecessor (j) of node[i]
  begin
   if j ≠ k then
   begin
    p:=p+1;
    traverse(j,p);
   end;
  end;
 end;
 remove node[i] from graph
end; {traverse}


procedure partition;

{This procedure partitions an acyclic directed graph G}

begin
 {initialize partition no. for first partition}
 p:=1;
 {traverse the critical paths beginning with the root node}
 traverse(root(G),p);
end; {partition}
```

Figure 3.18 : Graph Partitioning

place all nodes on this path which have not yet been assigned to a new partition. This step is repeated for each edge of the assigned node under consideration, in decreasing order of path length. Each assigned node is examined in the reverse order in which they were assigned, until all the nodes in the graph have been assigned to a partition. This algorithm will ensure that every connected node is assigned to a partition, and that no node will be assigned to more than one partition.

The algorithm for partitioning a graph is described in Figure 3.18. The partitioning of the graph of Figure 3.19 is demonstrated in Figure 3.20. In Figure 3.19 node numbers are listed beside the node, processing time inside the node and communication times on the arcs. In Figure 3.20 the time for the longest path (Section 3.3) of each node is shown beside the node. Note that the longest path (lp_i) of a node includes the time to communicate to its successors. The partitions are labelled in the order they are formed.

Associated with a partition are several timing parameters [LANG 77] obtained from the longest path time information in the graph. We associate various timing parameters with the partitions in order to capture the timing characteristics of the program graph. In the next chapter we will use this abstract model of the program graph, for allocation based on deadline constraints.

The *completion time* $(T_{\max})$, of each partition is the time by which the result from the last task in that partition should be available in order to achieve minimum response time execution of the graph. The *deadline* (T_d) is the completion time of the partition containing the longest path in the graph. The *deadline* (T_d) is the minimum

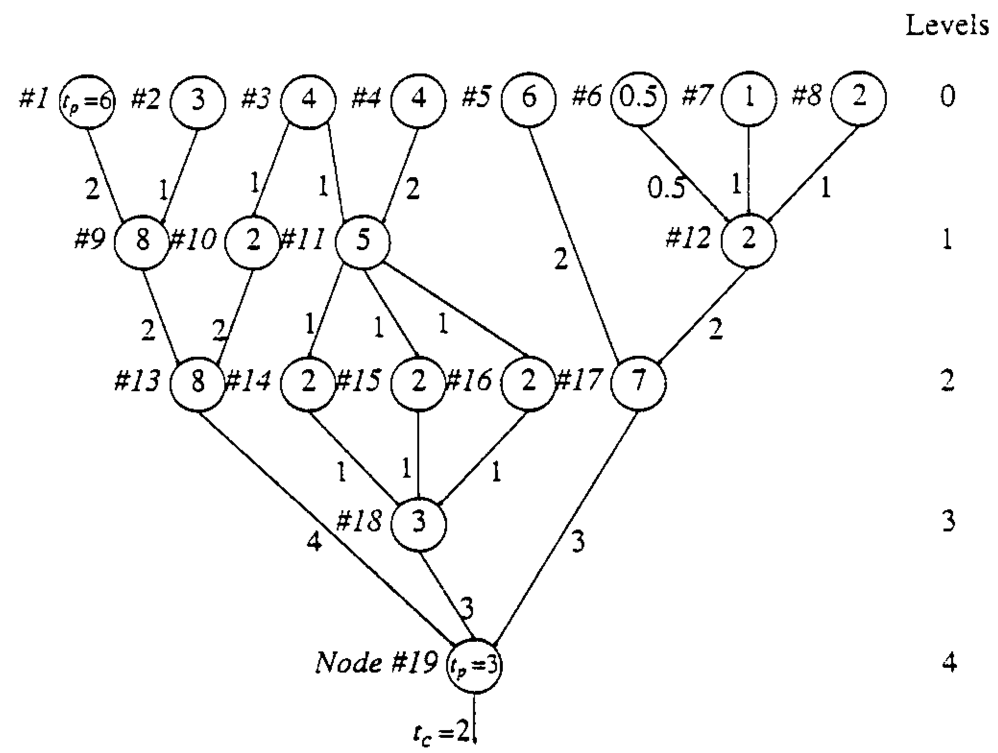


Figure 3.19: Labelled Graph with Processing and Communication time

response time for the program graph. During the execution, if the completion time for any partition is not met, then the deadline will not be met. The $T_{\max}$ for a partition is calculated by going to the result of the highest level node of the partition, i.e., the result of the last task in the partition. That is,

$(T_{\max})_{partition} = (lp_r - t_{p_r} - t_{c_r})$

where r is the result node of the highest level (Section 3.4) node in the partition.

If the highest level node of the partition has several result nodes, then

$T_{\max} = \underset{all\ res\ nodes(r)}{Minimum} \ (lp_r - t_{p_r} - t_{c_r}).$

For example the $T_{\max}$ for partition 4 in Figure 3.20 is $(lp_{19}-t_{p_{19}}-t_{c_{19}})$ for node 19 (refer Figure 3.19), which is equal to 35 - 3 - 2 = 30 time units. Another interpretation of $T_{\max}$ is that it is the maximum longest path of all the predecessors of the result node of the highest level node of the partition. Hence, the $T_{\max}$ of partition 4 is the maximum of the longest paths of nodes 13, 17 and 18 which is 30 time units.

The *earliest initiation time* (T_{EI}) is the time at which the execution of the partition can begin. The *earliest initiation time* (T_{EI}) is specified by its bounds, $\underline{T_{EI}}$ and $\overline{T_{EI}}$.

If the partition extends to the top of the graph i.e. it has a node at level 0, then it can begin execution at t=0, as the input to the program is available initially. In this case $\underline{T_{EI}}=\overline{T_{EI}}=0$. But if the initiation of a partition depends on the result of a previous partition, then the *earliest initiation time* will reflect the flexibility of execution of the previous partition. Hence $\underline{T_{EI}}$ is the earliest time when all the inputs to the first node in the partition can arrive; and $\overline{T_{EI}}$ is the latest time when the inputs to the first node arrive.

We can compute $\underline{T_{EI}}$:

$\underline{T_{EI}} = (lp_k - t_{p_k} - t_{c_k})$

where k is the lowest level node of the partition This is equivalent to defining $(\underline{T_{EI}})$ as:

1. If the lowest level node in the partition is at level 0, then $\underline{T_{EI}} = 0$

2. Else $\underline{T_{EI}}$ = Max (longest path of predecessor nodes) of the lowest level node

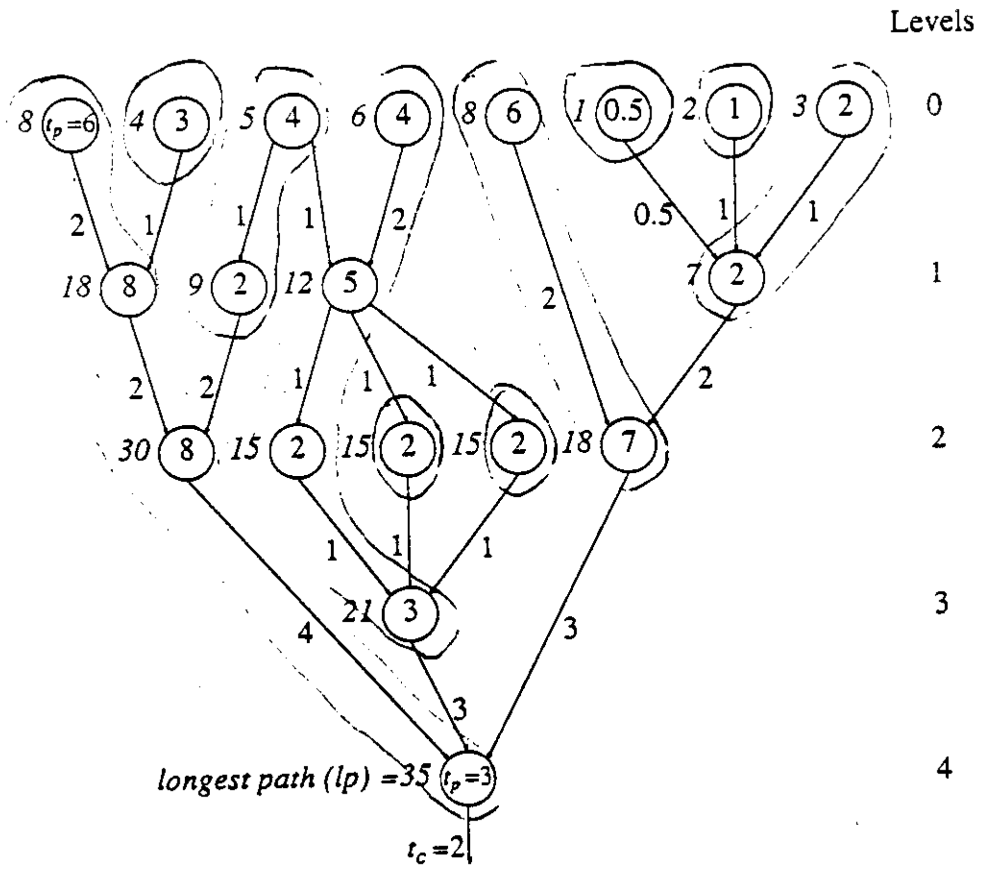


Figure 3.20 : Partitioning of Data Flow Graph

in the partition.

The upper bound ($\overline{T_{EI}}$) is :

1. If the lowest level node in the partition is at level 0, then $\overline{T_{EI}} = 0$

2. Else it is the latest time when the predecessor partition can delivers its results.

Delta (δ) of a partition is a parameter such that $\delta = (T_{\max} - T_E)$, where T_E is the *execution time* of the partition. If each partition is completed by its *completion time*, then the nodes of the partition will never have to wait for predecessors once the execution of that partition has begun. Hence $(\delta - \overline{T_{EI}})$ is the flexibility in starting the execution of a partition, without violating the deadline.

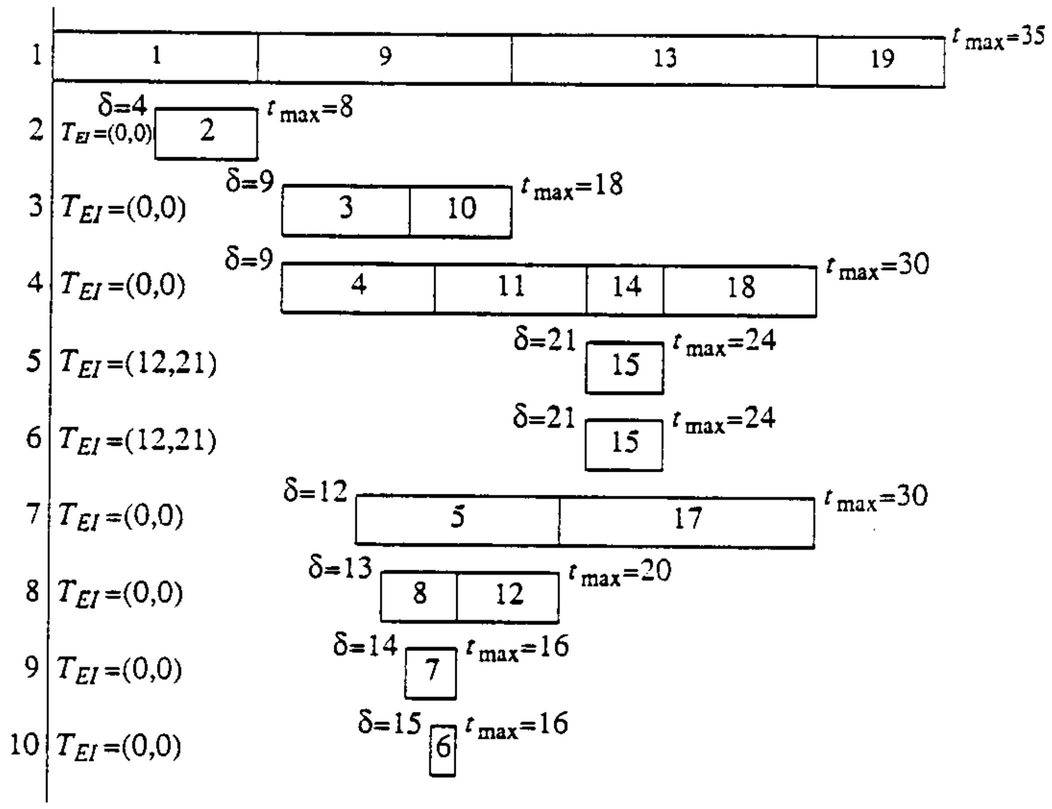


Figure 3.21 : Gantt Chart

The timing constraints of the partitions can be represented by a modified Gantt chart. The partitions of Figure 3.20 are illustrated in Figure 3.21 with the timing parameters. $T_{\max}$ is the time by which the execution of the partition should be completed. In Figure 3.21, any shift of a partition to the right of $T_{\max}$ will result in non-minimal response times. The partition can be shifted between $\overline{T_{EI}}$ and δ, i.e., the partition can be scheduled to start at any time between these two times. The modified Gantt chart represents the timing parameters associated with partitions in order to execute the program in minimum time, i.e., by the deadline (T_d).

Thus we have obtained an abstract timing model of the FPL program graph after translation to a dependence graph, graph reduction and partitioning. In the next chapter we use this abstract timing representation of the program and allocate the partitions based on deadline constraints.

CHAPTER IV

Allocation

4.1 Introduction

Allocation is an attempt to effectively utilize the available processors to satisfy the processing requirements of a computation. Partitions which were formed by grouping tasks are assigned to the available processors. The performance objective is to minimize of the response time. It is assumed that token traffic will be within acceptable limits so that no additional overhead is incurred due to the congestion of the intercommunication network. It is also assumed that storage capacity of a processor is large enough and will not be a constraint in allocating partitions to processors. All the processors in the architecture under consideration are identical in processing capabilities.

In our model we attempt to satisfy the deadline requirement to achieve the minimum response time and in order to do so, each task has to meet its schedule. In a data flow model, tasks can only be allocated to a processor and cannot be deterministically scheduled. This is because the sequencing is done by the data flow activation mechanism which activates tasks based on data dependencies alone. In such circumstances several tasks in a processor may be activated at the same time and the order of execution chosen is random. This is because unlike the program counter concept in von-Neumann machines, the order of task execution in each processor is not fixed. All tasks which have been activated compete for execution in the processor, and the outcome is non-deterministic as no priority scheme is enforced. In order to meet the deadline requirement with the given number of processors, it may be necessary to give priority to some tasks during execution. Moreover due to the unequal processing time of tasks, it is sometimes desirable not to obey the data flow activation mechanism, and execute tasks in an order different from the order they were activated. This is so that tasks on the critical path can be given priority in execution. Hence we need an additional scheduling mechanism to influence program execution.

Task preemption is not permitted due to the assumption that task switching involves a large amount of overhead. Introduction of *priorities* for tasks will suit the purpose of selecting a particular task for execution when several are activated; but will not allow keeping the processor idle even though tasks are ready. Any separate mechanism for scheduling will introduce an additional level of control which has to be implemented in the architecture and is undesirable. Later in this chapter (Section 4.3.4) we present a method of forcing execution in favor of critical paths based on the introduction of additional dependencies.

The two problems we deal with in this chapter are:

1. Obtain the minimum number of processors, and the corresponding allocation required to complete the computation within the deadline, which is the critical

path in the graph.

2. Given the number of processors available, find the minimum response time and the corresponding allocation for the computation.

We first formulate the allocation problem and then discuss the details of the allocation algorithm.

4.2 Problem Formulation

Let the program graph be represented by n partitions $S_1, S_2 .... S_n$. Each partition, as discussed in the previous chapter, is represented by the parameters (Figure 4.1) *earliest initiation time* (T_{EI}), *completion time* ($T_{\max}$) and *execution time* ($T_E = T_{\max} - \delta$).

All the precedence information of the partitions is captured by the timing constraints of the model. The objective is to obtain an allocation $\{P_i\}$, $1 \le i \le k$, given k processors such that P_i = {Set of all j | S_j is allocated to processor i}. When the partitions are allocated to the processors, each processor has similar timing parameters associated with it.

Initially each partition is assigned to a processor as in Figure 4.2. The allocation algorithm reduces the number of processors required by allocating to one processor partitions assigned to two different processors. This can be achieved if the partitions when combined together still satisfy the timing constraints. A heuristic algorithm is developed to solve the problem which is essentially a bin packing

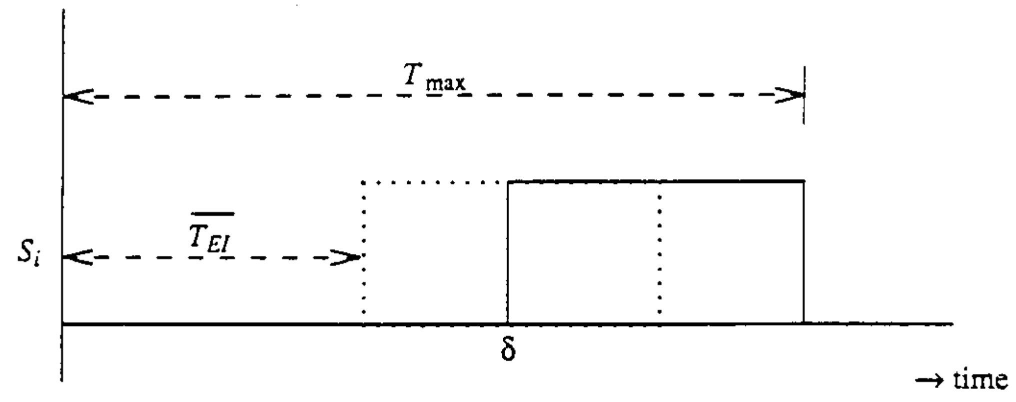


Figure 4.1 : Timing Parameters of A Partition

problem with timing constraints.

4.3 Allocation for Minimum Response Time

In this section we discuss an algorithm for allocation of partitions to indefinite number of processors for execution in minimum time. The algorithm heuristically reduces the number of processors required for execution in minimum time.

4.3.1 Formation of Groups

Initially partitions $P_1, P_2 \cdots P_k$ are divided into distinct groups depending on their position on the time scale in the Gantt chart (Figure 4.3). Partitions which exhibit temporal locality face similar deadlines and timing constraints, and hence can

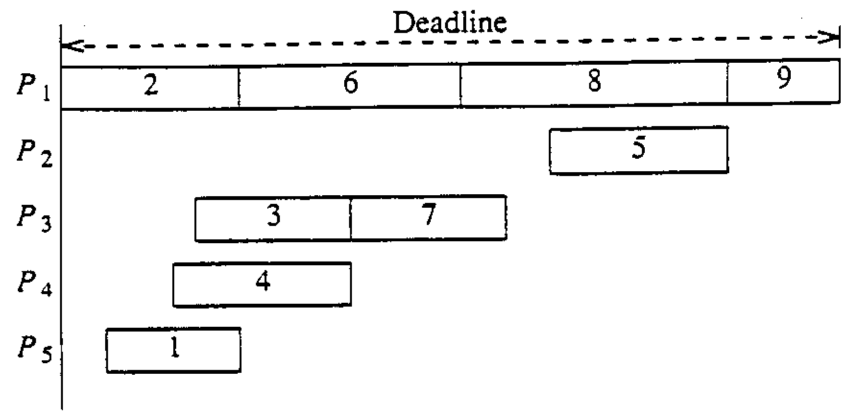


Figure 4.2 : Initial Assignment of a Single Partition to a Processor

be clustered together into a group. This also facilitates the allocation, as groups with immediate deadlines can be considered first and assigned to processors.

Partitions are sorted according to T_{max}, and starting from the smallest T_{max} partitions are placed in group 0.

A new group is started if the T_{max} of the last partition assigned to the old group is greater than the δ of all the partitions not yet assigned. The critical partition (S_0) is considered separately and is assigned to group 0.

The allocation is done group by group starting with the partitions of Group 0. The partitions in a group with a larger index are those not required to execute until later in time.

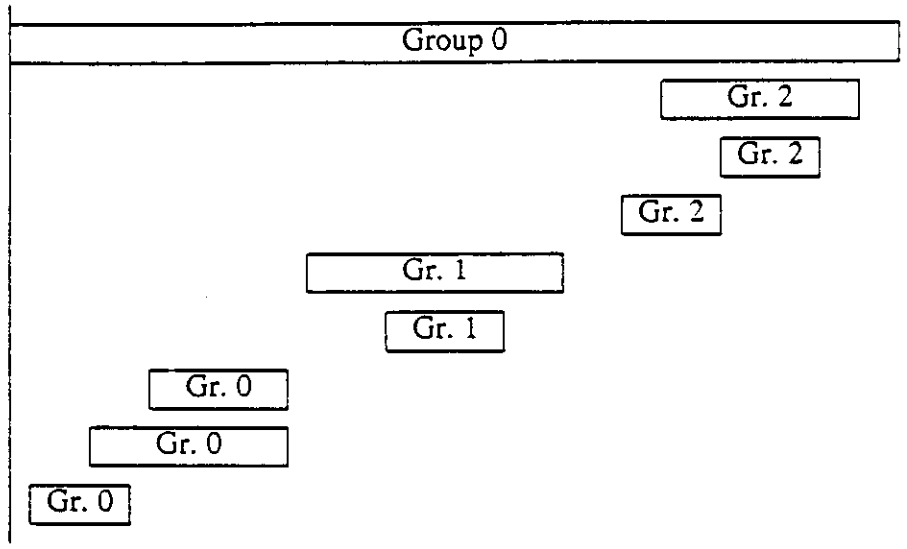


Figure 4.3 : Division of Partitions into Groups

4.3.2 Assignment of Partitions

We can reduce the number of processors by allocating partitions to the same processor, if the individual partitions in the new processor continue to satisfy their individual timing constraints. Two partitions can be assigned to a processor if their results will be available by their respective completion times ($T_{\max}$).

Consider the partitions in Figure 4.4, where we attempt to combine partitions S_2 and S_3 in the processor P_2. As both partitions can be activated at t=0 (i.e., T_{EI} =(0,0)), their order of execution is random and cannot be predicted. In processor P_2 either task 2 or tasks 3-4 can execute first. If task 2 executes first then partition S_3

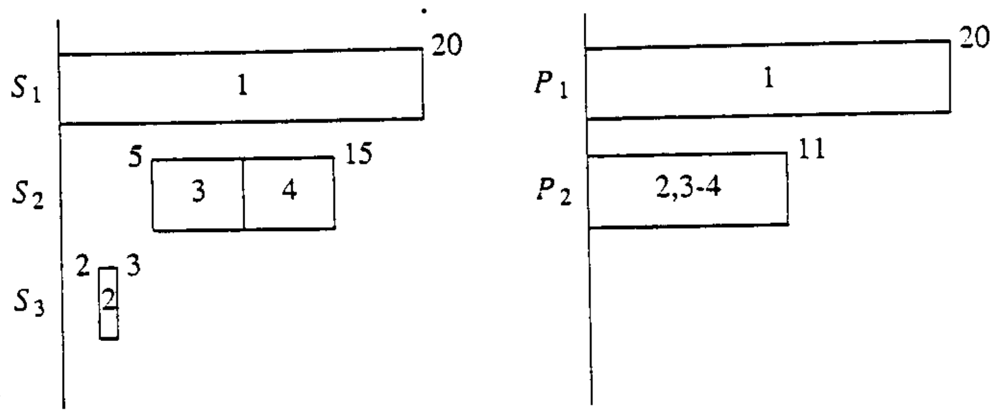


Figure 4.4 : Improper Combination of Partitions In a Processor

will complete execution by 1 unit, and S_2 will finish by 11 units. As 1 is less than $T_{\max_3}$=3, and 11 is less than $T_{\max_2}$=15, hence each of them satisfies its deadline. But if tasks 3-4 execute first, then partition S_2 will complete at 10 units, and S_3 will be completed by 11 units. As S_3 completes execution at 11 units which is greater than $T_{\max_2}$=3 units, the completion schedule will not be met. We cannot permit S_2 and S_3 to be allocated to the same processor as the deadline constraints are violated.

On the other hand in the Gantt chart of Figure 4.5, partitions S_2 and S_3 can be allocated to the same processor, because whatever order they execute in, they will be completed by their individual $T_{\max}$.

The criterion for allocating a partition S_j (S_3) with the given partition S_i (S_2) in processor P_i (P_2) is:

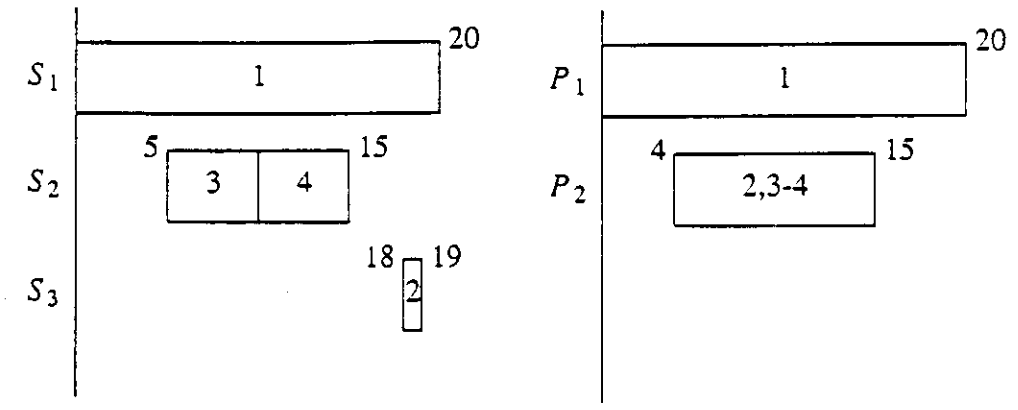


Figure 4.5 : Combination of Partitions into a Processor

1. $T_{\max_i} < T_{\max_j}$ $(T_{\max_2} < T_{\max_3})$

2. $\delta_i \geq \overline{T_{EI_j}} + (T_{\max_j} - \delta_j)$ $(\delta_2 \geq \overline{T_{EI_3}} + (T_{\max_3} - \delta_3))$

If both conditions are satisfied then the number of processors can be reduced as in Figure 4.5. The first condition ensures that the deadline (T_d) will be met. If T_{EI_3} = (0,0) then the second condition reduces to $\delta_2 \geq (T_{\max_3} - \delta_3)$, which is to check if the partition can be packed in the available time slot.

For the new processor the *completion time* is $T_{\max} = T_{\max_i}$ $(T_{\max_2})$,i.e., the smaller of the two completion times. The new δ of the processor is $\delta = \delta_2 - (T_{\max_3} - \delta_3)$. The new *earliest initiation time* (T_{EI}) of the processor P_2 is made equal to the larger of the initiation times of S_2 and S_3.

4.3.3 Allocation of Group 0 Partitions

We begin with the partitions of group 0. The partitions are ordered by increasing $T_{\max}$, and further separated by those with $T_{EI} \neq (0,0)$, and those with $T_{EI} = (0,0)$.

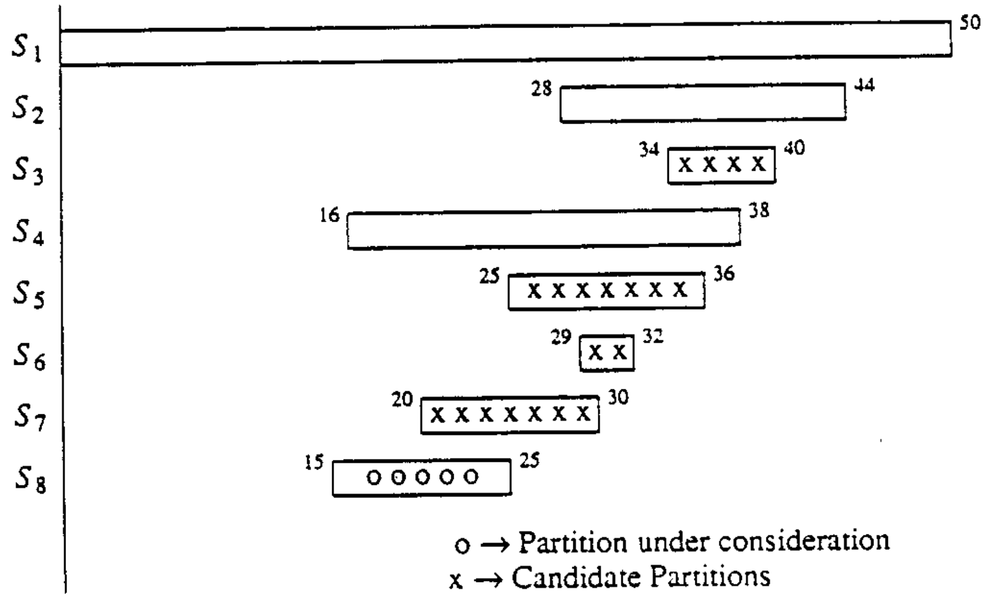


Figure 4.6 : Identification of Candidates for Allocation with Partition S_8

We first choose for allocation a partition with $T_{EI} \neq (0,0)$, and of these partitions one with minimum $T_{\max}$. The partitions which cannot be initiated at the beginning of the computation are favored, because they have more stringent constraints, which makes it difficult to allocate them later. This heuristic ensures that partitions whose timing constraint is more stringent is allocated first.

Having chosen a partition we identify all possible candidates which can be allocated to the same processor. The criterion for candidacy of a partition to be allocated along with existing partitions in a processor are the same two conditions as discussed in Section 4.3.2. The identification of candidates is illustrated in Figure 4.6.

Amongst the candidates, some of the partitions are selected for allocation with the partition under consideration in the processor. The candidates are ordered according to decreasing execution times of the partitions. The larger partitions are allocated first, and then as many smaller ones in the order of decreasing partition size as possible (Figure 4.7). This is because if the smaller partitions are allocated first then the available time will be split up and fragmented, thus leaving no time slot large enough for larger partitions.

After considering the partition with $T_{EI} \neq (0,0)$ and the minimum $T_{\max}$, and combining other partitions with this partition in a processor, we repeat this process for the remaining partitions in group 0. Each partition in group 0 which has not yet been allocated is considered, in increasing order of their $T_{\max}$. When the number of processors required for group 0 partitions cannot be further reduced then we will have completed allocation of group 0 partitions to processors.

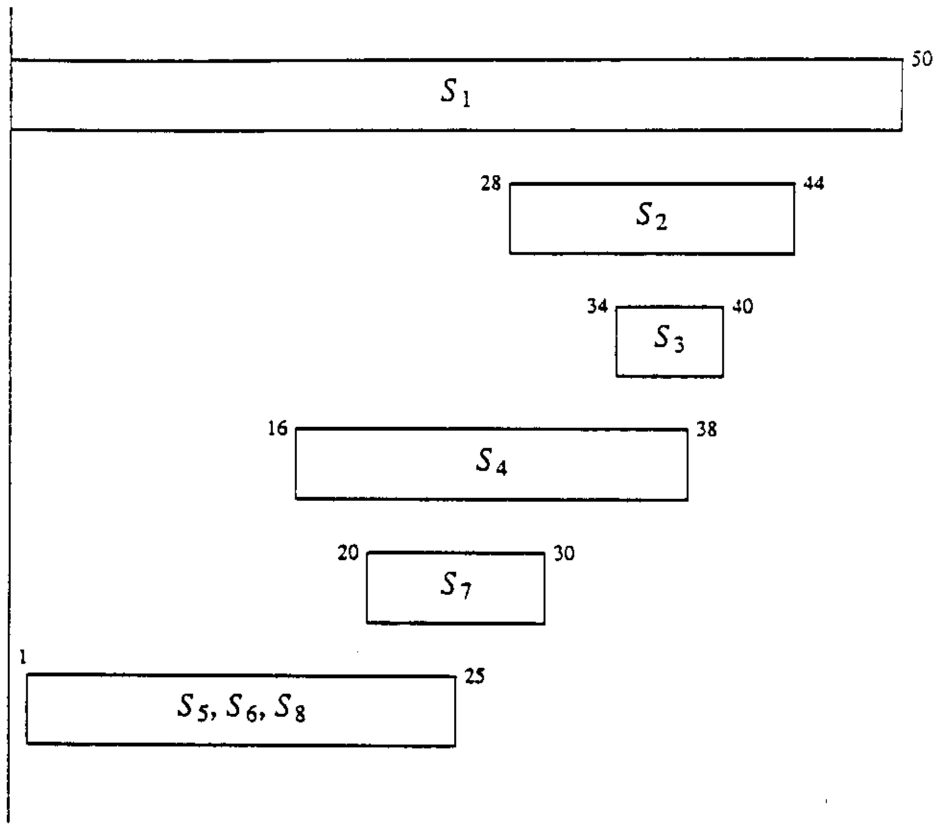


Figure 4.7 : Combination of Partitions with S_8

4.3.4 Allocation of Remaining Groups

We now examine the partitions of the next group. We allocate the partitions of this group to the existing processors.

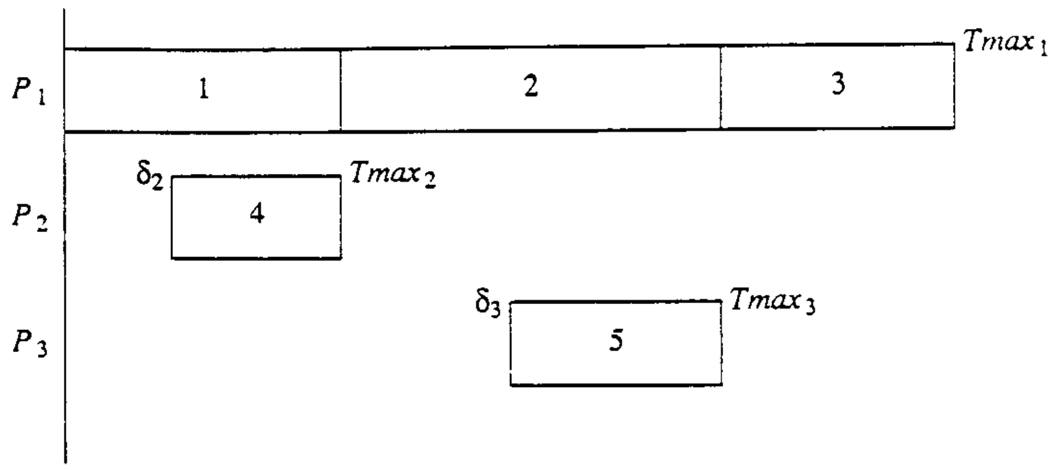


Figure 4.8 : Allocation of Graph (Figure 4.9a) to 3 processors

The partitions of this group are sorted according to their size (i.e. execution times). The processors are ordered in decreasing order of their δ. The largest tasks of this group are allocated to the processors with the smallest δ to get heuristically a good fit. The criterion for combining partitions with the existing partitions in the processor are the same two conditions of Section 4.3.2. After we try to fit all the partitions of the group into slots available in the processors, there may still be some partitions left in the group which have not yet been allocated.

The attempt to allocate the remaining partitions to the processors illustrates the problem of forcing schedules in a data flow environment. Consider the situation shown in Figure 4.8. A strict data flow approach will require three processors to execute the above tasks within the deadline (T_d). Intuitively, one would expect it to be possible to allocate task 5 to the processor containing the task 4, as here is a time slot after task 4 which task 5 can occupy. But we have a problem here due to the lack of a specific order of execution in data flow. Both tasks 4 and 5 are activated at the same time (t=0) and hence can be executed in random order. If task 5 is executed first, task 4 will be unable to satisfy its completion time.

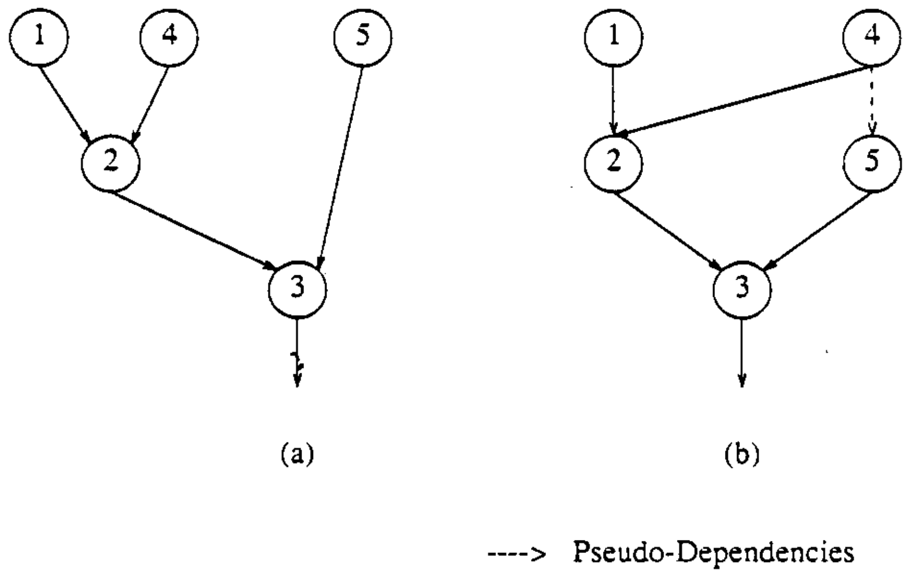


Figure 4.9 : Introduction of Pseudo-Dependencies in Graph

We note that task 5 is activated much before its results are needed. To force an order of execution in data flow we introduce *pseudo-dependencies* between tasks. *Pseudo-dependencies* are additional dependencies which are introduced in the graph. Just as data dependencies in the graph are implemented by the use of data tokens, similarly the pseudo-dependencies are enforced by control tokens. Control tokens are similar to data tokens except they do not contain any data values.

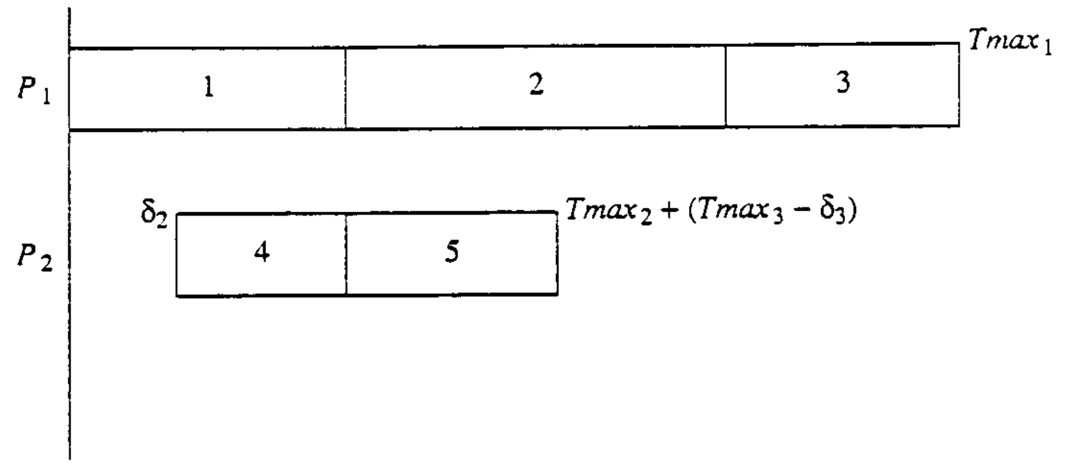


Figure 4.10 : Allocation of Graph (Figure 4.9b) to 2 processors

The graph of Figure 4.9a can be transformed by adding a pseudo-dependency between nodes 4 and 5 (Figure 4.9b). This graph can now be allocated to two processors (Figure 4.10), compared to three processors required for the graph of Figure 4.9a.

This solution is quite satisfactory except for the increased token traffic due to additional dependencies in the graph. This overhead becomes more serious if the processor has already been allocated several partitions, as we have to introduce a dependency between each of the partitions in the processor and the new partition being allocated to the processor. Each dependency results in additional traffic due to control tokens.

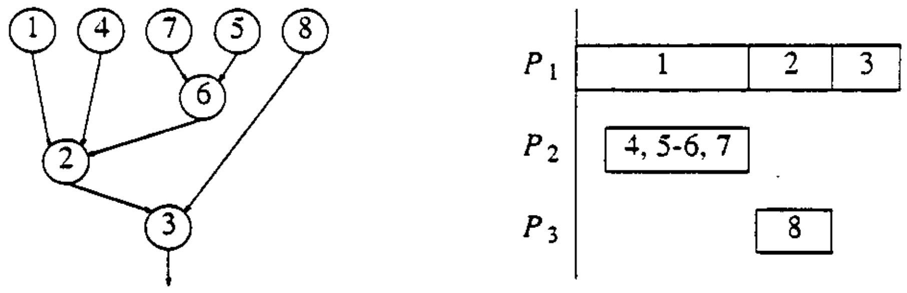


Figure 4.11 : Graph and Gantt chart to illustrate Proliferation of Tokens

For example, in Figure 4.11, processor P_2 has been allocated three partitions 4, 5-6 and 7. To allocate partition S_3 containing task 8 to the same processor, we have to ensure that the existing partitions in the processor execute before task 8. In order to do so, we will have to introduce dependencies between every partition in P_2 and the partition S_3. The graph of Figure 4.11 is transformed to Figure 4.12, spawning a large number of control tokens.

In such cases we take advantage of the critical path which ranges over the entire time scale from t=0 to the deadline (T_d). The critical partition contains a single path which is the critical path, and each node on this path can be treated as a reference point on the time scale. Instead of making the newly allocated partition dependent on all the previously allocated partitions in the processor, it can be made

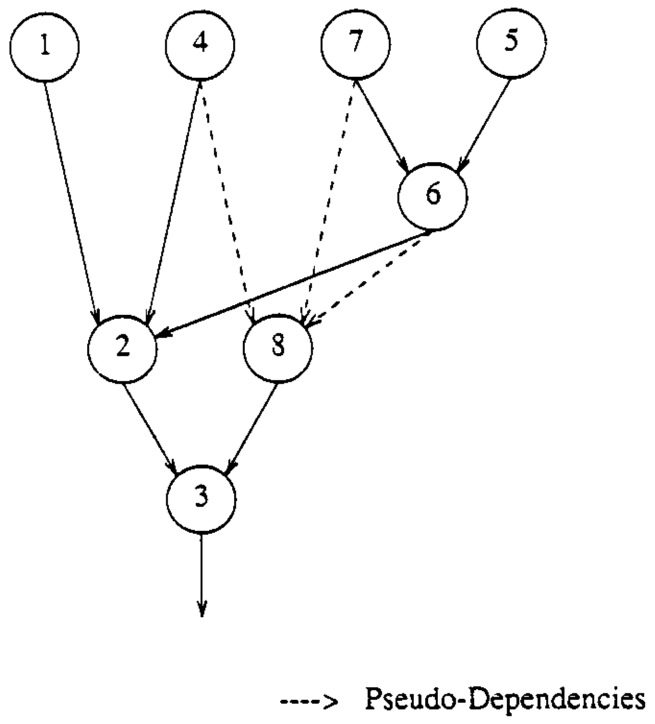


Figure 4.12 : Previous Graph with Large Number of Pseudo-Dependencies

dependent on a single node on the critical path. We look for a reference point on the critical path closest to and greater than or equal to the completion time ($T_{\max}$) of the processor. Figure 4.13 illustrates the solution adopted to avoid introducing a large number of control dependencies. The new execution time of the processor ($T_{\max}-\delta$) contains an idle period caused by the availability of only discrete reference points on the critical path to synchronize nodes.

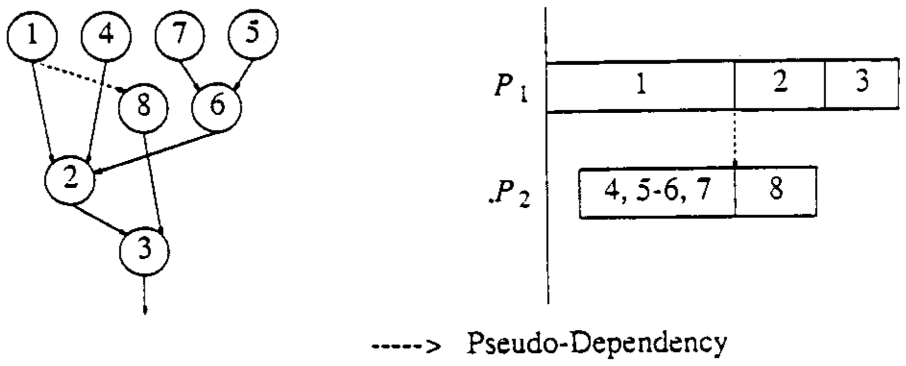


Figure 4.13 : Graph and Gantt chart with Single Pseudo-Dependency

The tasks left in the group which have not yet been allocated, are now allocated to the existing processors with additional dependencies. This is done as follows. Initially we identify a time (T_{dep}) for each processor which is greater than or equal to its $T_{\max}$. The dependency point (T_{dep}) is the closest time on the critical path after the $T_{\max}$ of the processor under consideration. The processors are now ordered according to their T_{dep}. The partitions in the group yet to be allocated are sorted according to their $T_{\max}$. We select the partition of the group with the largest $T_{\max}$ for allocation to the processors having the largest T_{dep}. But before the actual allocation of the chosen partition of the group to the processor, we examine if any other partitions can be allocated in the time slot between the $(T_{dep})_{processor}$ and the $\delta_{chosen\ partition}$. The order of execution of partitions is enforced by adding control dependencies and hence the condition 1 of Section 4.3.2 for the combination of processors i.e. the condition on $T_{\max}$ can be relaxed. Therefore we select partitions from the group that will fit the time slot ($\delta_{chosen\ partition}$ - $(T_{dep})_{processor}$), and allocate them along with the chosen partition to the processor.

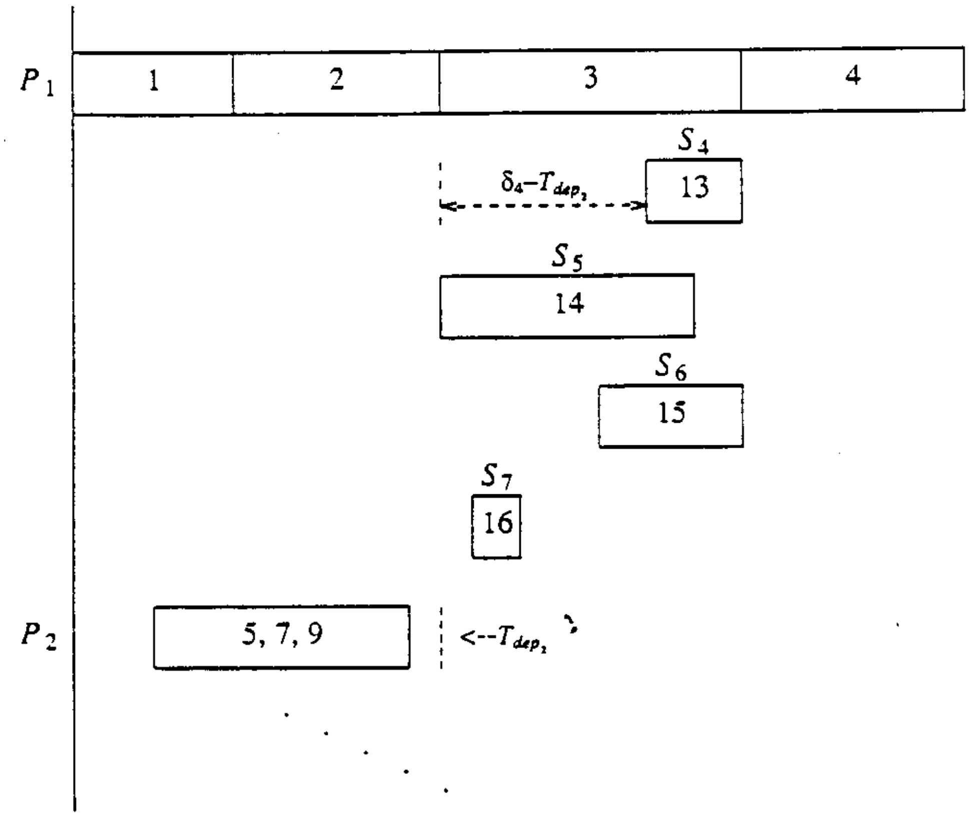


Figure 4.14 : Choosing Partitions to combine with S_4 in Processor P_2

In Figure 4.14 the partition S_4 containing task 13 is chosen for allocation with processor P_2. But before allocating the partition S_4 to P_2, we look for other partitions which can be fit into the time slot ($\delta_4 - T_{dep_2}$). Partitions S_6 and S_7 can be combined with S_4 and allocated to P_2 after introducing control dependencies between task 2 and 16, task 16 and 15, and task 15 and 13. The resulting allocation is shown in Figure 4.15.

This process is repeated for all the existing processors, and if there is still a partition left which cannot be allocated then a new processor is created for that partition.

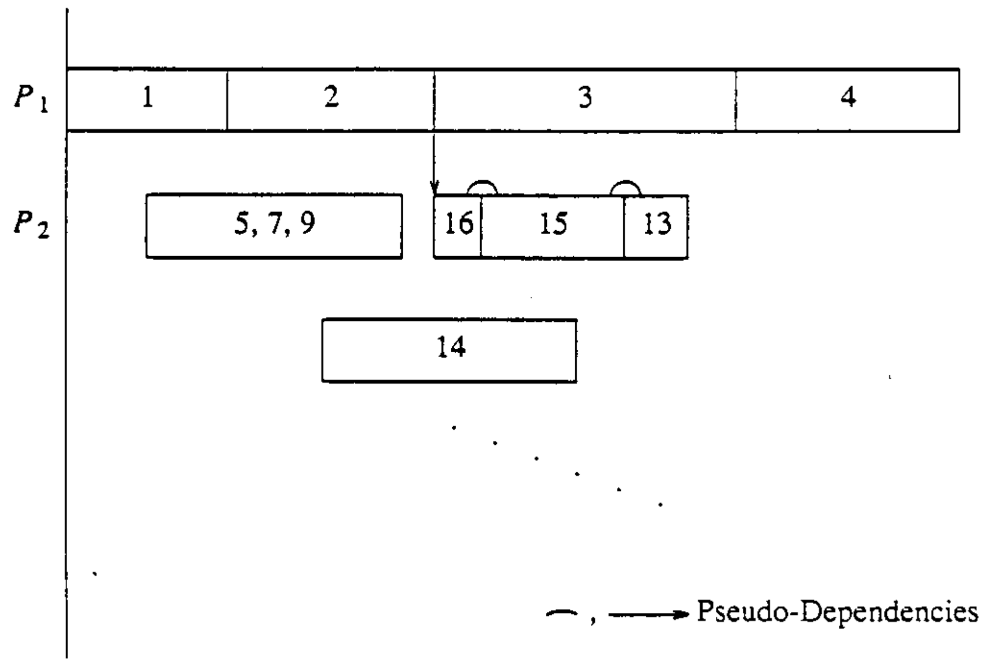


Figure 4.15 : Combination of Partitions of other Groups with partitions of Processor P_2

This procedure for allocation is repeated for each group, till all the groups have been allocated. Finally, we obtain an allocation requiring k processors, and which assures program execution in the minimum response time. The number of processors (k) is not optimal but rather the minimum required by the heuristic algorithm.

To summarize the algorithm for allocation for minimum response time :

1. Initially partitions are divided into groups. Partitions are allocated group by group.

2. We start with group 0 partitions. Partitions are seperated into those with $T_{EI} \neq (0,0)$, and those with $T_{EI} = (0,0)$ respectively. These two subdivisions are ordered according to increasing $T_{\max}$.

3. We choose the first partition on the above list and identify all possible candidate partitions which can be allocated with that partition in the processor. The candidates are ordered according to decreasing execution times of the partitions and are chosen for allocation with the partition if the time constraints are not violated.

4. Step 3 is repeated for all partitions belonging to group 0 till all of them have been allocated.

5. Now the remaining groups are allocated. We start with the next group. The partitions of this group are sorted according to decreasing execution time. The partitions with largest execution time are allocated to the processors with the smallest δ.

6. The remaining partitions of the group (from step 5) are allocated after imposing additional dependencies. Partitions of the group with the largest $T_{\max}$ are

allocated to processors with the largest T_{dep}. However before allocating the partition with largest $T_{\max}$, we examine if any other partitions can allocated in the time slot between the $(T_{dep})_{processor}$ and the $\delta_{chosen\ partition}$. Step 6 is repeated till there are no unallocated partitions left in the group.

7. Steps 5 and 6 are repeated for all groups.

4.4 Allocation For a Fixed Number of Processors

In this section we allocate the partitions to a given number of processors. If the number of processors given is greater than or equal to the number of processors obtained in the unlimited case then the response time will be minimum, and we can use the same allocation as obtained in the previous case. If the number of processors is smaller, then the response time will no longer be equal to the time taken by the critical path in the program.

The algorithm for a fixed number of processors is similar to the previous algorithm. While partitions are being allocated we keep track of the number of processors being utilized. Whenever we require another processor for a partition, in order that its completion time ($T_{\max}$) be satisfied, we check to see if a processor is available.

If another processor is required to satisfy the *completion time* ($T_{\max}$) of a partition, and it is not available then the *deadline* (T_d) can no longer be met. This situation arising out of the limited resource restriction is dealt with in two different ways.

If the *earliest initiation time* (T_{EI}) of the partition under consideration is non zero, then the partition is placed in the same processor as the partition on which it is dependent (i.e. the argument partition). This is done in order to avoid creating a large number of additional dependencies.

If the *earliest initiation time* (T_{EI}) of the partition is zero, then we look for the largest gap (δ) of all the processors, and even though the execution time of the partition is greater than the δ of the processor, we still insert it.

In both cases the $T_{\max}$ of the processor has to be shifted to the right in the Gantt chart, to accommodate the partition. By inserting the partition, all the processors and partitions shift on the time scale by an amount ρ.

When the *earliest initiation time* (T_{EI}) is non zero, the shift in the time scale (ρ), is equal to the difference between $(T_{\max}-\delta)_{partition}$ and the space available in the processor. In the second case the shift is equal to the difference of $(T_{\max}-delta)_{partition}$ and $\delta_{processor}$.

In Figure 4.16, when the initiation times are zero, only two processors are available, and hence we assign partitions S_2 and S_3 to the same processor even though the completion time for S_2 is violated. In the resulting Gantt chart:

$\delta_{n_1} = (T_{\max_3}-\delta_3)-\delta_2$

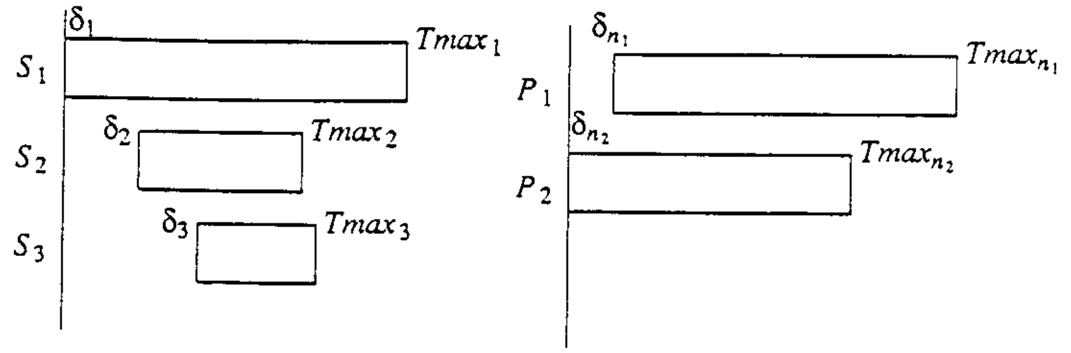


Figure 4.16 : Shift of Deadline under Limited Resources

$$T_{\max_{n_1}} = T_{\max_1} + (T_{\max_3} - \delta_3) - \delta_2$$

$$T_{\max_{n_1}} = T_{\max_2} + (T_{\max_3} - \delta_3) - \delta_2$$

The *deadline* (T_d) and all the *completion times* $(T_{\max})$, shift by the difference of the partition and the size of the largest gap.

This gives us the allocation for a fixed number of processors and the corresponding response time of the program.

CHAPTER V

Performance Evaluation

5.1 Overview

In this chapter we illustrate the partitioning of a program in FPL and its allocation to varying number of processors. Statistics are obtained by executing different programs and studying the effect of changing parameter values on the response time. To demonstrate the performance of the algorithm, programs with upto 1730 nodes were run on a simulated timing model of a high level data flow machine. Finally the results obtained are interpreted and its characteristics accounted for.

5.2 An Example

To demonstrate the partitioning and allocation system, we take the example of the multiplication of a (2X3) matrix by a (3X2) matrix.

An FPL program for matrix multiply is:

IP = CM(AI+, AP, TR);*

Main = CM(APAPIP, APDL, DR, [SL1, CM(TR, SL2)]);

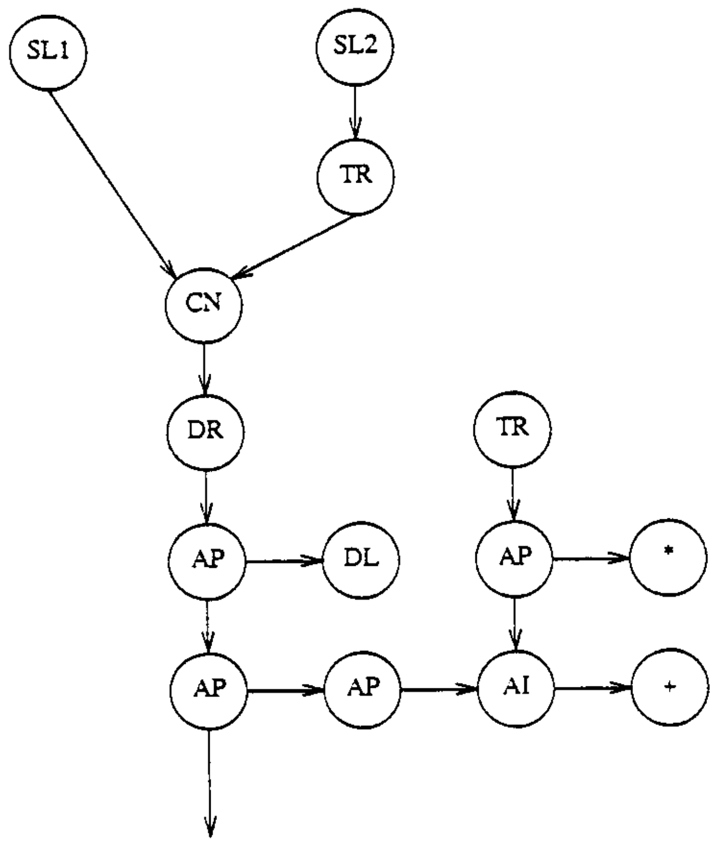


Figure 5.1 : Initial Program Graph for (2X3) by (3X2) Matrix Multiply

;

The FPL program is parsed into the corresponding CDF form, and an initial program graph (Figure 5.1) is obtained. This graph contains primitives such as +, *, *Transpose (TR)* etc. and functional forms such as *associative insert (AI) and apply-to-all (AP).*

The processing time (t_p) of the nodes is estimated by the number of cycles required to execute the instructions in each node. The functions used in this example and their estimated execution times are listed in Table 5.1.

Table 5.1: Functions Required for Matrix Multiply and Estimated Execution Times

Function	Execution Time
SLk (Select)	0.1
CN (Construct)	0.1
DR (Distribute Right)	1
DL (Distribute Left)	1
+	1
*	3
TR (Transpose)	5

The communication time parameters given are α=0.2 and β=1. The communication time therefore is :

t_c=0.2n+1 where

n is an indication of the amount of data in the result and is equal to the number of structures passed in the result token.

To obtain the number of structures passed in the result token from each node, the input data structure is symbolically interpreted. The symbolic input to this program is:

(((x,x,x),(x,x,x)),((x,x),(x,x),(x,x)))

The functional forms *associative insert (AI) and apply-to-all (AP)* are resolved based on the size of the input structure to the functional form and the processing times and communication times of the nested function.

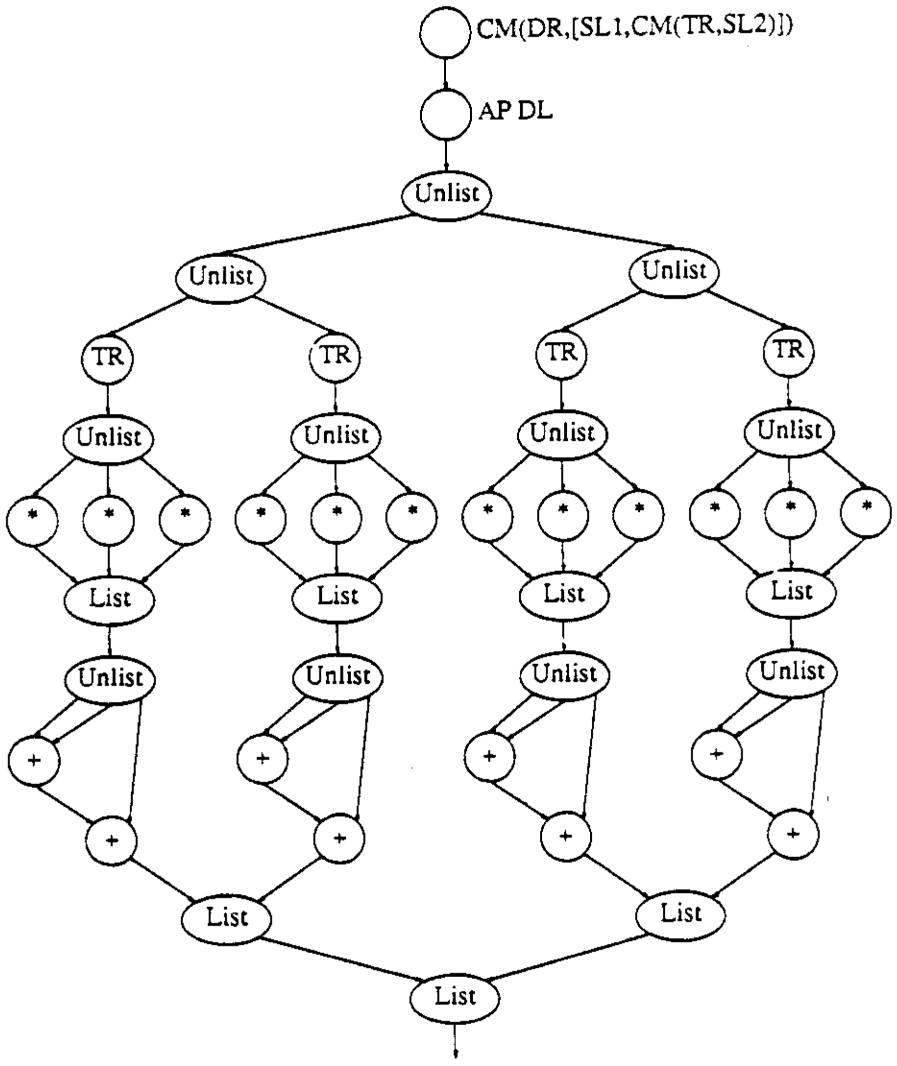


Figure 5.2 : Program Graph after Resolution of Functional Forms

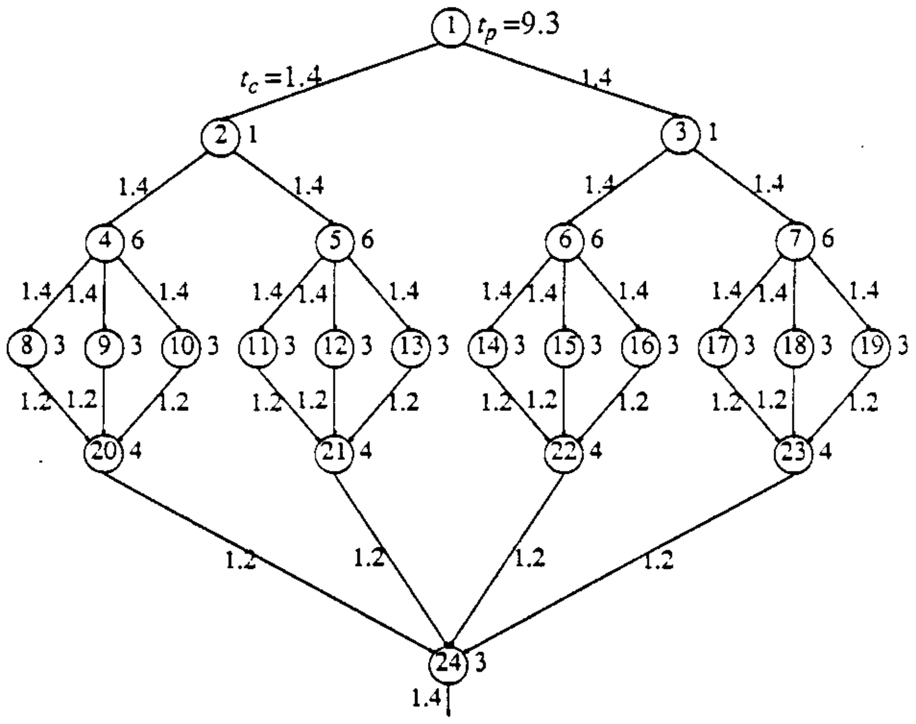


Figure 5.3 : Data Flow Graph with Processing and Communication Times

Another parameter for resolving the *apply-to-all (AP)*, is the degree of concurrency sought to be exploited. In this example we exploit maximum parallelism and obtain a single directed data flow graph with 44 nodes (Figure 5.2), containing only primitive FPL functions.

We apply the graph reduction algorithm in order to effectively utilize the parallelism in the graph. After the combination of sequential nodes and reduction of parallelism, we obtain the final high level data flow (task) graph (Figure 5.3). The graph now consists of 24 nodes, with many nodes having been lumped together.

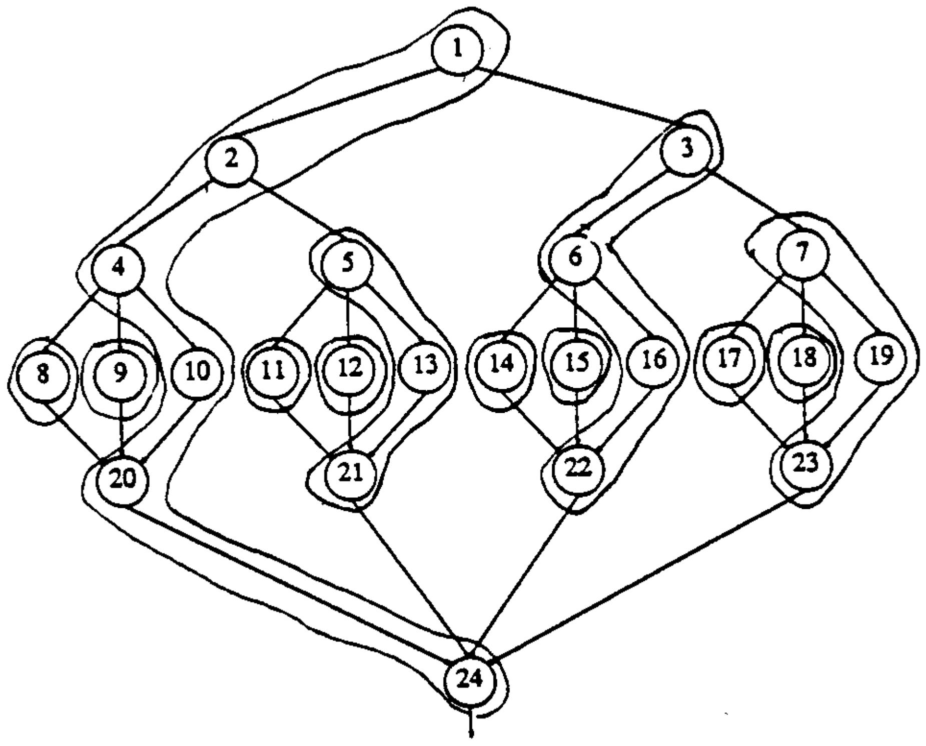


Figure 5.4 : Partitioned Data Flow Graph

Based on the longest path of the root node and the critical paths in the program, the data flow graph of Figure 5.3 is partitioned as shown in Figure 5.4. The modified Gantt chart showing partitions and representing the timing parameters associated with them is shown in Figure 5.5.

The allocation of the partitions of the matrix multiply program to two and four processors is shown in Figure 5.6. The response times are 67.9 and 42.7 time units respectively. They turn out to be the optimal response times because of the regular structure of the graph.

Another example whose performance we study in the next Section is the FFT. A FPL program for the 16-point FFT algorithm [LAHTI 81] is given below.

```
st1=CM(transform,align,[ID,table]);
st2=CM(transform,align_halves,[ID,table]);
st3=CM(transform,align_quarters,[ID,table]);
st4=CM(transform,align_eighths,[ID,table]);
align_all=CM(CT,APalign,TR);
align_halves=CM(align_all,APSP);
align_quarters=CM(align_all,APbreak,APSP);
align_eighths=CM(align_all,APbreak,APbreak,APSP);
break=CM(CT,APSP);
align=CM(DR,[CM(shuffle,data),CM(top,w_vector)]);
data=SL1;
top=SL1;
w_vector=SL2;
shuffle=CM(TR,SP);
table=K((1.0,0.0),(.71,-.71),(0.0,-1.0),(-.71,-.71),(-1.0,0.0),(-.71,.71),(0.0,1.0),(.71,.71));
transform=CM(unshuf,APbutterfly);
unshuf=CM(CT,TR);
butterfly=CM([c+,c-],[x,CM(c*,[y,W])]);
W=SL2;
x=CM(SL1,SL1);
y=CM(SL2,SL1);
c*=[CM(#-,AP#*,TR),CM(#+,AP#*,TR,[SL1,CM(RV,SL2)])];
c+=CM(AP#+,TR);
c-=CM(AP#-,TR);
fft=CM(st4,st3,st2,st1);
main =fft;
;
```

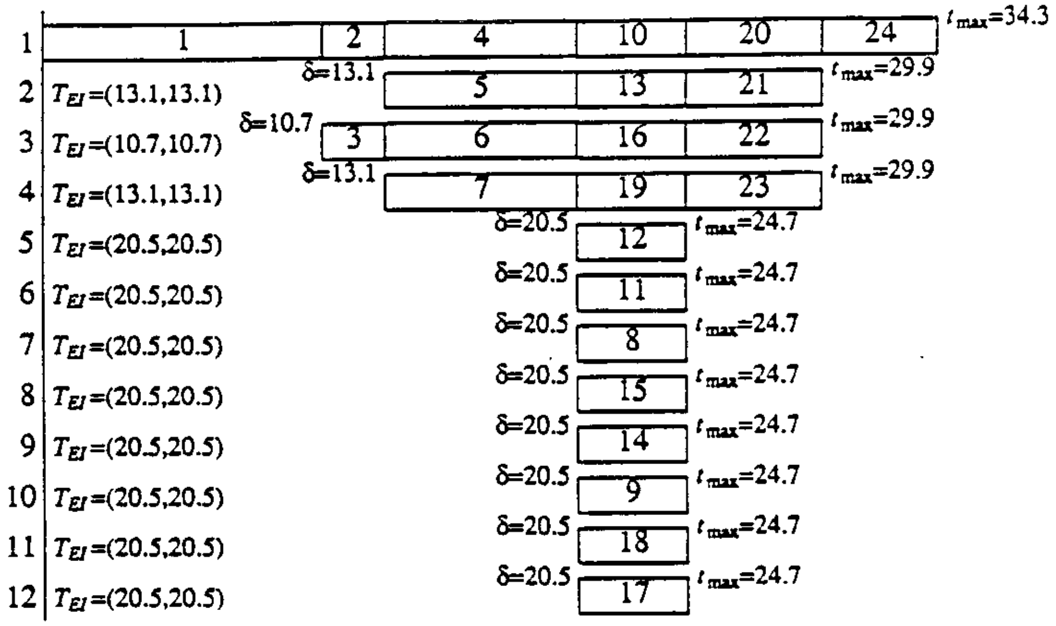


Figure 5.5 : Gantt Chart for Program Partitions

5.3 Analysis of Results

To study the performance of our system, two benchmark programs were chosen - Matrix Multiply and a 16 point Fast Fourier Transform (FFT). The matrix multiply program is canonical and permits us to obtain graphs of arbitrarily large size, by increasing the dimension of the matrices to be multiplied.

We first examine the speedup of the data flow system by plotting time (T) against the number of processors (N). Figure 5.7 illustrates the speed-up graph of the example of the multiplication of a (2X3) matrix by a (3X2) matrix from the previous section. As this example has a maximum of twelve "*" operations that can be

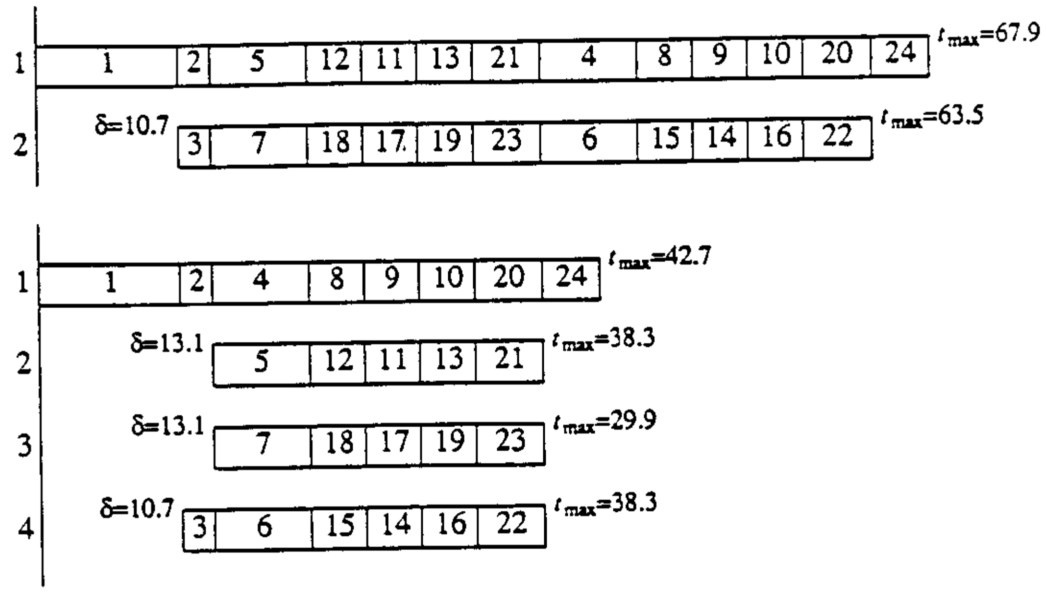


Figure 5.6 : Allocation for N=2 and N=4 Processors

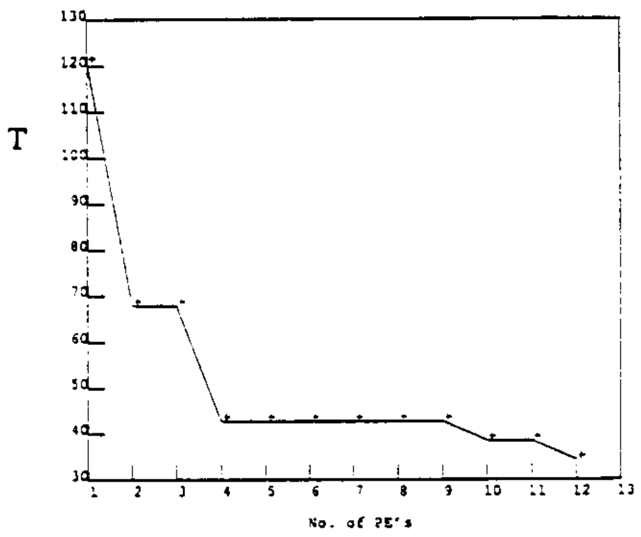


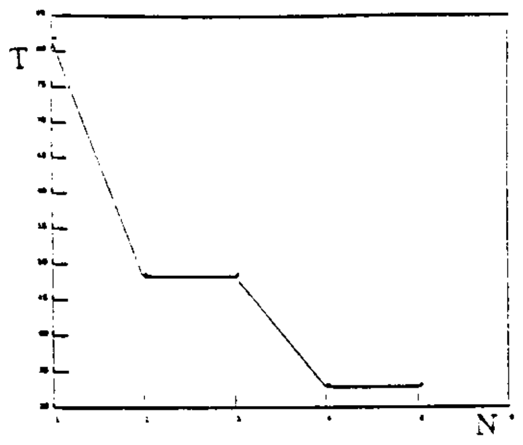
Figure 5.7 : Speedup Curve for 2X3 by 3X2 Matrix Multiply

executed in parallel, the smallest response time is achieved for twelve processors. Increasing the number of processors further, will not give any reduction in the response time.

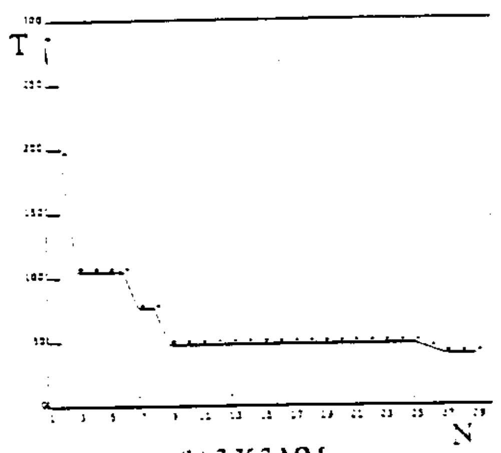
Figure 5.8a to 5.8e demonstrate the speedup of the multiplication of square matrices. In these examples, the degree of concurrency exploited in the partitioning step is maximum; i.e., when the *apply-to-all (AP)* is resolved, the graph produced before reduction has the maximum degree of parallelism possible, irrespective of the number of processors present in the system.

From Figure 5.8 we observe that the response time falls rapidly for the first few processors. This is the region where the amount of parallelism exceeds the number of processors. The speedup is almost linear and the processors are highly utilized. In this region the actual throughput is lower than the ideal throughput because of the constraints due to dependencies in the program. Subsequent flat regions in the speedup curve are due to constraints imposed by the program, and due to the degree and nature of concurrency in the algorithm. Matrix multiply provides a highly synchronous load and a cliff effect is observed when the number of processors is a factor of the total parallelism present. In the (5X5) matrix multiply of Figure 5.8d, the maximum degree of concurrency is 125 "*" operations. We observe flat regions beginning at N=5 and N=25 processors.

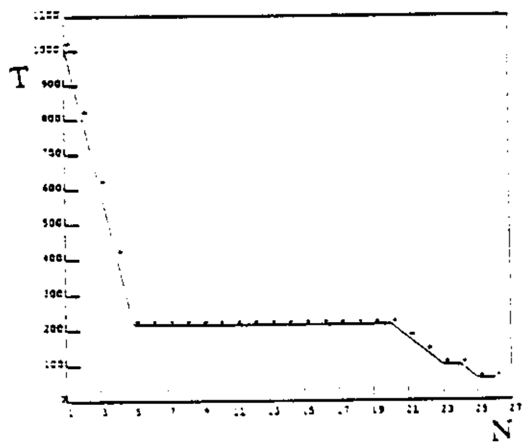
Ultimately, when the number of processors exceeds the degree of parallelism present, then the curve saturates. The (10X10) matrix multiply of Figure 5.8e stabilizes to the minimum response time at N=100.



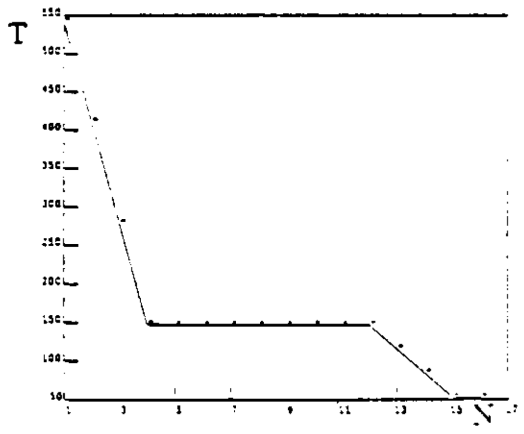
(a) 2 X 2 MM



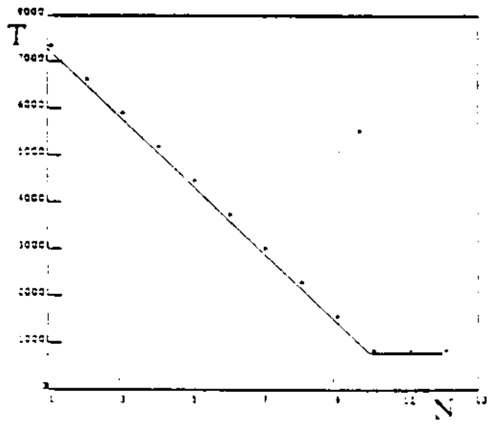
(b) 3 X 3 MM



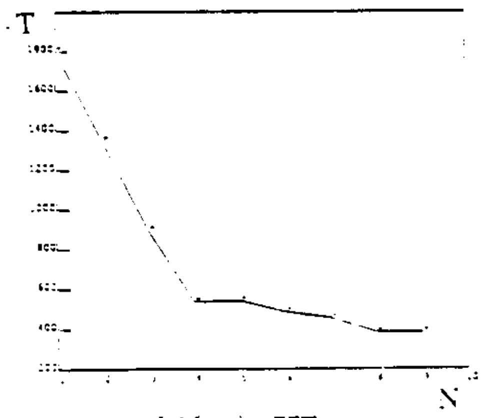
(c) 4 X 4 MM



(d) 5 X 5 MM



(e) 10 X 10 MM



(f) 16-point FFT

Figure 5.8 : Response Time Vs. No. of Processors for MM (a-e) & 16-point FFT (f)

A 16 point FFT algorithm requiring no bit reversal is described in [LAHTI 81]. In this example, the concurrency in the *apply-to-all (AP)* was exploited only to a limited extent, i.e. to the degree four against a maximum possible degree of eight.

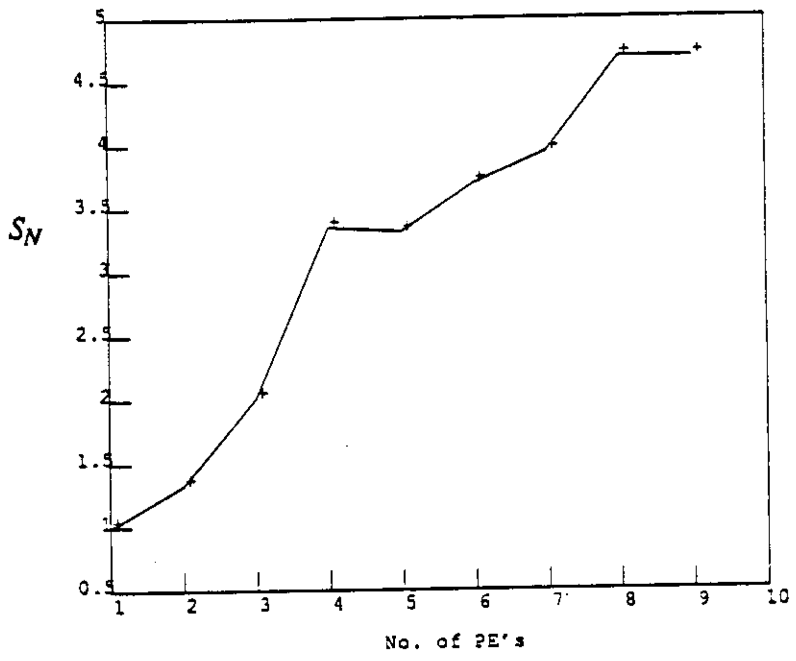


Figure 5.9 : Speedup (S_N) Vs. No. of Processors (N) for 16-point FFT

The speedup curve for the 16 point FFT is illustrated in Figure 5.8f. The speedup achieved for four processors is 3.35, with an efficiency of 0.83. The variation of speedup ($S_N[FFT]$) over a single processor with the number of processors (N) for the FFT program is shown in Figure 5.9. The efficiency of the utilization of the processors ($E_N[FFT]=S_N[FFT]/N$) is plotted against the number of processors (N) in Figure 5.10. The efficiency is limited by the dependencies in the

FFT algorithm and the amount of parallelism present.

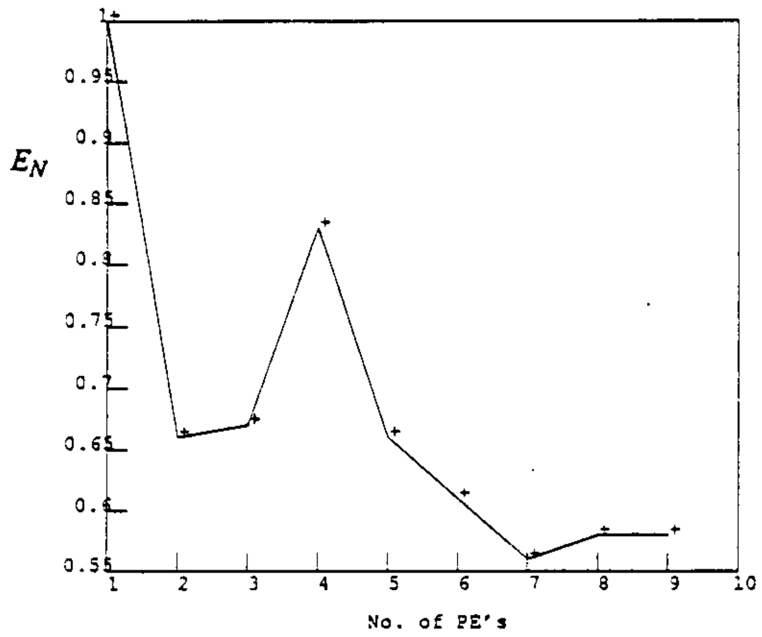


Figure 5.10 : Efficiency (E_N) Vs. No. of Processors (N) for 16-point FFT

Next, the execution time of the matrix multiply program was studied. For different dimensions of the (r X r) matrices, problems of the same structure but of different sizes are obtained. Figure 5.11 demonstrates the variation of execution time with the matrix dimension (r). The bottom curve represents the critical path (deadline) of the graph i.e., the minimum execution time. The critical path is almost constant, because by increasing the dimension of the matrices only the parallelism present increases. The other curves show the execution time complexity for 2, 4 and 16 processors.

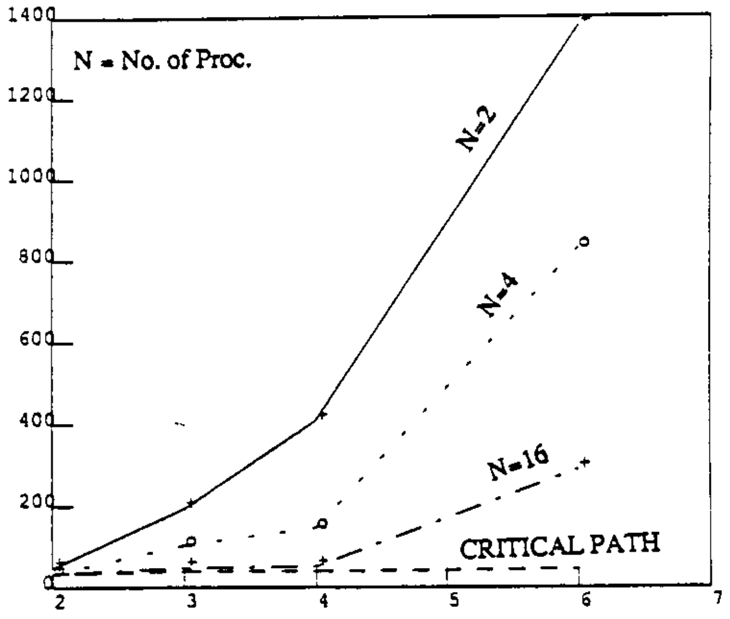


Figure 5.11 : Response Time (T) Vs. Matrix Dimension (r) for MM

Sequential execution of the multiplication of two (r X r) matrices takes r^3 multiplications. Sequential addition of r products takes O(r) time. Moreover the data has to be distributed to the right processor and communicated between the processors. From Figure 5.11, we observe that the curves are initially linear. This is the region where the number of processors is equal to or exceeds the amount of parallelism present. The execution time is O(r) in this region. Finally, when the amount of parallelism far exceeds the number of processors (N), then the curve is almost $O(r^3/N)$. Figure 5.12 demonstrates the variation of execution time with the number of nodes in the graph i.e. problem size.

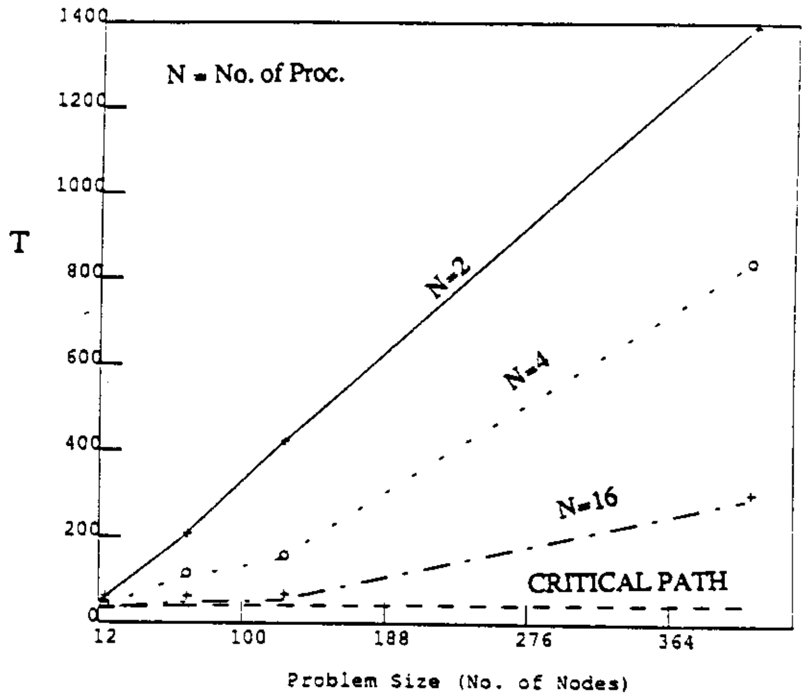


Figure 5.12 : Response Time (T) Vs. Problem Size (No. of Nodes) for MM

Finally we study the effect of partitioning on the response time. In particular we examine the effect of exploiting limited concurrency in the 16 point FFT program, when the number of processors available was limited. In Figure 5.13, we compare the response times obtained when functional forms are completely expanded and when they are expanded so that the degree of concurrency exploited from the functional forms is equal to the number of processors available. The upper curve (solid) shows the variation of execution time against the number of processors, when maximum concurrency is exploited. The dashed curve reflects the response time variation when only limited parallelism is exploited. The parallelism is limited by choosing the degree of concurrency (k) which can be exploited from a functional form to be less than its maximum value. By doing so functional forms are not expanded to the number of elements present in the vector to which the functional form is applied to, but to a smaller number equal to the number of processors available. From the curves of Figure 5.13 we conclude that when the number of processors is limited, exploiting much larger degrees of parallelism from the graph than the number of processors that exist, can result in overhead due to communication.

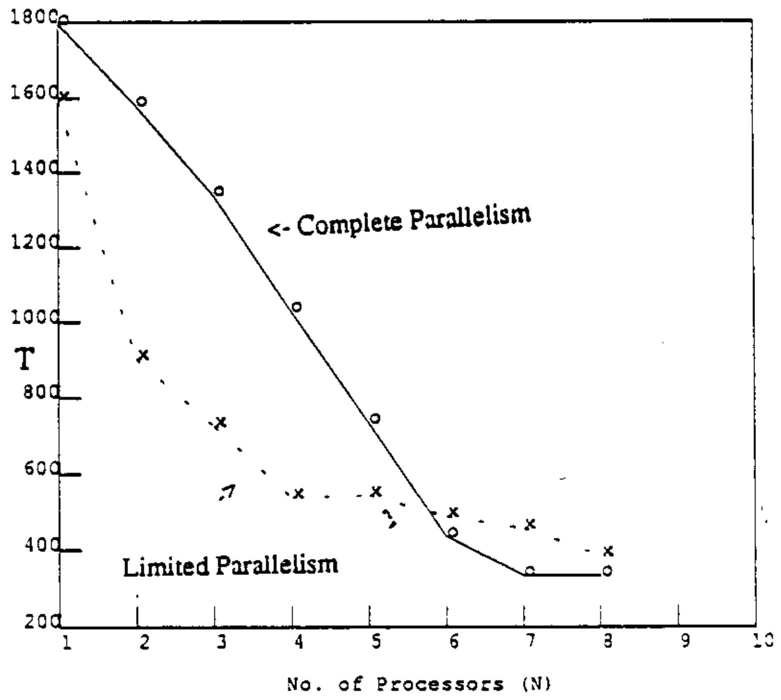


Figure 5.13 : Speedup Curves by Exploiting Complete and Limited Parallelism

CHAPTER VI

Conclusion

The design of multiprocessor systems has been beset with problems due to the extension of uniprocessor design principles to multiprocessors. Data Flow techniques provide an alternate strategy for the design of high performance multiprocessor systems.

We have shown that High Level Data Flow Architectures utilize the positive features of both conventional Data Flow Architectures and high performance uniprocessors. We demonstrate the use of Data Flow techniques at the optimum level of granularity instead of the fine grain, single operator level. A Graph Reduction algorithm to reduce a fine grain data flow graph to a task level graph, based on processing time and communication time criterion has been developed and applied (Section 3.5).

In Chapter III we demonstrate the ease with which parallelism inherent in an algorithm can be exploited by using Functional Programming Languages (FPL). The choice of FPL as a specification language for algorithms avoids the complexity, common to imperative languages like Fortran, of compiler techniques to detect parallelism. However the approach taken in this work is not restricted to FPL, but is applicable whenever a directed graph can be obtained from a program.

We show how the directed graph is extracted from the program and then reduced to a task graph of appropriate granularity. The task graphs are then placed in different partitions, which become the smallest unit for allocation. The partitions are used only for allocation and each task in the partition has to be activated separately during execution.

In Chapter IV we describe two algorithms for allocation; one to obtain minimum response time and to the other using fixed number of processors. An order of execution of tasks in a data flow processor is forced by introducing *pseudo-dependencies* between tasks. *Pseudo-dependencies* ensure that some tasks are executed before the others can begin, even though their data dependencies place no such restriction.

Finally we have shown the efficacy of allocation and scheduling for Data Flow Architectures. In Chapter V we observe the speed-up obtained by increasing the number of processors. We also note the constraints in the speed-up due to limited parallelism in the algorithm and data dependencies. An interesting result observed was the improvement in performance observed when only appropriate degrees of parallelism are extracted instead of complete parallelism.

To conclude we discuss some of the open problems and alternate strategies that can be adopted.

A different approach to the execution of Functional Languages is to use vector processors which can efficiently execute functional forms like *Apply-to-all (AP), Insert (IN) and Associative Insert (AI)*. If vector processors are available in the architecture, functional forms can be executed without having to unfold the functional form and distribute the data. In order to do so the user-written FPL program has to be algebraically transformed to make the vector evaluation of the *apply-to-all* more efficient. Transformations which convert the *apply-to-all* of functional forms to the *apply-to-all* of primitive functions will have the effect of distributing the *apply-to-all's* and hence not restrain parallelism. Such a scheme will benefit due to the systematic execution of vector operations and the reduction in overhead due to locality.

A problem to be solved is the incorporation of recursion and loops in our model. A suggested approach is to handle loops hierarchically and resolve the inner loops first.

Another approach to allocation and scheduling is to use List Schedules based on critical paths in the program. To force an order of execution in processors, each processor can maintain an ordered list of tasks resident in it.

Static allocations will not be very suitable for multiple-application time-shared systems. In such cases dynamic allocations and load balancing schemes have to be investigated.

References

[ACKE 82]	Ackerman, W. B., "Data Flow Languages," *Computer,* Vol. 15, No. 2, Feb 1982, pp. 15-25.
[ALLAN 80]	Allan S. J. and A. E. Oldehoeft, "A Flow Analysis Procedure for the Translation of High-Level Languages to a Data Flow Language," *IEEE Trans. on Computers,* Vol. C-29, No. 9, Sep. 1980, pp. 826-831.
[ARNO 82]	Arnold, C. N., "Performance Evaluation of Three Automatic Vectorizer Packages," *Proc. of Intl. Conference on Parallel Processing,* 1982, pp. 235-242.
[BABB 84]	Babb II, Robert. G., "Parallel Processing with Large Grain Data Flow Techniques," *Computer,* July 1984, pp. 55-61.
[BACKUS 78]	Backus, J. , "Can Programming be liberated from the von Neumann Style ? A Functional Style and its Algebra of Programs," *CACM,* Vol. 21, No. 8, Aug. 1978, pp. 613-641.
[BADE 83]	Baden, S. and D. R. Patel, "Berkeley FP - Experiences with a Functional Programming Language," *Proceedings COMPCON,* Spring 1983, pp. 274-277.
[BERN 66]	Bernstein, A. J., "Analysis of Programs for Parallel Processing," *IEEE Trans. on EC,* Oct. 1966, pp. 757-763.
[CHAN 84]	Chan, P. K., "A Dataflow Multimicroprocessor Architecture," M.S. Thesis, UCLA Computer Science Department, Report No. CSD-840044, Nov. 1984.
[CHOU 82]	Chou, T. C. K., and J. A. Abraham, "Load Balancing in Distributed Systems," *IEEE Trans. on Software Engineering,* July 1982, pp. 401-412.
[DENN 80]	Dennis, J. B., "Data Flow Supercomputer Languages," *Computer,* Nov. 1980, pp. 48-56.

[EL-DESS 81] El-Dessouki, O., W. Huen and M. Evans, "Towards a Partitioning Compiler for a Distributed Computing System," *Computer Science Press*, 1981.

[ENSL 77] Enslow, P. H., "Multiprocessor Organization - A Survey," *Computing Surveys*, Vol.9, No. 1, March 1977, pp. 103-129.

[ERCE 83] Ercegovac M. D. and S. L. Lu, "A Functional Language Approach In High Speed Simulation," *Summer Computer Simulation Conference*, 1983, pp. 383-387.

[ERCE 84] Ercegovac M. D. , P. K. Chan and T. M. Ravi, "A Dataflow Multiprocessor Architecture for High-Speed Simulation of Continuous Systems," *Proc. International Workshop on High-Level Architecture*, 1984.

[ERCE 84 a] Ercegovac M. D. et al., "Task Partitioning, Allocation and Simulation for a Dataflow Multiprocessor System," *Proc. Summer Computer Simulation Conference*, 1984.

[ERCE 86] Ercegovac M. D. and Tomas Lang, "General Approaches for Achieving High Speed Computations," *to appear in Supercomputers*, Ed. S. Fernbach, North Holland, 1986.

[FELL 81] Feller, M., *CS259 Seminar Report*, UCLA Computer Science Department, 1981.

[GAJS 82] Gajski, D. D., D. A. Padua and D. J. Kuck, "A Second Opinion on Data Flow Machine and Languages," *Computer*, Vol. 15, No. 2, Feb 1982, pp. 58-69.

[GAREY 79] Garey M.R. and D. S. Johnson, *Computers and Intractability - A Guide to the theory of NP-Completeness*, W. H. Freeman and Company, San Francisco, 1979.

[GAUD 82] Gaudiot, J. L., "On Program Decomposition and Partitioning in Data-Flow Systems," Ph.D. Thesis, UCLA Computer Science Department, Report No. CSD-821212, Dec. 1982.

[GAUD 84] Gaudiot, J. L., and M. D. Ercegovac, "Performance Analysis of a Data-Flow Computer with Variable Resolution Actors," *Proc. 4th Intl. Conf. Distributed Computing Systems*, 1984, pp. 2.9-2.17.

[GEHR 82] Gehrig, E. et al., "The CM* Testbed," *IEEE Computer*, Oct. 1982, pp. 40 -53.

[HAES 80] Haessig, K., and C. J. Jenny, "An Algorithm for Allocating Objects in Distributed Computing Systems," *IBM Research Report*, RZ 1016 (#36244), June 1982.

[HWAN 81] Hwang, K., S. P. Lu and L. M. Ni, "Vector Computer Architecture and Processing Techniques," *Advances in Computers*, Vol. 20, Academic Press, 1981, pp. 115-197.

[IRANI 82] Irani K. B. and K. W. Chen, "Minimization of Interprocessor Communication for Parallel Computation," *IEEE Trans. on Computers*, Nov. 1982, pp. 1067-1075.

[ISLAM 81] Islam, N. , T. J. Myers and P. Broome, "A Simple Optimizer for FP-like Languages," *Proc. of the 1981 Conf. on Functional Programming Languages and Computer Architecture*, Oct. 1981, Portsmart, New Hampshire.

[KIEB 81] Kieburtz, R. B., "Transformations of FP program Schemes," *Proc. of the 1981 Conf. on Functional Programming Languages and Computer Architecture*, Oct. 1981, Portsmart, New Hampshire.

[KOGG 81] Kogge, P. M., *The Architecture of Pipelined Computers*, McGraw-Hill, New York 1981.

[KUCK 81] Kuck, D. J. et al., "Dependence Graphs and Compiler Organizations," *8th Annual ACM Symposium on the Principles of Programming Languages*, Williamsburg, VA, Jan. 1981.

[LAHTI 81] Lahti, D. O., "Applications of a Functional Programming Language," M.S. Thesis, UCLA Computer Science Department, Report No. CSD-810403, Apr. 1981.

[LANG 77] Lang, T. and E. B. Fernandez, "Improving the Computation of Lower Bounds for Optimal Schedules," *IBM Journal of Research and Development*, Vol. 21, No. 3, May. 1977, pp. 273-280.

[LU 84] Lu, S. L., "A Compiler for a Functional Programming System," M.S. Thesis, UCLA Computer Science Department, Report No. CSD-840045, Nov. 1984.

[MAR 79] Mariani M. P. and D. F. Palmer, "Tutorial : Distributed System Design," *IEEE Computer Society*, 1979, pp. 221-223.

[RAMA 72] Ramamoorthy C. V. , K. M. Chandy and M. J. Gonzalez, "Optimal Scheduling Strategies in a Multiprocessor System," *IEEE Trans. on Computers*, Feb. 1972, pp. 137-146.

[RAVI 83] Ravi T. M., "Graph Reduction," *NASA Data Flow Multiprocessor Project,* Status Report #3, Apr. 1983.

[STON 80] Stone H. S. "Parallel Computers," *Introduction to Computer Architecture,* SRA, Chicago, Ill., 1975.

[WADL 81] Wadler, P., "Applicative Style Programming, Program Transformation, and List Operators," *Proc. of the 1981 Conf. on Functional Programming Languages and Computer Architecture,* Oct. 1981, Portsmart, New Hampshire.