

# **AUTOMATIC PARALLELIZATION OF SEQUENTIAL PROGRAMS**

*Dissertation submitted to Jawaharlal Nehru University  
in partial fulfilment of the requirements  
for the award of the degree of*  
**MASTER OF TECHNOLOGY**

IN

**COMPUTER SCIENCE & TECHNOLOGY**

By

**RAJESH RAI**

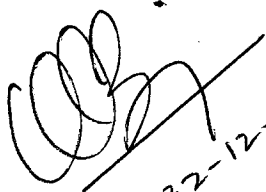
**SCHOOL OF COMPUTER AND SYSTEMS SCIENCES  
JAWAHARLAL NEHRU UNIVERSITY  
NEW DELHI-110 067  
DECEMBER, 1993**

## CERTIFICATE


This is to certify that the dissertation titled "Automatic Parallelization of Sequential Programs" being submitted by me to Jawaharlal Nehru University, New Delhi in partial fulfilment of the requirements for the award of the degree of **Master of Technology**, is a record of the original work done by me under the supervision of **Dr. C. P. Katti** Associate Professor, School of Computer and System Sciences, Jawaharlal Nehru University, New Delhi during the year 1993, Monsoon Semester.

The results reported in this dissertation have not been submitted in part or in full to any other university or institution for the award of any degree or diploma.

  
RAJESH RAI

  
22-12-93

Prof. K. K. Bharadwaj  
Dean  
School of Computer and  
System Sciences,  
Jawaharlal Nehru University  
New Delhi

  
Dr. C. P. Katti  
Associate Professor  
School of Computer and  
System Sciences  
Jawaharlal Nehru University  
New Delhi

## ACKNOWLEDGEMENTS

This dissertation would not have happened without the help of many people. Of all these, I am particularly indebted to **Dr.C.P.Katti**, Associate Professor, School of Computer and System Sciences, Jawaharlal Nehru University, whose gracious and valuable guidance helped bring my endeavours to fruition. I thank him for that and much else.

My warmest thanks to **Prof.K.K Bharadwaj**, Dean, SC&SS, J.N.U., who provided me with the excellent academic environment and facilities of the school, which enabled the successful completion of my project.

My greatest thanks are due to many of my fellowmates, specially **Mr. Sumitra Kumar Srivastava**, and **Mr.Manoj Kr. Sarangi**, who listened to my ideosyncrosies with attention, gave their company in the long hours of night and advised me in the times of disappointment. Many others who are too many to acknowledge individually, gave moral and academic support in commpleting this dissertation on deadline.

This dissertation ows a lot of thanks to **Mr. Rajendra** who helped in the personal computers' lab.

My affectionate thanks to my parents and family members who were always a great motivating force to me.

DATE: Dec 14, 1993

RAJESH RAI

# CONTENTS

## CHAPTER 1

*INTRODUCTION* 1 - 5

## CHAPTER 2

*PARALLEL ALGORITHMS* 7 - 15

2.1 Characteristics of Parallel Algorithms 7

2.2 Performance Measures 12

## CHAPTER 3

*DEPENDENCE ANALYSIS* 16 - 27

3.1 Types of Dependencies 16

3.2 Detecting Dependencies 21

3.2 Removing Dependencies 22

## CHAPTER 4

*PARALLELIZATION OF SEQUENTIAL PROGRAMS* 28 - 73

4.1 Block of Assignment Statements 28

4.2 Arithmetic Expressions 30

4.2.1 Tree Height Reduction 33

4.2.2 Distribution Algorithm 38

4.2.3 Recurrence Relations 43

4.3 Loops 48

4.3.1 Simple Loops 50

4.3.2 Nested Loops 53

4.4 IF Statements 62

4.5 Towards Automatic Parallelization 63

## CHAPTER 5

*CONCLUSION* 74

*REFERENCECES* 77

# CHAPTER ONE

# INTRODUCTION

There has been tremendous growth in interest in parallel architecture and parallel processing in recent years, resulting in dozens of new machine designs, prototypes and programming languages for parallel and distributed computing. This basic research in architecture has led to over a dozen commercially available parallel systems. However comparatively little research has attacked the problem of how to characterize the parallelism in programs and algorithms. To improve the performance of the computers it is necessary that the parallelism in the programs is explored and exploited.

Developing more powerful computers can be done either by increasing the hardware speed or by searching for new computing techniques. In modern supercomputing design both ways have been followed. Most supercomputers are grownup derivatives of the well-known von Neumann processor architecture, widely in use today. The speedup of this basic architecture is based on improving the memory organization, enhancing the processor speed and applying parallel processing techniques.

The communication path between the processor and the memory for fetching instructions and reading or writing data limits the amount of work that can be done between two memory accesses. An instruction normally involves the following phases:

IF : Instruction Fetch	---	Memory Action
ID : Instruction Decode	---	Processor Action
OF : Operands Fetch	---	Memory Action
EX : Execute	---	Processor Action

Clearly, the speed of execution depends to a large extent on the memory bandwidth. A first way to enhance the processing power is to increase the memory accessibility through a wider data bus. Whereas in conventional computers the databus is 16 to 32 bits wide, supercomputers use a 64, 128 or even 512 bits wide datapath.

In conjunction with the wider bus, a faster memory is used. Since most supercomputers obtain their maximum speed on processing large arrays of data, memory speed is of prime importance. The slow processor\_memory interconnection is the bottleneck of the von Neumann architecture. In order to circumvent this bottleneck, which is largely due to the technological limitations of the memory, a technique known as *interleaving* allows to create a virtual access time of a few nanoseconds, when the proper conditions are met. The memory is organized in  $n$  banks in which the data at location with address  $i$  is stored in bank  $i \bmod n$ . This organization allows to access the consecutive array elements in shorter time.

In a balanced system, the processor speed has to match the speed of the memory. Increasing the processing speed starts by using a faster technology e.g. by using ECL, higher integration, greater clock rate, smaller dimensions in order

to obtain a lower propagation time, and the like. Most of these measures have been taken in advance machines, thereby paying a high price for approaching the borders of today's physical possibilities.

The only way to increase the processing power beyond the physical and economical limits of conventional architecture, is to operate different processors in parallel on the same program. When the experimental dataflow architecture is excluded i.e. if only program driven instead of data driven execution is considered, then the well known Flynn \_ topology is helpful in classifying the existing supercomputer architectures. Flynn discerns single (S) or multiple (M) instruction (I) and data (D) streams, and forms four theoretical architectures, of which the closest real architectures are :

SISD : the conventional von Neumann architecture.

SIMD : the array processors.

MISD : the pipelined processors.

MIMD : the shared memory multiprocessors.

In array processors several independent identical arithmetic units (ALU) or processing elements (PE) operate in parallel under supervision of the control unit (CU).

Conversely, in pipeline computers, a single datum is operated upon by the different stages of one functional unit (FU). Parallelism is achieved by the simultaneous execution of the different stages on a stream of sequential data. In the two preceding techniques, the execution time of an ele-



mentary operation of a processing element or a pipeline stage is one time unit. This facilitates the synchronization between processing elements, since all PE's operate in lockstep under control of the same clock.

Multiprocessors on the other hand allow different processors to execute parts of the program asynchronously. While this allows greater flexibility, such an architecture also requires a fast synchronization system. This is probably the main reason why today's supercomputers don't use multiprocessing techniques in general.

It is clear that there is no single answer to the need for faster processors. There are a number of techniques, and the computer architects have realized machines with a great peak performance only from the blend of the following techniques :

- Fast processor and memory technology.
- Wide high speed data bus.
- Multiple pipelining
- Lockstep operation
- Array processing.

Supercomputers are not the product of a nicely developed breakthrough in computer architecture, but incorporates a balanced mix of all known technologies to speed up the processing power, whose selection ultimately is based on economical tradeoffs.

There are two approaches in implementing parallel processing. The first and obvious approach is to develop completely new programs based on new algorithms that are suitable for parallel processing. This approach is guaranteed to result in very good programs in terms of both execution speed and efficiency. However it is generally difficult to write a program for parallel computer. The reasons are :

1. *Difficulty in exploiting parallelism.* It is not an easy task for a programmer to divide a program into a set of tasks that are executable in parallel.

2. *Difficulty in utilizing processors effectively.* It is a difficult task to perfectly balance the load equally in all the processors available so that no processor sit idle.

Another approach is to let the compiler expose and exploit parallelism from an existing or ordinary program. Compilers for a vector computer have been developed and are being used extensively. These compilers are not powerful enough to apply to a parallel processing computer, however. In vector computers, parallel computation is done only in terms of vector data. Scalar operations cannot be computed in parallel even if they are independent of each other. Similarly independent tasks cannot be computed in parallel. A parallel processing computer on the other hand, has more freedom. Any independent operations may be computed simultaneously. Hence, although, it is sufficient for a vector computer compiler to check only whether operations on ele-

ments of a vector can be performed simultaneously, a parallel processing computer compiler must do more. It must extract and expose more parallelism.

In this project, an attempt has been made to discover operations that may be performed simultaneously by examining programs at the statement level and the ways to execute them in parallel have been explored.

## **CHAPTER TWO**

# PARALLEL ALGORITHMS

## 2.1 CHARACTERISTICS OF PARALLEL ALGORITHMS

Parallel algorithms can be characterized in many ways, along a number of dimensions. Similarly, parallel architectures can be described in terms of a number of attributes. Ideally, a set of orthogonal characteristics would describe parallel algorithm and a corresponding set of orthogonal characteristics would describe parallel architectures, with a unique bijection performing the mapping from one to the other. Experience shows that the relationship between parallel algorithm and parallel architecture is clearly too complex to conform to such a desirable model. In the absence of independence, completeness therefore becomes relevant goal.

The main characteristics of the parallel algorithms are:

### 1. Nature of parallelism :

#### *Data parallelism versus function parallelism:*

Parallelism can be achieved by dividing the data among the processors, by decomposing the algorithm into segments that can be assigned to different processors, or by pipelining. The type of parallelism will affect allocation of data, the assignment of processes to processors, and basic decision as to what mode of parallelism (SIMD / MIMD / pipeline) to use. Function parallelism will almost always imply MIMD

operation. Data parallelism will often be amenable to SIMD implementation; however, data parallelism alone is not sufficient to guarantee good performance in SIMD mode. At some level of decomposition, an algorithm must exhibit both data and function parallelism in order for pipelined operation to be applicable. Memory organisation can be tied to the type of parallelism in that data parallelism is often well implemented using local memories. The type of parallelism will also act as a broad indicator of the number of processors. With data parallelism, the number of processors will typically be proportional to the data set size, and utilization of large numbers of processors will not be uncommon. Algorithms based on function parallelism will more typically use number of processors counted in tens rather than the thousands.

*Data granularity:* Data granularity deals with the size of the data items processed as a fundamental unit, and will have a bearing on the data allocation, communications requirements, processor capability, and memory requirements. Fine\_grain algorithms will often be suitable for SIMD or pipelined operations using local memories. The data granularity generally will not affect the overall memory requirements, but may bear on the amount of the memory that must be readily accessible to each processor. The data granularity will provide an indication of the bandwidth needed to communicate a single data item.

*Module granularity:* Module granularity quantifies the amount of processing that can be done independently, either

of other processes or of operations being performed in other processors. It is essentially a measure of the frequency of synchronisation, and will affect the choice of SIMD versus MIMD operation, the assignment of the processes to processors, the memory organisation, the communication requirements, and the likelihood of equalizing the execution times of component parts of the algorithm. Algorithms characterized by fine\_grain module granularity will require frequent synchronisation. If possible, SIMD or pipelined execution, in which communication can be performed with less overhead than in MIMD operation, will be preferred, as will a local memory organization. Because of the frequent communications, a fast network capability is imperative. Large\_grain algorithms typically have less of a need for efficient communications. The large amount of processing done between synchronization points often suggests MIMD execution.

2. Degree of parallelism: This will be related to both the data granularity and the module granularity. Its most direct impact will be on the choice of machine size and on the maximum speed attainable. In addition, degree of parallelism will in practice often be related to the mode of operation and the memory organization (with massive parallelism, global memory organization can lead to significant contention in accessing memory).

3. Uniformity of the operations: If the operations to be performed are uniform (e.g., across the data or feature set), then SIMD or pipeline processing is feasible. Uniformity will generally be associated with data

parallelism. If the operation are not uniform, then MIMD processing will be chosen and strategies to equalize the computational load across the processors may come into play. These strategies may be applied statically at compile time or dynamically at execution time.

4. Synchronization requirements: In addition to the synchronization requirements implied by the module granularity, consideration of precedence constraints is implicit in characterizing the synchronization requirements. This will affect the assignment of processes to processors and the scheduling of various components of the algorithm.

5. Static/ dynamic character of the algorithm: The pattern of process generation and termination will affect the processor utilization, the scheduling of sub processes, the mode of processing, the memory organization and the communication requirements. Identification of an algorithm as being dynamic generally rules out SIMD or pipelined execution. A dynamic algorithm will need to be supported by either a global memory organization or a communication network and an I/O system capable of providing a fast means of loading the local memories with the data and code needed for a few process.

6. Fundamental operations: The basic operation performed in the algorithm will dictate the processor capabilities needed. To the extent that the operation identified as being the basic unit of processing also determines some communications requirements, this characteristic will also have a bearing on the network and memory organization.



7. Data type and precision: The atomic data types and data precision will bear most directly on the individual processor capability and on the memory requirements, but may also imply requirements for communications bandwidth.

8. Data structures: Many algorithms can be characterized as having a natural data structure on which operations are performed. The ability of an architecture to support the needed access patterns, to exploit possible regularity in the structures, and to allow the needed interactions between parts of the structures will affect algorithm performance. In using the algorithm characteristics, it is necessary to make a distinction between an attribute that is required for a particular architecture implementation and one that is preferred. For example, uniformity is a requirement for SIMD processing; although it is possible to construct an SIMD implementation of a non\_uniform algorithm, performance will be so bad that it is not reasonable implementation to consider. In contrast, if an algorithm has a high degree of uniformity, then, depending on its other attributes, it may be that an SIMD implementation is preferable, but an MIMD implementation may run only slightly more slowly.

## 2.2 PERFORMANCE MEASURES

The performance of any system can be measured by knowing the efficiency, speedup and execution time. In the case of parallel processing, we will define these terms here. Let a group of processes collaborate in an algorithm. Assume that the processes are residing in separate machines so that they all may execute at the same time. In this case, if a process is idle, it is assumed that the processing power of that machine is wasted.

One more assumption is made, that is, processes do not share memory; instead, they communicate by sending each other messages. It is assumed that the messages are relatively expensive, requiring times on the order of tens of milliseconds for delivery.

Execution time  $T(p,n,A)$  : It is the time needed by algorithm A to compute a problem of size n on p processors. Execution time includes initialization and communication time, and is measured from the time the first process starts to the time the last one terminates.

Speedup : Speedup of a parallel machine gives the comparison of the time required by that algorithm to the time needed in doing the same work by the best known serial algorithm. It is defined as:

$$S(p,n,A) = \frac{\text{Time required by the best serial algorithm}}{\text{Execution time } T(p,n,A)}$$

Values of speedup  $S$  range from zero to infinity. By definition, the value of  $S(1)$  will be in between 0 and 1. If a comparison is made of the distributed algorithm against  $T(1,n,A)$ , we get a value which is called the *rough speedup*,  $RS$ . It is a less honest measure of performance than the true speedup. If  $RS(p) > p$ , we say that the algorithm exhibits a *speedup anomaly*. It shows that the distributed algorithm is poor when  $p = 1$ . If the distributed algorithm is the same as the serial algorithm for  $p = 1$ , a speedup anomaly implies that the serial algorithm is suboptimal.

Efficiency : In case of parallel algorithms we get the speedup in comparison to the serial counterpart at the cost of more than one processors. The speedup  $S$  won't be able to tell whether the processors are working at their full capability or not. To measure this, we define the efficiency as the speedup obtained per processor.

$$\text{Efficiency } E(p,n,A) = \frac{S(p,n,A)}{p}$$

Values of  $E$  range from 0 to 1. The rough efficiency is defined analogously to the rough speedup.

The useful - process point  $U(p,A)$  : It is the size  $n$  of problem that makes it worthwhile to use as many as  $p$  processors.

$$U(p,A) = \text{smallest } n \text{ such that } T(p,n,A) \leq T(p-1,n,A)$$

In general as the problem size increases, the expenses of distribution (which may be dependent on  $p$ ) begin to be outweighed by its benefits.

Cost factor : This also is a major criterion in evaluating any system. The cost of a computer system to a user is the money that the user pays for the system, namely its price. To the designer the cost is the cost of manufacturing including the cost of the development and capital tools for construction. With the developments in technologies, the hardware cost of the system is decreasing sharply whereas the cost of software is steadily rising with inflation and complexity and with apparently little relief from advances in software tools.

Software and hardware costs each have two components, a one time development cost and a per unit manufacturing cost. Because of the production of hardware in bulk, the per unit manufacturing cost is very less. But the one time development cost for hardware is much more than the software cost. As the user is to pay the per unit price, for him the hardware is very cheap.

We can evaluate the architectures by there cost and performance. The effectiveness of an architecture must be measured on workloads for which the architecture is intended. An architecture that is inefficient because of wasted resources will compete poorly against the simpler but mor efficient architecture.

There are a dozen more criteria for measuring the performance, such as maximum program and data size, weight,

power consumption, volume and ease of programming, that may have relatively high significance in particular cases. But, in general the performance will be measured using the criteria discussed above.

## **CHAPTER THREE**

# DEPENDENCE ANALYSIS

The first and foremost thing to be taken into consideration whenever a need to execute serially written program onto the parallel machines arises, is to check for the dependence relations among the various statements in the program. Any two statements may be executed in parallel (or concurrently) if we get the same result in executing them in any order. This is possible only when one is not dependent on the other.

## 3.1 Types of Dependencies

The statements may be dependent on one another in one of the following ways :

1. *Data dependent.* The second statement may be requiring a few or all the values which are the resultant of the execution of the first statement i.e. the output variable of the first is the input variable of the second. We say that the second statement is data dependent on the first. This is called WRITE/ READ dependency as the values of the variables being modified (or written) after the execution of the first statement are read by the second.

*Examples.*

```
A = B + C          ----- S
D = A + E          ----- T
```

The above two statements can not be executed in parallel because the variable 'A' to be used in statement 'T' is being written (modified) by the statement 'S'. 'T' is thus

data dependent on 'S'. The statement 'T' must be executed after the execution of 'S'.

```
For i := 1 to N do
    A[i] := A[i-1] / B[i];
end; {* of the loop *}
```

In this example, in place of two separate statements, we have one statement which is to be executed 'N' times. It can be seen that the two iterations are not independent because a variable written in previous iteration is read in each iteration.

2. *Antidependent.* It may be the case that the first statement is using some variables which are to be modified by the execution of the second statement. In this case also, we cannot execute them simultaneously. This is the case just opposite to the first one and we say that the second statement is antidependent on the first statement. This is also called READ / WRITE dependency as a few variables are to be read by the first statement before those are written by the other.

*Examples.*

D = A + E            ---- P

A = B + C            ---- Q

Here the first statement 'P' uses a variable 'A' which is to be modified after the execution of the second statement 'Q'. Hence the two are not independent and cannot be executed in parallel.



```
For i := 1 to N do
    A[i] := A[C[i]];
end;
```

In this case, if the value of  $C[i]$  is less than  $i$  then there will be READ / WRITE dependency among the statements of different iterations and if the value of  $C[i]$  is greater than the value of  $i$  then WRITE / READ dependency is observed. Moreover, if the values of  $C[i]$  are computed during execution, the compiler cannot determine which dependency exists and therefore cannot optimize the code. Therefore, the compiler can detect loop to loop dependencies only when all subscript expressions in an iteration and the loop increment have values known to the compiler. Optimizing compilers are forced to assume that the dependencies are present if index variables depend on execution\_time program behavior. Otherwise, the optimizing process is likely to produce a translated program that runs incorrectly.

3. *Output dependent.* If the two statements are writing in the same memory location allotted for a fixed variable then they won't be independent and hence cannot be executed simultaneously. Any memory location cannot be written into by more than one processors at the same time and in that case any value would be stored, giving the incorrect result. This type of dependency is also called WRITE / WRITE dependency as both the statements are writing in the same variable.

*Examples.*

```
A = B + C          ---- U
A = C + E          ---- V
```

The above two statements are having the same output variable 'A' and thus are output dependent to each other. They cannot be executed in parallel.

```
For i := 1 to N do
  A[i] := B[i] * 5;
  A[i-1] := B[i] + C[i];
end;
```

Here exists WRITE / WRITE dependency as the second statement in current iteration is modifying the same variable as the first statement in the previous iteration leading to the dependency for variable A[i-1]. If the loop index is increased by 2 instead of 1, then there is no dependence caused by writing two successive values into 'A'.

The data dependencies in an algorithm will play the largest role in dictating data allocation patterns and communications characteristics. They will also have a major part in the decision to use a global versus local memory organization. In the characteristics based approach, the handling of data dependencies is conceptually a graph isomorphism problem. The library contains known data dependency structures and (potentially multiple different) mappings of these structures onto architecture configurations. The purpose of

the stored information is to make available mappings which experience has shown to be useful but which might be difficult or prohibitively time consuming to derive directly. Although this approach uses the intermediary of the stored database of patterns, the major steps involved in mapping the data dependency graph for the current algorithm onto one of the stored structures are similar to those employed for mapping an algorithm directly onto an architecture.

Although the identification of the communications pattern can be formulated conceptually as a graph isomorphism problem, the complexity of graph isomorphism will in most cases make the direct and exhaustive test for isomorphism infeasible. So the organisation and representation of the stored structures is done such that the searching and the matching process can be performed efficiently. The search process may be assisted by the inclusion of auxiliary information with each of the stored patterns. Characterizing the data dependencies lies in the heart of the algorithm to architecture mapping problem, and there are many approaches to be explored.

### 3.2 Detecting Dependencies

A general procedure for detecting dependencies is to list the names of the variables read and written in a loop iteration. If a name appears on both lists, it potentially leads to a READ / WRITE or WRITE / READ dependence. All variables that are written are potentially WRITE / WRITE

dependences. The compiler has to examine each case further to determine if an actual dependence exists.

Let the set of input and output variables of a statement X be defined by  $\text{in}(X)$  and  $\text{out}(X)$  respectively. Then the two statements S and T will be said to be independent if and only if :

$$\text{in}(S) \text{ int. out}(T) = \text{phi} \quad \text{--- (1)}$$

$$\text{in}(T) \text{ int. out}(S) = \text{phi} \quad \text{--- (2)}$$

$$\text{out}(S) \text{ int. out}(T) = \text{phi} \quad \text{--- (3)}$$

where 'int.' is the intersection between two sets and 'phi' denotes the empty set.

The first condition ensures that none of the variables which are being modified by statement T is required for the execution of the statement S. The second condition checks whether the variables modified by statement S are used by statement T or not. The third condition is necessary for checking the output dependence, to assure that both statements are not writing into the same variable.

If the above three conditions are satisfied, the two statements can be executed in parallel.

Similarly, for n statements  $T_i$ ,  $i = 1, 2, \dots, n$  to be executed in parallel the following conditions must be satisfied

$$\text{in}(T_i) \text{ int. out}(T_j) = \text{phi} \quad \text{for all } i \langle \rangle j$$

$$\text{out}(T_i) \text{ int. out}(T_j) = \text{phi} \quad \text{for all } i \langle \rangle j$$



We will be able to execute a certain set of statements in parallel if the above dependence relation holds good. However, in general, one or other type of dependence will always be there. Some of them can be removed by using different methods as the case may be. In that case the program will be able to run in parallel. But we can not remove all the dependencies each time. Where ever it is not possible, the statements can not be executed simultaneously and that much part of the program is bound to be serial.

### 3.3 Removing Dependencies

Artificial data dependencies impose a constraint on the execution sequence of statements and instructions, which is not required by the algorithm and therefore limit the useful parallelism in the programs. These dependencies can be removed by the following methods :

1. *Renaming of variables* : A variable is the symbolic reference to a memory location. Each reassignment of the variable therefore signifies a rewrite of that memory location. Consequently, the instructions reading a variable have to be sure to read the proper version of that variable. This is implicitly guaranteed by the programmer in a sequential execution, but creates unnecessary and undetectable constraints for a parallel execution. For example in the following program :

```
DO 10 I = 1, 100
    A(I) = I
    B(I) = -I
```

```

10  CONTINUE
    DO 20 I = 1, 20
        X(I) = (Y(I - 1) + Y(I + 1)) / 2
    20  CONTINUE

```

Although visibly unrelated, the two loops require sequential execution, since they both use the loop variable I, which is stored in a unique memory location.

Another example of the same type is :

```

READ (5, 50) N, A
CALL GAUSS (A, X, N)
.....
.....                                old N, X
-----
READ (5, 50) N, X                    new N, X
CALL ORDER (X, N)

```

The use of same memory locations for N and the array X prevents a parallel execution. Even an intelligent compiler which recognizes the life time of the variables I and X in the previous examples, is incapable to see that programmers reuse some variables in different unrelated parts of the same program, merely to save memory space.

The common characteristic is that variable has a global scope and any reassignment of the variable partitions the program into two consecutive parts : a first part, using the old value and a second part, using the new value of the

variable. Such a sequencing constraint is only permitted for effective data dependencies in the algorithm.

The renaming transformation assigns different names to different uses of the same variable. As a consequence some output dependence and antidependence relations may be removed.

**Example :**

```
A = B + C          --- S
D = A + E          --- T
A = A + D          --- U
```

Here the three statements S, T and U are not independent as the variables A and D have been used at more than one places, in one statement, for reading and in other, for writing. Between statements S and T, there is WRITE/ READ dependency as the variable A which is to be used by T, is being modified by S. In statements T and U, we have READ / WRITE dependency and WRITE / WRITE dependency is observed in statements S and U. If we rename the variable A as A1 at one place where it is being written, then the WRITE / WRITE dependency in statements S and U can be removed.

```
A1 = B + C        --- S'
D = A1 + E        --- T'
A = A1 + D        --- U'
```

However, by this method, we can not remove all the dependencies. For example, it won't be possible to rename one of the two uses of variable D, as U has to use the value of D after the modification is made by the statement T. Thus, by this method only READ / WRITE and WRITE / WRITE dependencies can be removed and not the WRITE / READ dependency.

2. *Forward substitution* : The data dependencies i.e. WRITE/ READ dependencies can be removed by this method. It can be done by substituting right hand side of one assignment statement into the write hand sides of other assignment statements.

*Example* : Consider again the same example :

A = B + C --- S

D = A + E --- T

A = A + D --- U

If we substitute the assignment of variable A from statement S into T and U, we will be able to remove the data dependency of T and U on S. After the substitution we get

D = B + C + E --- T''

A = B + C + D --- U''

Again, the assignment of D may be substituted from T'' into U'', to remove the data dependency between them.

D = B + C + E --- T'''

A = B + C + B + C + E --- U'''



Now, the two statements T''' and U''' are completely independent and can be executed in parallel.

3. *Scalar expansion* : It changes a variable used inside a loop into an element of a higher dimensional array. Thus, in different iterations, the variable is not modified each time, rather, for each iteration, different array elements of the array variable are assigned the respective values.

*Example :*

```
DO 10 I = 1, N
      X = C(I)           --- S
      D(I) = X + 1      --- T
10    CONTINUE
```

Here, the different iterations of the loop are not independent as the value of the variable X is changed each time the value of C(I) changes. If we expand X to make an array variable X(I), the different values of C, as I changes, will be assigned to corresponding element of the array X(I). Now the iterations are independent.

```
DO 10 I = 1, N
      X(I) = C(I)       --- S'
      D(I) = X(I) + 1  --- T'
10    CONTINUE
```

The enforcement of the single assignment rule may require an exhaustive amount of memory, especially to store the different copies of arrays. However, a careful analysis of the data dependencies will allow a more efficient memory allocation. Each variable has a limited life time in the course of the program, i.e. from its definition upto the point where all immediate successor tasks have consumed the variable is dead and its memory space can be freed to contain the results of other tasks. In contrast to the artificial dependency constraints, this recovering of free memory space maintains the parallelism.

## **CHAPTER FOUR**

# PARALLELIZATION OF SEQUENTIAL PROGRAMS

Any sequential program consists of the following four main parts :

1. A block of assignment statements
2. Arithmetic expressions
3. Loops
4. IF statements

To convert the given sequential program, it is necessary to deal with all these parts separately, as each one of them will require different methods for parallelization.

## 4.1 BLOCK OF ASSIGNMENT STATEMENTS

In an assignment statement, we modify the value of a variable and write into it the value of the right hand side of the statement. The right hand side may be a constant, a variable, or an arithmetic expression. In the case of arithmetic expression, the value of the right hand side is to be calculated first and then it is assigned to the left hand side variable. The calculation of arithmetic expression is a problem in itself and requires large attention. This part will be dealt in later on. Here we assume that the right hand side is very simple and involves hardly any mathematical calculation. That may be a few additions and multiplications which will be performed sequentially and which has hardly any scope for reduction in time by parallel computation.

A block of assignment statements (BAS) is a sequence of one or more assignment statements with no intervening statements of any other kind. Given a block of assignment statements, we can rewrite the block after substituting from one statement into other and can obtain a set of expressions which can be executed simultaneously.

**Examples:** The method can be explained with the help of following examples:

$$X = B + C$$

$$Y = A * X$$

$$Z = X + D$$

If we execute this on a sequential machine, we will require four steps which will be performed in four time units (assuming one time step for each arithmetic operation), ignoring memory activity. But we can substitute the value of X from the first statement into second and third which then can be executed in parallel. It should be noted that before substitution, the statements are not independent.

After statement substitution, we get :

$$X = B + C$$

$$Y = A * B + A * C$$

$$Z = B + C + D$$

This can be evaluated in three steps on parallel machine.

Sometimes, after substitution, we get the assignment statement which involves many calculations. For example:

$$A = B * C + D$$

$$E = F + G * H$$

$$I = A + D$$

$$K = A + I * E$$

After backsubstitution, we get the value of K as :

$$K = (B * C + D) + (A + D) * (F + G * H)$$

The right hand side can be executed on parallel machine by performing different independent operations in parallel. This can be done using the methods for parallelization of arithmetic expressions which has been described later on. In this particular example, the tree height reduction technique can be used for the faster execution.

#### 4.2

#### ARITHMETIC EXPRESSIONS

Arithmetic expressions constitute a significant part of any program. In the sequential machine, therefore, a major part of the time will be spent in calculating the arithmetic expressions. Instead of serial computation, if we can devise a technique, by which more than one operations of an expression may be performed simultaneously by different processors, it would be possible to reduce the time taken in calculations.

In a single processor machine, where each operations are to be performed by only one processor, the time taken is obviously, proportional to the number of operations. But, in parallel machine, the operations which are independent, are allocated to different processors at the same time. The results obtained from them are, then combined into one in other time slots. The time taken in this case will be, in between  $n$  and  $\log n$  where  $n$  is the number of operations in that expression. Here  $\log n$ , which gives the best time in which ' $n$ ' operations can be performed, is the height of a **balanced binary tree** of number of leaves equal to the number of operations  $n$ . The concept of the tree formed by the expression which is called the **syntactic tree**, gives one way to parallelize the arithmetic expressions. As we know, height of the balanced binary tree will be minimum of all the trees that can be formed with the same number of leaves. The idea here is to make a syntactic binary tree with the given expression and then try to reduce the height as much as possible, or in other words, the attempt is to be made in the direction to make the tree, a balanced binary tree.

It won't be possible to reduce all the trees into balanced binary tree, because all the operations will not be independent and hence a few of them are to be executed only after certain operations are performed, even if some of the processors are sitting idle at that instant. In this case, then the attempt is made to keep the height as low as possible, not necessarily equal to  $\log n$ .

Here, it is important to note that the amount of parallelism will depend upon the number of processors used, because, suppose we have 10 operations in an expression which are independent and can be performed in parallel, but if we have a machine having only 8 processors, only 8 operations can be performed simultaneously. The remaining two will have to be performed in the next time slot. This now becomes the job scheduling problem and will not be dealt here. For the sake of simplicity, it has been considered throughout this project that the number of processors available in the machine is large enough to satisfy the requirements. It is assumed that at any instant at least one processor is sitting idle to ensure that number of processors is not a constraint.

An arithmetic expression is any well formed string composed of at least one of the four arithmetic operations (+, -, \*, %), left and right parentheses, as needed, and atoms, which are constants or variables. Let the arithmetic expression E of n distinct atoms be denoted by  $E\langle n \rangle$ .

In any expression many rules may be applied to change the form of that expression. These are:

1. Associative laws
2. Commutative laws
3. Distributive laws



The transformation is needed because we are interested in making a balanced binary tree. If from the given expression the formed tree is not balanced, the different forms of the expression obtained using the above laws are checked and the tree with the least height is selected.

#### 4.2.1 Tree Height Reduction

The height of the syntactic binary tree can be reduced using different laws as stated above. The details of the methods are described below :

##### Using associative laws :

Let us illustrate this with the help of examples. Let an expression be

$$X = \{ \{ (a + b) + c \} + d \}$$

If we calculate it as it is written, it will take three time units in obtaining the value of X, even if there are more than one processors available. It has become just like the sequential calculation on the parallel machine. The use of parantheses has imposed unnecessary constraints on the order of execution. The syntactic tree of this expression is the tree of height three as shown in fig [1.1]. If we use the associative law for addition, we can rearrange the parantheses and transform the expression into the following form:

$$X = \{ (a + b) + (c + d) \}$$

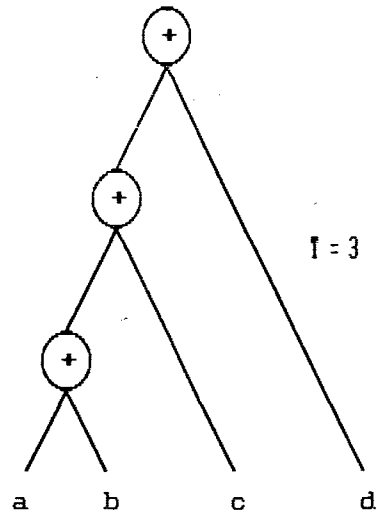


Fig.[1.1]

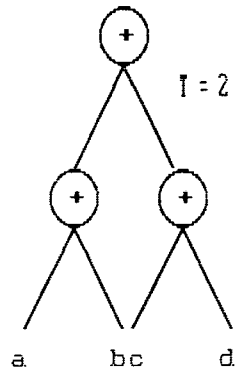


Fig.[1.2]

Tree Height Reduction Using Associative Laws

Now the two operations  $(a + b)$  and  $(c + d)$  can be performed in one time slot using two processors. The results of the two will be added in the next time slot to give the final value of  $X$ . Thus it takes only two time units in comparison to the three time units being taken by the single processor. The height of the tree has been reduced by one with the use of associative laws. The tree is shown in fig[1.2].

$$\text{Speed up in the above case} = \frac{3}{2}$$

This speedup is obtained by using 2 processors. Hence

$$\begin{aligned} \text{Efficiency} &= \frac{\text{Speed up}}{\text{number of processors used}} \\ &= \frac{3}{4} \end{aligned}$$

The efficiency is not hundred percent because in the second time unit only one addition is to be performed and thus one processor is sitting idle.

The time taken in evaluating an arithmetic expression  $E\langle n \rangle$  is given as

$$T[E\langle n \rangle] \geq \log n$$

where the base of the log is 2 and in case of log  $n$  not being integer, the ceiling of the value is taken.

**Using commutative laws:**

Let an expression be given as:

$$X = \{a + (b * c) + d\}$$

If we calculate the expression as it is given, it will take three time units. In the first time slot, the operation  $(b * c)$  is performed, then 'a' is added into the result in the next time unit and finally, in the third time unit, 'd' is added in the result of the second operation to give the value of X. The height of the tree is three. However, by the use of commutative law for addition we can reduce the height to two as shown in fig[2]. The expression in the transformed form becomes:

$$X = \{(a + d) + (b * c)\}$$

Here one addition and one multiplication operations are performed using two processors simultaneously in one time unit. The results are then added in the next slot to give the final result.

In this example also the speed up is  $3/2$  and the efficiency is  $3/4$ . Here again one processor is sitting idle in the second time unit as only one operation is to performed, leading to the efficiency less than one.

It should be noted that after the application of associative and commutative laws, the total number of operations to be performed remains the same. This point is important

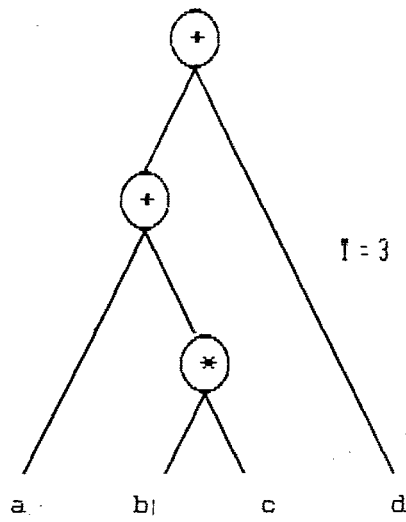


Fig.[2.1]

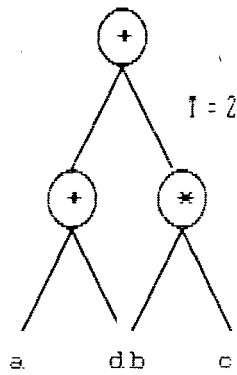


Fig.[2.2]

Tree Height Reduction Using Commutative Laws

because it is certain that application of these rules will not lead to a tree of the height more than the original one. The height will either be reduced or in the worst case, will be the same. This is not the case with the distributive laws, as will be discovered shortly, where the height of the tree may increase also. Hence it is always safe to apply the associative and commutative laws, but care has to be taken in applying the distributive laws.

**Using distributive laws:**

When the tree height reduction is not possible using associative and commutative laws, then the attempt should be made to apply the distributive laws. Application of this law increases the number of operations in the expression and in some cases may lead to the increment in the tree height. Take, for example, the following expression:

$$X = a * (b * c * d + e)$$

The syntactic tree of this expression is of height four as can be seen from fig[3.1] and contains four operations. By use of associativity and commutativity, no lower height tree can be formed. But by using the arithmetic law for the distribution of multiplication over addition, we obtain the expression

$$X = a * b * c * d + a * e$$

which has the tree of minimum height three as shown in fig[3.2]. However, the number of operations is now five. Unlike the other two operations, the distribution has introduced an extra operation. It looks rather surprising that inspite of increament in the number of operations, the time taken in evaluating the expression is reduced. This is because more operations now can be performed simultaneously than the earliar case which also compensates the increament in the operations.

In some cases the height of the tree will be increased after the application of the distribution. For example, the expression

$$X = a * b * (c + d)$$

can be computed in two steps in its undistributed form. But if we distribute the multiplication over addition, the following form is obtained:

$$X = a * b * c + a * b * d$$

which takes three steps.

Hence, non discriminative distribution is not the solution of the problem, and some sort of guidelines are required, to see whether the application of distribution will lead to a better result or not.

Before developing the algorithm for the effective distribution, it is important to know how much the tree can

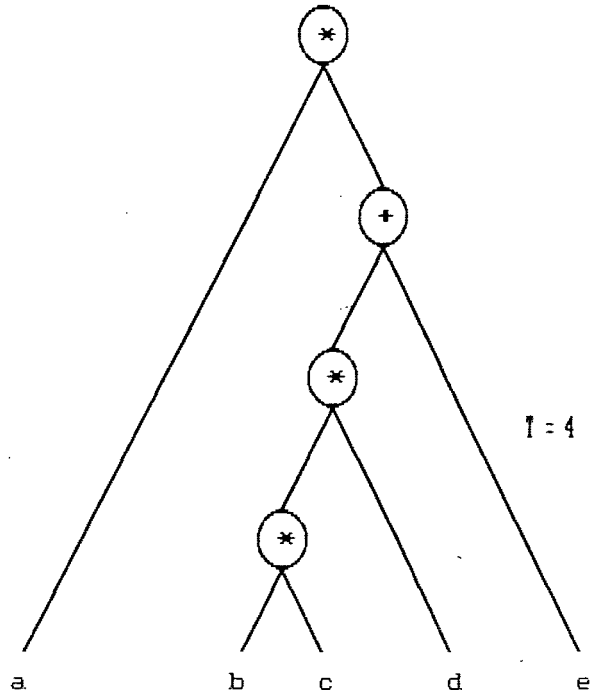


Fig.[3.1]

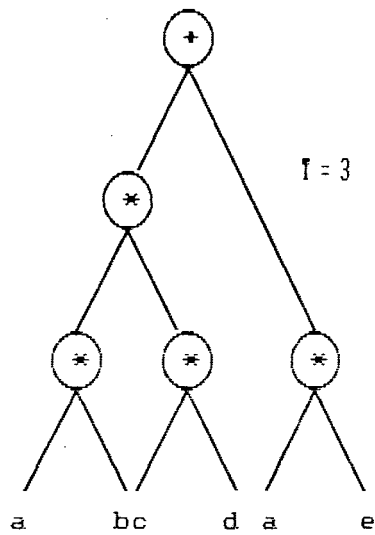


Fig.[3.2]

Tree Height Reduction Using Distributive Laws



be reduced by using associative, commutative and distributive laws. A lot of research has been done regarding this and the following theorems will provide the answer:

Let  $E\langle n \mid d \rangle$  be any arithmetic expression with depth  $d$  of parenthesis nesting. By use of associativity and commutativity only,  $E\langle n \mid d \rangle$  can be transformed in such a way that

$$T[E\langle n \mid d \rangle] \leq \log n + 2d + 1.$$

If the depth of parenthesis nesting  $d$  is small, then this bound is quite close to the lower bound of  $\log n$ . The more the depth of parenthesis nesting, the more will be the time required and will tend towards  $n$ , the linear time complexity.

Unfortunately, many expressions can not be transformed using only associativity and commutativity into such a form as to give the minimum tree height. Use of distribution is required almost every time. Given any expression  $E\langle n \rangle$ , by the use of associativity, commutativity and distributivity, it can be transformed such that

$$T[E\langle n \rangle] \leq 4 \log n$$

with number of processors  $P \leq 3n$ .

#### 4.2.2 Distribution Algorithm:

To guide about when to use the distribution for the minimization of tree height, an algorithm is required. This

will be called as distribution algorithm. As we know, the number of leaves in a balanced binary tree is the integer power of 2. So, if there are  $2^n$  operands which can be used simultaneously, they will form a part of the balanced tree. If this is not the case, then a 'hole' will be there, which has the capacity to include some more operands in the same height. This point gives one possibility of getting a reduced height tree than the earlier one. This could be better illustrated with the help of examples. Let the expression be:

$$X = a * (b * c * d + e)$$

Here there are two multiplications in the term inside the paranthesis which has three operands b, c, and d. This is not the whole power of 2 and we say that there is a hole available in it. This hole can be filled by any extra operand without affecting the tree height. This operand can be obtained, if we distribute the multiplication over addition to get the expression as:

$$X = a * b * c * d + a * e$$

Now the hole has been filled and the tree height which was four earliar, is reduced to three.

Now consider the expression

$$X = a * (b * c + d) + e$$

in which the paranthesized expression contains no hole. This is because we won't get the reduction in tree height

after the distribution is applied. The expression :

$a * (b * c + d)$  and its distributed form

$a * b * c + a * d$  have the same tree height.

But the expression X requires four time units in evaluating in its original form whereas the distributed form:

$X = a * b * c + a * d + e$

requires only three steps. This is because one space which is empty, is filled by the operand 'e'. This is called the space filling operation and it reduces the tree height for the above example because the lowest height trees for  $a*b*c + a*d$  and  $a*b*c + a*d + e$  are of equal height while the lowest tree for  $a*(b*c + d) + e$  is higher than that for  $a*(b*c + d)$ . What happens here is that  $a*(b*c + d)$  is opened by distributing a over  $b*c + d$  and the "space" to place 'e' is created.

Given an assignment statement A, the distribution algorithm derives the assignment statement  $A^d$  by distributing multiplications over additions properly so that the height of  $A^d$ , denoted by  $h[A^d]$  is minimized. The algorithm works from the innermost paranthesis level to the outermost paranthesis level of assignment statement and requires only one scan through the entire assignment statement. Let us assume that additions and multiplications require the same amount of time, i. e. one unit of time.

The minimum height of tree using only the associative and commutative laws can be built as follows. Let us assume that either  $A = \text{sum of } (t_i)$  or  $A = \text{product of } (t_i)$ , where  $t_i$  denotes arbitrary arithmetic expression, including single variables. Let for each  $i$  a tree of minimum height  $T[t_i]$  has been built. Then first we choose two trees, say  $T[t_p]$  and  $T[t_q]$ , each with a height smaller than the height of any other tree. We combine these two trees and replace them by a new tree whose height is one higher than the maximum of  $\{h[t_q], h[t_p]\}$ . These procedure is repeated from innermost paranthesis level to the outermost paranthesis level, and at each level it is repeated until all trees are combined into one tree.

To see the effect of distribution of multiplication over addition, let us examin all the ways, paranthesis can occur in an expression. There are only four possible ways:

$$P_1. \quad \dots + (A) + \dots$$

$$P_2. \quad \dots \# (t_1 * t_2 * \dots * t_n) * (t_1 * t_2 * \dots * t_m) \# \dots$$

$$P_3. \quad \dots \# a_1 * a_2 * \dots * a_n * (A) \# \dots$$

$$P_4. \quad \dots \# (t_1 + t_2 + \dots + t_n) * (t_1 + t_2 + \dots + t_m) \# \dots$$

where  $\#$  denotes either addition or multiplication or no operation.

From careful inspection we can conclude that distribution in case  $P_3$  and partial distribution in case  $P_4$  are the

only cases that should be considered for lowering tree height. In cases  $P_1$  and  $P_2$ , removal of parenthesis leads to a better result or at least gives the same tree height. Full distribution in case  $P_4$  always increases the tree height and should not be done. Also, it should be clear that in any case the tree height of an arithmetic expression cannot be lower than that of a component term even after distribution is done. This assures that evaluation of distribution can be done locally. that is, if the distribution increases the tree height for a term, then the distribution should not be done because once the tree height is increased, it can never be remedied by further distributions.

At each level of parenthesis pair, for cases  $P_3$  and  $P_4$ , instances of holes and spaces are checked and proper distribution is performed.

In brief, the algorithm goes as follows. Start from the innermost parenthesis level and find the 'hole' and 'space' available in that. See if it can be filled using the distribution. Then go to the next parenthesis level and repeat the same. At each level, it is necessary to check the tree height to insure that it has not become more than the original tree height. In that case the remaining distributions will unnecessarily be done and it is beneficial to leave the attempt for the reduction of tree height. The same process is repeated until we come to the outermost parenthesis level. Before the use of distribution, the application of associative and

commutative laws should be applied where\_ever possible. If this does not reduce the tree height, it is sure that it won't increase also.

The algorithm reduces the tree height. But it should be noted that it does not give the minimum tree height arithmetic expression, e.g., it does not factor the expression

$$A = a * c + a * d + b * c + b * d \text{ into}$$

$$A' = (a + b) * (c + d)$$

to reduce the tree height.

Subtractions can be introduced into an arithmetic expression without affecting the distribution algorithm. The only modification necessary is to change operators as required, e.g.,  $A = a + b - (c + d)$  for  $A = a + b - c - d$ . Divisions may require special attention since a numerator cannot be distributed over denominator, e.g.,

$$a/(b + c + d) \langle \rangle a/b + a/c + a/d.$$

Hence in general, tree height reduction is done for the numerator and denominator independently, then distribution of the denominator over the numerator is considered.

#### 4.2.3 Recurrence Relations:

Linear recurrences share with arithmetic expressions a role of central importance in computer design and use, but they are somewhat more difficult to deal with. While an

expression specifies a static computational scheme for a scalar result, a recurrence specifies a dynamic procedure for computing a scalar or an array of results. Linear recurrences are found in computer design, numerical analysis, and program analysis, so it is important to find fast and efficient ways to solve them.

Recurrences arise in any logic design problem that is expressed as a sequential machine. Also, almost every practical program that has an iterative loop contains a recurrence. Not all the recurrences are linear but a vast majority found in practice are.

**Examples:**

Consider the problem of computing an inner product of vectors  $a = (a_1, a_2, \dots, a_n)$  and  $b = (b_1, b_2, \dots, b_n)$ . In the recurrence form this can be written as

$$x = x + a_i b_i, \quad 1 \leq i \leq n$$

where  $x$  is initially set to zero and finally set to the value of the inner product of  $a$  and  $b$ .

This equation can be expanded by substituting the right hand side into itself (statement substitution) as follows:

$$\begin{aligned} x &= a_1 b_1 \\ x &= a_1 b_1 + a_2 b_2 \\ x &= a_1 b_1 + a_2 b_2 + a_3 b_3 \\ &\vdots \\ &\vdots \\ &\vdots \end{aligned}$$

After  $n$  iterations we can have expression which can be mapped onto a balanced binary tree.

As another example of a linear recurrence which produces a scalar result, the evaluation of degree  $n$  polynomial  $p_n(x)$  in Horner's rule form can be expressed as:

$$p = a_i + xp_i \quad 2 \leq i \leq n$$

where  $p$  is initially set to  $a_1$  and finally set to the value of  $p_n(x)$ .

#### **Evaluation of recurrence relations on parallel machines:**

As we saw earlier, how the recurrence relations can be expanded to give a set of equations by the substitution of right hand side into itself. If only the final value is required, the last equation can be treated as the arithmetic expression, which can be parallelize using the methods dealt for the same. The tree (syntactic tree) is formed which then is reduced in the height to form the balanced binary tree. If instead of only the last value, we require all the values from 1 to  $n$ , then another approach is to be applied. The method can be explained with the help of an example:

Take the example of solving the set of  $n$  linear equations with the help of Gauss method. We will be able to



transform the given equations into the lower triangular matrix form as given below :

$$x_1 = c_1$$

$$x_2 = a_{11}x_1 + c_2$$

$$x_3 = a_{21}x_1 + a_{22}x_2 + c_3$$

⋮  
⋮

$$x_n = a_{(n-1)1}x_1 + a_{(n-1)2}x_2 \dots + c_n$$

The value of  $x_1$  which is known from the first equation to be equal to  $c_1$ , is broadcasted to all the equations and the first column is calculated simultaneously and the values of constants are added in all by  $n$  processors in one step. Now we have the value of  $x_2$  which is again broadcasted to the remaining  $(n-1)$  equations and the second column is now evaluated and added to the previous result to give the value of  $x_3$ . This procedure is repeated till the last value  $x_n$  is evaluated. Thus each step consists of broadcasting the previous value, multiplication with the coefficients and addition to the previous result. We require  $(n-1)$  processors in the first step,  $(n-2)$  in the second and fewer thereafter. Thus only  $(n-1)$  processors are sufficient for this method.

If we solve these equations with the single processor machine, we have to perform 2 operations (1 addition and 1 multiplication) for evaluating  $x_2$ , 4 operations for  $x_3$ , 6 for  $x_3$  and similarly,  $2(n-1)$  for  $x_n$ . Thus the total operations required by a single processor machine is given by :

$$\begin{aligned}
T_s &= 2 + 4 + 6 + \dots + 2(n-1) \\
&= 2 [1 + 2 + 3 + \dots + (n-1)] \\
&= 2 [n(n-1)/2] \\
&= n(n-1) \\
&= O(n^2)
\end{aligned}$$

On the parallel machine, by the method described above, the time taken in solving the set of equations is given by:

$$\begin{aligned}
T_p &= 2(n-1) \\
&= O(n)
\end{aligned}$$

Hence the speedup obtained by the above method is

$$\begin{aligned}
S_p &= \frac{n(n-1)}{2(n-1)} \\
&= n/2
\end{aligned}$$

The efficiency of the system is given by :

$$\begin{aligned}
E_p &= \frac{S_p}{p} \\
&= \frac{n/2}{(n-1)} \\
&> 1/2
\end{aligned}$$

The efficiency of the above described method is more than fifty percent and also this method is of order  $n$  in comparison with  $n^2$  of the serial algorithm. Hence it can be said that this method is quite reasonable.

A parallelizing compiler explores parallelism and generates parallel code to take advantage of the underlying parallel architecture. In order to be valid, any transferred parallel code of a given program must honour the dependence constraints in that program. It is true in general that the more precisely and completely dependence information is explored, the more efficient parallel code can be generated. Loops construct a major part of any program and are most difficult to parallelize. It seems, at first sight that loops can not be parallelized, as each iteration requires the value of the previous iteration. But it is not the case, as will be seen shortly. There will be some independent iterations which can be executed in parallel by different processors. The aim here is to explore the iterations which may be evaluated simultaneously and run them in parallel.

In a loop, there will be two kinds of dependencies present. One is the dependence of one statement on the other of the loop body, and the other is the dependence of one iteration on the other. The first one is the same as dealt in earlier in the chapter 'Dependence Analysis' and is very simple in comparison to the second kind of dependency. Here we will see the dependence constraints among different iterations and will make an attempt to remove them to the extent possible.

A DO loop in which no cross\_iteration dependence relation is present is called a DO all loop. All the iterations of such a loop can be executed in any order or even simultaneously without any extra synchronization. Parallelizing a DO all loop is simply assigning iterations to different processors.

On the other hand, a loop is termed DO serial loop if some cross iteration dependences exist. To obey dependence constraints, a DO serial loop can be simply executed in serial by a single processor. However, any nested loop can be executed in parallel as long as the cross iteration dependences are satisfied.

There are some major difficulties in parallelizing nested loops which carry cross iteration dependences. First to correctly insert synchronization primitives, compilers or programmers have to find out all cross iteration dependences. It is very difficult to find a dependence analysis technique which can effectively find out all cross iteration dependences unless the dependence pattern is uniform for all iterations. Second, even though all cross iteration dependences can be identified, it is difficult to systematically arrange synchronization primitives especially when the dependence pattern is irregular. Also, the synchronization overhead which will significantly degrade the performance should be minimized.

### 4.3.1 Simple Loops

There seems to be very less scope for parallelization in case of simple loops. But there are some cases where we will be in position to execute that in parallel. Take for example the execution of the following loop:

```
DO 10 i = 1,n
      x(i) = a(i-1) * x(i-1)
10 CONTINUE
```

We must usually repeat the computation sequentially for each index value, if the value of  $x(n)$  is to be calculated. In other words we have to calculate the following set of  $n$  statements sequentially:

```
x(1) = a(0) * x(0)
x(2) = a(1) * x(1)
.
.
.
x(n) = a(n-1) * x(n-1)
```

Instead of computing sequentially, we can get the value of  $x(n)$  in comparatively less time if we use the backsubstitution to get the value of  $x(n)$  and execute that in parallel using the tree height reduction technique.

```
x(n) = a(n-1) * x(n-1)
      = a(n-1) * a(n-2) * x(n-2)
      .
      .
      .
      = a(n-1) * a(n-2) * ... * a(1) * a(0)
```

This will be completed in time of the order of  $\log n$  in comparison to the order  $n$  for the serial calculation.

The different iterations of a loop will be executed in parallel if the body consists of vector elements, for example:

```
DO 20 I = 1, 100
    A(I) = B(I) + C(I)
20 CONTINUE
```

Here the 100 iterations of the loop will be computed in one time unit if we use 100 processors. Each variable is a vector array and stores all hundred values in hundred memory locations. Corresponding values are used by different processors and the values of other locations are not of any use to that.

#### **Simultaneous execution of statements in a loop:**

The statements in the body of a loop can be executed in parallel with slight modifications which make them independent. Take, for example, the following loop:

```
DO 10 I = 1, 40
    A(I) = A(I-1) * B(I-1)    --- S1
    B(I) = A(I) / 4          --- S2
10 CONTINUE
```

In the above example, the two statements S1 and S2 cannot be executed in parallel for all values of I. However, S1 and S2 can be computed simultaneously when I varies sequentially if the index expression in S2 is changed as follows:

```

DO 10 I = 1, 40
      A(I) = A(I-1) * B(I-1)      --- S1
      B(I-1) = A(I-1) / 4        --- S2
10  CONTINUE

```

Now the two statements are independent and hence can be computed concurrently.

#### Partial execution of loops:

In some cases it is not possible to parallelize the whole loop as given. In that case we look for the partial parallelization. This will be clear from the example below:

```

DO 20 I = 1, 100
      B(1) = A(2*I + 500)        --- S1
      A(7*I + 35) = C(I) * C(I+1) --- S2
20  CONTINUE

```

Here, the two statements don't seem to be independent and in fact they are not for all values of I. But for values of  $I < 95$ , the computation of S1 does not use a result of S2. So, for these values, S1 may be computed simultaneously. But for  $I > 94$ , the loop must be computed sequentially.

#### 4.3.2 Nested Loops

The case of nested loops is more promising and a challenging job and will be explored now. It has to be noted that we will not always be in a position to parallelize any kind of loop. Only a few very simple loops in which the cross iteration dependences can be easily found, can be made to run in parallel.

##### **Wave Front Method:**

For loops with cyclic dependencies involving variables with more than one subscript, the wave front method may be used to parallelize the loop to some extent. It effectively extracts the array operations from the loop. In evaluating the value of a variable in a particular iteration we may require some values from the previous iterations. Thus it seems that the present iteration can not be executed until unless the previous iterations have been performed. But it is to be noted that not all the values of previous iterations will be required. The present value is dependent only upon a few of them. This basic point is exhausted in the wave front method. We can always calculate all those values in parallel which are not required by each other.

As the same statements are executed in each iteration, there becomes a specific pattern of the values which can be calculated simultaneously in each iteration. This specific pattern is called the wave front and it may be a line, a



plane or any other pattern. This will depend upon the statements in the loop i.e. the body of the loop. The objective here is to find out that particular pattern and execute all points on that in parallel.

**Examples:**

```
DO 10 I = 1,10
DO 10 J = 1,10
A(I, J) = A(I, J-1) + B(J)
10 CONTINUE
```

To evaluate this on a single processor system would require 100 unit time steps for addition. However on a parallel machine the ten statements given below can be computed simultaneously in I while J takes on the values 1, 2, 3, ..., 10, sequentially.

```
A(1, J) = A(1, J-1) + B(J)
A(2, J) = A(2, J-1) + B(J)
.
.
.
A(10, J) = A(10, J-1) + B(J)
```

Thus the the computation time is reduced to ten units.

Here we calculated all the operations simultaneously along a line parallel to the J axis. Thus the wave front in this case is a line. Instead of calculating along J axis, we may also use the I axis and will end up with the same result.

We can further reduce the time by using back substitution and building a minimum height tree for each of these ten statements. Thus we have for each I,

$$A(I, 10) = A(I, 0) + B(1) + \dots + B(10)$$

This can be computed in four time units.

Now consider the another example:

```
DO 10 I = 1, L
DO 20 J = 1, M
DO 30 K = 1, N
    U(J, K) = [U(J+1,K) + U(J,K+1) + U(J-1,K)
              + U(J,K-1)]
30     CONTINUE
20     CONTINUE
10     CONTINUE
```

The above loop will take LMN time units on a sequential machine. We want to speed up the execution by performing some of the operations concurrently. First take the inner two loops only. At first instance it seems that the value of  $U(J,K)$  is dependent upon all previous values of current and next values of last iteration. But there is, however, a parallel structure that can be exploited here. If the cells of the matrix are laid out on a chess board, then the iteration of the program shows how to update the value of a black squire by adding the values of the neighbouring white squires and similarly, how to update a white squire by the

values of its neighbouring black squares. The black and white squares form two sets of variables, since no white square depends directly on a white square and no black square depends upon the black square. In fig[4], it is seen that in order to calculate  $U(1,1)$ , only values of  $U$  on edge points are needed. To calculate  $U(1,2)$  and  $U(2,1)$  we require only a value on the edge point and  $U(1,1)$ . More generally, to calculate values of  $U$  on a given diagonal, we require the values of  $U$  on the previous diagonal. Hence all the diagonal elements can be calculated simultaneously.

Here the wave front is the line parallel to the diagonal of the square. So, if we calculate the values of  $U$  along the transformed axis  $J'$  and  $K'$  such that

$$J' = J + K$$

$$K' = K$$

we will be able to calculate the diagonal values concurrently.

After the transformation, the inner two loops become

```

DO 20 J' = 2, M+N
    DO 30 CONC FOR ALL K'
        U(J'-K',K') = [U(J'-K'+1,K') + U(J'-K',K'+1)
                        + U(J'-K'-1,K') + U(J'-K',K'-1)]
    30    CONTINUE
20    CONTINUE

```

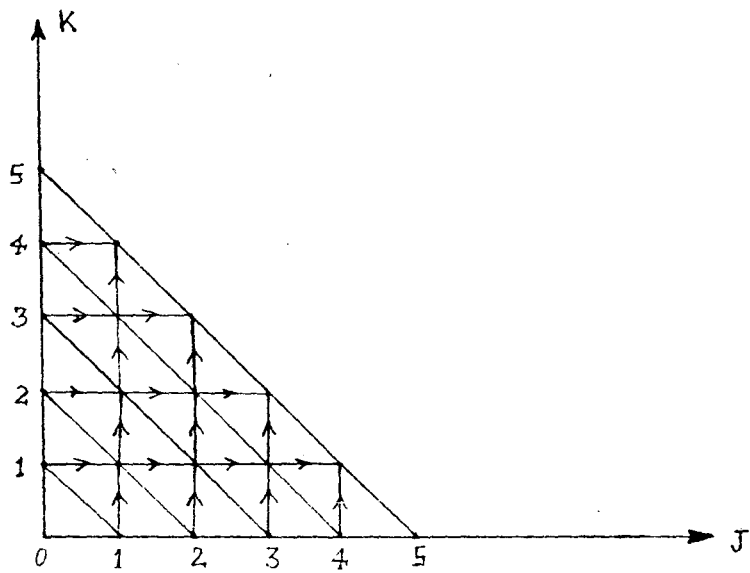


Fig [4]

The execution of the above loop will require only  $M+N-1$  sequential iterations instead of the  $MN$  iterations necessary for the computation of the original loop.

Now we will consider all the three loops with the three indices  $I, J, K$  of the above considered example. It is seen that the values at planes parallel to  $2x + y + z = \text{constt.}$  are independent and hence can be calculated in parallel. This is equivalent to the transformation

$$I' = 2I + J + K$$

$$J' = I$$

$$K' = K$$

After this transformation in the loop, we get the following program segment:

```

DO 10 I' = 4, 2L + M + N
      DO 20 CONC FOR ALL (J', K')
            U(I'-2J'-K',K') = [U(I'-2J'-K'+1, K')
                                +U(I'-2J'-K', K'+1)
                                +U(I'-2J'-K'-1, K')
                                +U(I'-2J'-K', K'-1)]
20      CONTINUE
10      CONTINUE

```

Here, the computation requires  $2L+M+N-3$  sequential iterations instead of the  $LMN$  iterations necessary for the computation of the original loop.

It should be noted that the wave front method is applicable to the arrays of more than two dimensions only. This method is of no use for one dimensional arrays, since it degenerates to a serial computation in this case. In those cases we will consider the cyclic dependence as a linear recurrence relation and can make them to execute in parallel.

The diagonal scheme has one more serious disadvantage. Some diagonals will be very short while the others may be quite large. As all elements of a diagonal are to be executed in parallel, the no of processors in the case of a long diagonal will be greater than the case of shorter diagonal. Thus many processors will remain idle for a large portion of time which limits the efficiency.

#### **Distribution of loops:**

For acyclic graphs, it has been shown that the statement substitution can be performed between any pair of nodes which have a dependence relation. As in a block of assignment statements, we substitute for each LHS variable of one statement, the RHS of another statement, which is the cause of dependence relation, the corresponding arithmetic expression properly shifted. By applying statement substitution, the dependence relation is removed and a set of independent assignment statements results. Each of these represents a vector assignment statement, all of which can be executed simultaneously.

In loops with acyclic graphs, it is possible to reduce the graph for the entire loop to a set of independent nodes representing simultaneously executable array statements. By distribution of loops we mean the distribution of the loop control statements over individual or collections of assignment statements contained in the loop. The purpose of distributing a given loop is to obtain a set of smaller size loops, which upon execution give results equivalent to the original loop. By this it will be possible to reduce the dependence among them and thus they can be made to run in parallel. The distribution of loops is similar to the distribution in arithmetic expressions and may introduce more parallelism into a program loop than that obtained from an undistributed one.

The loop distribution algorithm goes as follows. By analyzing subscript expressions and indexing patterns, first a dependence graph  $G$  is constructed. On this graph  $G$  we get the partition of nodes and form a set in such a way that any two statements in the set are in same subset if there dependence relations constitute a cycle. Then a partial ordering on this block is established and the blocks should be sequenced such that the origin of an arrow is executed before its end. Now we can replace the original loop by loops for different blocks obtained so far.

If the dependence graph is acyclic, then assignment statements are handled as expressions for array operations. If the dependence graph is cyclic, the blocks are handled by

the recurrence method. The condition for partial ordering relation insures that data are updated before being used. Hence, any execution order of the set of loops which replaces the original one will be valid as long as this relation is not violated. In general, we can also use the statement substitution to remove this relation between some or all of the distributed loops.

**Example:** To understand the loop distribution, take the following program:

```

DO 50 I = 1, N
10      A(I) = B(I) * C(I)          --- S1
DO 30 J = 1, N
20      D(J) = A(I-3) + E(J-1)    --- S2
30      E(J) = D(J-1) + F         --- S3
DO 40 K = 1, N
40      G(K) = H(I-5) + 1         --- S4
50      H(I) = A(I-2) + 3         --- S5

```

The dependence graph of this program is shown in the fig[5.1]. Loop nesting has been denoted using brackets.

Now we form the partition  $p = \{p_1, p_2, p_3, p_4\}$  where  $p_1 = \{S_1\}$ ,  $p_2 = \{S_2, S_3\}$ ,  $p_3 = \{S_4\}$  and  $p_4 = \{S_5\}$ .

The partial orders are now obtained to be  $(p_1, p_2)$ ,  $(p_1, p_4)$ , and  $(p_4, p_3)$ . This means that the output of  $p_1$  is required to  $p_2$  and  $p_4$  and that of  $p_4$  to  $p_3$ . Hence  $p_1$  must be executed before  $p_2$  and  $p_4$  and  $p_4$  must be executed before  $p_3$ .



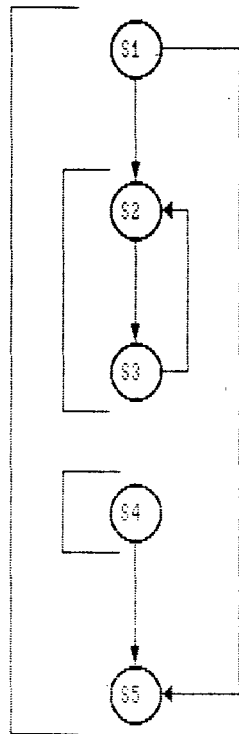


Fig. (5.1) Program Graph.

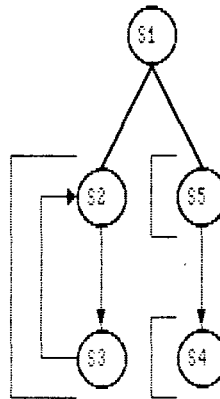


Fig. (5.2) Distributed Graph

This much information is sufficient to form the distributed graph which is shown in the fig[5.2].

This graph could be used to compile the given array operations as follows. First S1 is executed, which is a vector multiplication. After this either p2 or p4 is executed. Execution of p2 can be done in parallel by the method described for the simple loops, Execution of p4 is again a vector operation and can be executed in parallel. Finally S4 may be executed for all I and K simultaneously. This requires the broadcasting of elements of the H array to all elements in the columns of G.

The time required to execute p1, p3, and p4 using  $O(N)$  processors is a constant (independent of  $N$ ). The overall execution time is dominated by p2 and is  $O(\log N)$ . The number of processors required to achieve this time is  $O(N)$ .

In the above example, the statement substitution could have been used. By using statement substitution, we would have been able to obtain four p\_blocks, all of which could be executed at once. This would require the execution of several different operations at one time, while the technique used above allows all operations at each step to be identical. Because, p2 dominates the time here, very little additional speedup would have been possible by the substitution.

A program written in a conventional language tends to exhibit little parallelism because of many control dependencies due to IF statements. To select an appropriate path that follows an IF statement, one must wait until the condition part of the IF statement is executed. A model IF statement is shown bellow:

```
IF x(I)
    THEN S(I)
    ELSE T(I)
```

Here x is a Boolean expression and S, T are assignment statements. If the value of x is TRUE then the expression S is evaluated and if the value of x is FALSE then T is to be calculated.

To parallelize the IF statement, we can make a scheme according to which we calculate the statements S and T simultaneously, before calculating the condition part x(I). It means, before knowing exactly which path is to be followed, we have to calculate the values of all branches. These values are kept stored with a pointer and one of them is used when the condition part is evaluated. The remaining are of no use. Though this scheme causes much overhead in calculating even the expressions which are not required, it is naturally an effective way to overcome the serious limitation on parallelism because of IF statements.

To convert an existing sequential program into the one executable on parallel machines is the requirement for fast and efficient execution of the program. Today, almost all machines used in any field are based on parallel processing techniques. More faster chips are available today and research is still going on in that direction. But the speed of a physical device cannot be increased beyond a certain limit. The science is continuously moving towards that limit. The day is not very far when it will be said that now it won't be possible to increase the speed of chips any more. But since, our need for faster execution will never end, it is an obvious approach to look for the improvement in software part.

Most of the programs written today are sequential. If one has to use parallel machines, either a completely new program is to be developed or the existing program is to be converted in parallel form. The first approach is hardly scientific as the unnecessary work has to be done in developing the already developed program. So, it is almost always beneficial to use the second approach.

It is very tedious work to convert the serial program in parallel form manually and more than that it requires a large amount of time. If something can be done to reduce this time, it will save much time and human power. The automatic parallelization is the right approach in this direction.

By automatic parallelization, it is meant that the programmer is not to be worried about the conversion, rather this work is to be performed by the compiler itself. The compiler should be designed such that it accept the sequential program, check for the portion which can be parallelized and convert that into the parallel form.

One thing should be noted here that not all the programs can be changed into parallel form and even in a program, only some portion can be converted. The compiler has to identify that portion first and then it should apply the methods on only those portions.

#### ALGORITHM

START:

Read the statement S. Check the type of statement.

If it is an **assignment statement**, GO TO **ASSIGN()**.

If it is an **arithmetic expression**, GO TO **ARITH()**.

If it is a **loop**, GO TO **LOOP()**.

If it is an **IF statement**, GO TO **IF()**.

Else goto **OTHER()**.

**ASSIGN() :**

i =1.

10. Find **IN(S<sub>i</sub>)** and **OUT(S<sub>i</sub>)**.

i = i + 1.

Read (Si).

If it is assignment statement, GO TO 10.

n = i.

For all i = 1 to n

    For all j = 1 to n and i <> j

        A(k) = IN(Si) int. OUT(Sj).

        B(k) = OUT(Si) int. OUT(Sj).

    end (\* of loop)

end (\* of loop)

If B(k) <> phi for all k then rename the common elements of corresponding statement S to some other.

Note the statements for which A(k) = phi. These can be executed in parallel.

GO TO START.

ARITH() :

If the arithmetic expression is **recurrence relation**, then use the back substitution to get the simple arithmetic expression.

20. Read the character of the right side of the expression.

If it is operand, then write as it is.

else

    If it is '+', then if the next character is '(', drop the '(' keeping the '+'.

    If it is '-', then if next character is '(', drop

it changing all '+' into '-' and vice-versa till  
)' occurs which is then dropped.

If it is '\*', '/', '(' or ')' then just keep it.

GO TO 20.

Apply associative and commutative laws such as to keep  
one type of operations at one place.

If one or both the operands of '\*' are a group of  
operands, then count the number of operands on each  
side. If it is not the whole power of 2 then apply the  
distribution law.

Use the distribution algorithm to get the minimum  
height tree.

LOOP() :

Find the body of the loop. Also see whether it is  
simple or nested.

If the loop is simple, use the back substitution to  
form an arithmetic expression.

If there are more than one assignment statements, use  
the method for assignments statements to parallelize  
them.

If the loop is nested, and array variables have been  
used, then find the wave front along which all elements  
can be computed concurrently.

If there are more than one loops, use loop distribution.

IF() :

Keep the condition part as it is.

Find both the 'THEN' and 'ELSE' part of the IF statement. Both should be executed concurrently.

Parallelize the body of 'THEN' and 'ELSE' part by using the relevant methods described above.

OTHER() :

Write the statement as it is.

If it is the end of the program, mark the END else GO TO START.

#### Illustrative Example

The following program has been taken as an example to show how the algorithm will work.

(\* Sequential example program \*)

```
INTEGER A,B,C,D,E
DIMENSION P(50), Q(50), R(50)
DIMENSION S(50, 50)
READ (5, *) A,B,C,D,E
A = B * C
D = B + C
A = A + D + E
```



```

IF (A > 50) THEN
    A = C - B * (D + E) + E * (B*C*D + F) - (E - F) + D
    B = B + C * D - E
ELSE
    N = A
    DO 10 I = N-1, 1, -1
        A = A * I
10.    CONTINUE
        B = C / D
        E = A - B
END IF
READ (5, *) P,Q,R
DO 20 I = 1, 50
    P(I) = Q(I) + R(I)
20.    CONTINUE
    READ (5, *) S
    DO 30 I = 1, 50
        DO 40 J = 1, 50
            S(I, J) = S(I, J-1) + Q(J) - R(I-1)
40.    CONTINUE
30.    CONTINUE
        WRITE (5, *) A,B,C,D,E
        WRITE (5, *) P,Q,R
        WRITE (5, *) S
STOP
END.

```

The program written above is capable of being executed on a sequential machine. Although it does not perform any sensible work, it is intentionally written such that all types of parts in an ordinary program are covered here. Now it will be seen how the algorithm described above works in this case.

The algorithm starts by reading the statement. All the statements are kept as it is until we reach the first assignment statement  $A = B + C$ . Now it calculates  $IN(S1)$  and  $OUT(S1)$ .

$$IN(S1) = \{B, C\}$$
$$OUT(S1) = \{A\}$$

Similarly for all other assignment statements

$$IN(S2) = \{B, C\}$$
$$OUT(S2) = \{D\}$$
$$IN(S3) = \{A, D, E\}$$
$$OUT(S3) = \{A\}$$

Now it calculates

$$B(1) = OUT(S1) \text{ int. } OUT(S2)$$
$$= \text{phi.}$$
$$B(2) = OUT(S1) \text{ int. } OUT(S3)$$
$$= \{A\}$$
$$B(3) = OUT(S2) \text{ int. } OUT(S3)$$
$$= \text{phi.}$$

Now see which one of the above is not 'phi'. Only B(2) is so, which is equal to {A}. Hence we have to rename 'A' in S1 to 'A1'.

Now calculate A(1):

$$\begin{aligned} A(1) &= \text{IN}(S1) \text{ int. } \text{OUT}(S2) \\ &= \text{phi.} \end{aligned}$$

Similarly we find out all other values of A(k) and note which of them are 'phi'. As OUT(S1) and OUT(S2) are the elements in IN(S3), we substitute the values of A and D from S1 and S2 into S3.

Now, the three statements are independent and can be computed concurrently in one step.

DO ALL CONCURRENTLY

$$\begin{aligned} A1 &= B * C \\ D &= B + C \\ A &= B * C + B + C + E \end{aligned}$$

After this, the algorithm moves to the next statement which is IF statement. First the body of the 'THEN' and 'ELSE' part is determined and both parts are executed in parallel.

The first statement in the body of 'THEN' is the arithmetic expression. As per the algorithm, we first drop

the unnecessary paranthesis and apply associative and commutative laws. After this we get:

$$A = C + F - (B + E) + B * (D + E) + E * (B * C * D + F)$$

Now, the distributive law is applied only to the last term as the number of operands in that are not the whole power of 2.

After this we get the expression of minimum tree height as given below:

$$A = C + F - (B + E) + B * (D + E) + E * B * C * D + E * F$$

The above expression can be calculated in three steps only:

Step 1.

DO ALL CONCURRENTLY

$$x = C + F$$

$$y = B + E$$

$$z = E * F$$

$$w = E * B$$

$$u = C * D$$

$$v = D + E$$

Step 2.

DO ALL CONCURRENTLY

$$r = x - y$$

$$s = B * v$$

$$t = w * u$$

Step 3.

$$A = r + s + t + z$$

Similarly the second expression can be executed in two steps only:

Step 1.

DO ALL CONCURRENTLY

$$x = B - E$$

$$y = C * D$$

Step 2.

$$B = x + y$$

The body of ELSE part consists of a DO loop and two assignment statements. The loop can be changed into arithmetic expression using substitution. We get:

$$A = A * (A-1) * (A-2) * \dots * 2 * 1$$

This can be computed using tree height reduction. The other assignment statement can be computed simultaneously with the loop but the third has to be executed in the next step as the input variables of this are the output of the previous two.

After this we come across another loop. This has the array variables and hence can be executed concurrently.

DO ALL CONCURRENTLY

$$P(1) = Q(1) + R(1)$$

$$P(2) = Q(2) + R(2)$$

·  
·  
·

$$P(50) = Q(50) + R(50)$$

The next statement is the nested loop and we have to apply the wavefront method here. The wave front here is the line parallel to the I axis. For one value of I, we can calculate the loop for all J in one step. Similarly, all values for different I can be calculated. The calculations for different values of I are to be done serially as the two iterations are not independent for I.

After the loop, we come across the 'write' statements which as per the algorithm, are kept as they are. Finally, the end of the program is encountered. This marks the end of the algorithm.

The example described above gives the general idea how the sequential program may be converted in the parallel form.

## **CHAPTER FIVE**

## CONCLUSION

In this project an attempt has been made to develop the methods to automatically convert the existing sequential program into parallel form. All basic parts of an ordinary program were considered and some ways to parallelize them were described. For the major portion of time, the involvement was in the dependence analysis among the assignment statements, the methods to remove these dependencies, the ways to parallelize a block of assignment statements, arithmetic expressions, the tree height reduction technique of parallelization, recurrence relations and use of back substitution for executing them in parallel. The ways discussed can be used to automatically parallelize the corresponding part of the program.

Although very little time was available for the main part of the program, i.e. the loops and IF statements, yet some ways were described for certain special types of loops specially for simple loops. The nested loops with array variables were discussed and a method known as wave front method was explained for their parallelization. Although by seeing we can know which wave front is to be used in the given case, no method could be described how the compiler would know the equation of the wave front. Until unless this is known, this method is hardly useful in automatic parallelization. Due to the less time available, this has been left.



The loops discussed were very basic in nature and had one or other constraints imposed. A large variety of loops are still left untouched. Some method can be discovered which can be used for any general loop. Again due to short time the attempt in that direction could not be successful.

Very little has been done for the parallelization of IF statements. One way to achieve certain degree of parallelism was explained i.e. execute both the 'then' and 'else' parts concurrently even before the execution of the conditional part. Nothing has been described for a loop inside the IF statement or an IF statement inside the loop. These types of blocks are usually found and must be considered for effective parallelization.

The algorithm written is very broad and very small things which are left in that should be taken into account. For example, it was not explained how the different parts will be differentiated and what is the demarkation line between an assignment statement and an arithmetic expression. Even the read and write statements can be executed in parallel in some cases but that was left untouched.

The project will be of any use when the algorithm is converted into the program. This will be making a sort of pre compiler. This is rather a more difficult work and will take quite long time. One more thing which was assumed in the beginning is the number of processors available to be more

than the requirement which is not practical and the limitation on the processors will make the considerable change in the algorithm. Also, we then have to use the scheduling and load balancing techniques for the efficient execution.

The project, it can be concluded, is not just one project, rather it is the combination of a large number of projects. It is too big to expect the completion of even a considerable portion of it in such a short duration. The attempt was made to develop the algorithm and a major portion of it has been done. The things which have been left and which require the attention, have been listed above. If this work is enhanced through the points explained, it will prove to be a major breakthrough in the direction of faster computing.

## REFERENCES

1. Banerjee U., Gajski D. D., "Fast execution of loops with IF statements", IEEE Trans. Comp., C-33, 1984
2. Bernstein A. J., "Analysis of programs for parallel processing", IEEE Trans. Comp. vol 15, 1966
3. Brent R., "Parallel evaluation of general arithmetic expressions", J. ACM 21, 1974
4. Budnik P. and Kuck D., "The organization and use of parallel memories", IEEE Trans. Comp., C-20, 1971
5. Chen S. C. and Kuck D., "Time and parallel processor bounds for linear recurrence systems", IEEE Trans. Comp. C-24, 1975
6. Chen S. C., Kuck D. and Towle R., "Control and data dependence in ordinary programs"
7. Enslow P. H., "Multiprocessors and parallel processing"
8. Fang Z., Yew P. C., Tang P. and Zhu C. Q., "Dynamic processors self scheduling for general parallel nested loops", IEEE Trans. Comp. vol 39, 1990
9. Hellerman H., "Parallel processing of algebraic expressions", IEEE Trans. Comp. vol 15, 1966
10. Jaminsion L. H., "Characteristics of parallel algorithms"
11. Karp R. M., Miller R. E. and Winograd S., "The organization of computations for uniform recurrence equations", J. ACM 14, 1967
12. Kuck D. J., Muroka Y. and Chen S. C., "On the number of operations simultaneously executable in Fortran like programs and their resulting speedup", IEEE Trans. on Computer, C - 21, 1972

13. Kuck D. J., Lawie D. H. and Sameh A., "High speed computers and algorithm organization"
14. Lamport L., " The parallel execution of DO loops", Comm. ACM 14, 1974
15. Li Z., Yew P. C. and Zhu C. Q., "An efficient data dependence analysis for parallelizing compilers", IEEE Trans. Parallel and distributed systems, Jan. 1990
16. Manzo M. D., Frisiani A. L., Olimpo G., "Loop optimization for parallel processing", The Comp. J. vol 22 no. 3.
17. Muller D. E. and Preparata F. P., "Restructuring of arithmetic expressions for parallel execution", J. ACM 23, July 1976
18. Stone H. S., "An efficient parallel algorithm for the solution of tridiagonal linear system of equations", J. ACM 20, 1973
19. Stone, H. S. " High performance computer architecture"
20. Ten H. Tzen and Lion M. Ni, " Dependence uniformization A loop parallelization technique", IEEE Trans. on Parallel and distributed systems, May 1993