

1987

Development of parallel algorithms by balanced divide-and-conquer /

Ghassan Bakdash
Lehigh University

Follow this and additional works at: <https://preserve.lehigh.edu/etd>

 Part of the [Electrical and Computer Engineering Commons](#)

Recommended Citation

Bakdash, Ghassan, "Development of parallel algorithms by balanced divide-and-conquer /" (1987). *Theses and Dissertations*. 4733.
<https://preserve.lehigh.edu/etd/4733>

This Thesis is brought to you for free and open access by Lehigh Preserve. It has been accepted for inclusion in Theses and Dissertations by an authorized administrator of Lehigh Preserve. For more information, please contact preserve@lehigh.edu.

**DEVELOPMENT OF PARALLEL ALGORITHMS
BY BALANCED DIVIDE-AND-CONQUER**

by

GHASSAN BAKDASH

Thesis

Presented to the Graduate Committee

of Lehigh University

in candidacy for the degree of

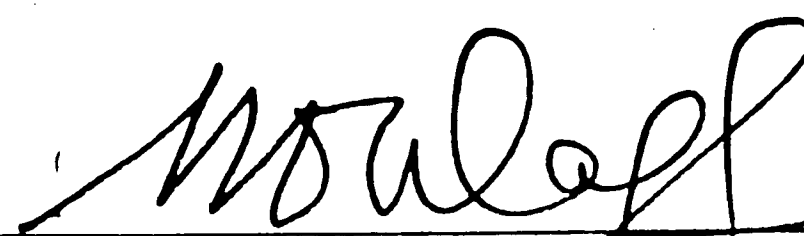
Master of Science in Electrical Engineering

Lehigh University

1987

This thesis is accepted and approved in partial fulfillment of the requirements for the degree of Master of Science in Electrical Engineering.

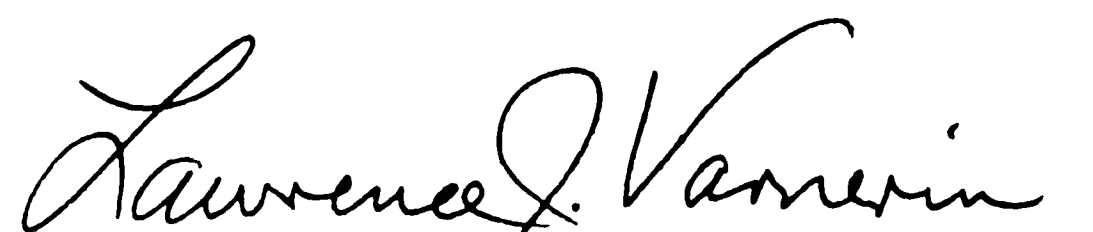
date: Jan 27, 1987



Advisor in charge



EE Division Chairperson



CSEE Department Chairperson

To my parents

OMAIMA

and

OSMAN

ACKNOWLEDGMENT

I am deeply indebted for Professor M.D. Wagh for his guidance and support. In addition, valuable help for producing the manuscript was provided by various fellow students, and I would like to thank all of them.

Table of Contents

1. INTRODUCTION	1
2. RECURSIVE PARTITIONING: MODELING AND ANALYSIS	4
2.1 Introduction	4
2.2 Recursive partitioning of problems with inherent parallelism	7
2.2.1 Preliminaries	8
2.2.2 Recursive doubling	9
2.3 Constant overhead	14
2.3.1 Optimal dynamically recursive partitioning	15
2.3.2 Statically recursive partitioning	28
2.3.3 Partitioning according to a simple function of order	29
2.3.4 Relative performace of different partitiong schemes	30
2.4 Limitations of the chosen model	31
3. EVALUATION OF POLYNOMIALS ON MULTIPROCESSOR ARCHITECTURES	32
3.1 Introduction	32
3.2 Computational models for parallel polynomial evaluation	32
3.3 Statically recursive partitioning	34
3.3.1 Implementation of static recursion for polynomial evaluation	35
3.4 Dynamically recursive partitioning	36
3.4.1 Complexity of the optimal algorithm	38
3.4.2 Number of operations	40
4. MATRIX OPERATIONS ON MULTIPROCESSOR ARCHITECTURE	45
4.1 Introduction	45
4.2 Model for matrix inversion	46
4.3 Analysis of variable overhead models	49
4.4 Optimal partitioning for variable overhead	54
5. HARDWARE IMPLEMENTATION OF PARALLEL ALGORITHMS	59
5.1 Introduction	59
5.2 Polynomial evaluation by statically recursive doubling	61
5.3 Implementation of dynamically recursive partitioning	67
5.4 Bit-Sequential polynomial evaluation	68
5.4.1 A Hardware implementation of Horner's rule	69
6. CONCLUSION	75

List of Figures

- Figure 2-1:** An example showing a problem heirarchy. 10
- Figure 2-2:** Timing involved in decomposing a problem to two problems. 12
- Figure 2-3:** The effect of altering k and λ on a parallel algorithm 20
- Figure 2-4:** The four possible plots for the polynomial $z^{k+\lambda} + z^\lambda - 1$, where $k+\lambda$ and λ can be even or odd. 22
- Figure 5-1:** A direct implementation of Horner's rule to evaluate $P_n = a_n x^n + a_{n-1} x^{n-1} + \dots + a_0$ 63
- Figure 5-2:** A processor which evaluates a polynomial P_ρ using Horner's rule. Notice the input sequence at every initialization of the module. 64
- Figure 5-3:** Direct hardware implementation of static parallel polynomial evaluation. The timing for each module is shown in the 2-tuple written besides each module. The first parameter shows the time when the input to the module must be presented and the second parameter shows the time when the ouput is available. 66
- Figure 5-4:** The layout of an architecture suitable for dyanamically recursive parallel algorithms. 68
- Figure 5-5:** A bit-sequential processor for the computation of first degree polynomials 70
- Figure 5-6:** A pipe containing segments of P_1 modules. The two-tuple numbers shows the time when the first and last bits of different modules are available. The module "TSE" truncates the $2\nu - 1$ bit result to most significant ν bits and sign extends it to $2\nu - 1$ bits. 74

List of Tables

Table 2-1:	A hypothetical parallel algorithm with $k=3$, $\lambda=5$, and $\rho=0$.	25
Table 2-2:	A hypothetical parallel algorithm with $k=3$, $\lambda=5$, and $\rho=1$.	26
Table 3-1:	The characteristics of the optimal dynamic algorithm for polynomials of degrees 1-25.	39
Table 4-1:	The time and optimal r values for parallel triangular matrix inversion.	48
Table 5-1:	A simulation of the module P_1 , with $x=7$, $a=-8$, $b=-8$ and $\nu=4$.	72

Chapter 1

INTRODUCTION

Computer technology has evolved considerably since its onset. However, the need for computer power has been growing at an even more rapid pace. Since the digital hardware technology is expected to mature, achieving further increase in computer performance depends not only on using faster digital devices but also on making radical improvements in computer architecture and processing techniques.

Until recently, virtually all computers were based on the Von-Neumann uniprocessor architecture. All advances in computer architecture were actually elaborations of the basic layout. However, the advent of VLSI technology has made it feasible to deploy several "central processors" in a single computer system. This will eventually lead to the proliferation of multiprocessor computer architecture and "parallel processing". Ideally, one would like to exploit the resources of multiprocessor computers by employing appropriate multiprocessor operating systems and "parallel languages" which alleviate the programmer from explicitly targeting the specifics of the machine architecture. Unfortunately, such tools are quite unlikely to develop in the near future, and therefore, other techniques have to be used to exploit the full potential of multiprocessor systems.

Currently, the only feasible method to efficiently utilize a multiprocessor architecture is to develop special purpose "parallel algorithms" which solve various common computational problems. A parallel algorithm may be created

by recognizing the inherent parallelism of a sequential algorithm. While this technique in some cases is relatively easy, it usually involves considerable restructuring of a computation in order to spread operations across many processors, and thus is usually difficult to apply. Another technique of creating parallel algorithms is problem decomposition by employing the *divide-and-conquer* strategy [1-3]. By partitioning a computational problem into many small and independent problems, a parallel algorithm suitable for multiprocessor architectures may be obtained.

Recursive doubling, a powerful method of generating parallel algorithms, is a special case of divide-and-conquer. The idea is to repeatedly partition each computation into *two* independent parts of equal complexity, which are then computed in parallel. Recursive doubling involves two steps. First, a scheme to *decompose* a problem is developed. Then, the decomposition scheme is applied to *partition* the original problem into two "subproblems" of comparable complexity. Traditionally, the size of both subproblems is arbitrarily chosen to be half the size of the original problem [4, 5]. This heuristically is assumed to generate problems of equal complexity, even though this assumption is, in general, not valid. Partitioning a problem introduces partitioning overhead as well as recombination overhead. These overheads dictate a specific partitioning strategy if optimality is to be achieved.

Thus, for recursive doubling, one needs to develop a systematic approach to guide in the development of parallel algorithms. Clearly, the decomposition algorithm is highly problem dependent, and thus is difficult to develop in a

general context. However, the partitioning strategy can be tackled for a broad range of computational problems.

In this thesis, a computational time-complexity model is developed to describe the performance of several partitioning strategies. This study yields partitioning schemes which give optimal performance, and help estimate the time complexity of the parallel algorithms which are developed using recursive doubling. In addition, the model predicts exactly when partitioning a problem into two equally sized subproblems is optimal. Partitioning types considered in this thesis include constant overhead and variable overhead with a *log* term. These cover a multitude of linear algorithms.

This thesis is organized in six chapters. Chapter 2 introduces the complexity model which describes partitioning, and analyses the model in the case of constant overhead. Chapter 3 illustrates the application of the results obtained in chapter 2 by developing parallel algorithms for the evaluation of polynomials. Chapter 4 presents an analysis of model representing decomposition schemes which generate variable logarithmic overhead, and illustrates a triangular matrix inversion algorithm which conforms to this model. Chapter 5 briefly describe the use of parallel algorithms to design fast processors dedicated towards a particular class of computational problems, and gives several specific examples. The final chapter summarizes the results of this work and suggests possible extensions.

Chapter 2

RECURSIVE PARTITIONING: MODELING AND ANALYSIS

2.1 Introduction

An algorithm is defined as a sequence of instructions which operates on certain input data to give specific results. In general, for reasons concerning efficiency, algorithms must be adjusted to adapt to the architectural features of the computing device on which they will be executed. A broad categorization of computer architecture provides two classes of hardware configuration; a single processor Von-Neumann architecture and architectures utilizing several cooperating processors to achieve either a better throughput or a better time response.

In this thesis, we are mainly concerned with the MIMD architecture. Efficient parallel algorithms useful on such an architecture should take advantage of the multiprocessor environment available to them to solve computational problems in less time. Parallel algorithms break down a particular problem to several smaller independent "subproblems", so that different processors can work on different parts of the problem concurrently.

This chapter deals with the computational models for parallel algorithms suitable mainly for Multiple Instruction Multiple Data stream (MIMD) computers. It also presents computational complexity results of a general nature i.e. results which are applicable to all algorithms fitting the model. The

MIMD architecture is not assumed to conform to a multiprocessor computer, but also includes multicomputers. Since a parallel algorithm is designed to run on a mutiprocessor architecture, its basic design involves partitioning a given computational problem to smaller and independant "subproblems" which can be worked on by a group of processors simultaneously. For the purposes of this thesis, subproblems are called independant if the result of any of them is not required to solve the others.

To quantify the complexity results, one often has to define the "size" of a problem. Let $T_m(n)$ denote the time required to solve a given problem $P(n)$ of size n on an m -processor machine. The ratio of $T_1(n)$ to $T_m(n)$ is called the "speed-up ratio" and is often of great importance. The speed-up ratio is a function of m and n and generally approaches a constant value as the number of processors is increased. Thus the speed up ratio $T_1(n)/T_\infty(n)$ is commonly used as a measure of how good the parallel algorithm performs relative to the best known sequential algorithm. It should be mentioned here that an infinite number of processors -as in $T_\infty(n)$ - does not mean that a parallel algorithm actually uses or requires an infinite number of them but, rather, that as many processors are available as are needed for optimal performance. Most of the results in this chapter are related to the computations of $T_\infty(n)$. However, $T_m(n)$ for finite m is also discussed.

To illustrate the terminology, consider the problem of matching a given number against a list of n numbers using linear search. The "size" of the problem is the size of the list, n . This problem can be divided into m smaller

independent "subproblems" by matching the given number against m lists of $\lceil n/m \rceil$ numbers, such that each of m processors is matching the number against a list of $\lceil n/m \rceil$ numbers. The speedup ratio in this case is $T_1(n)/T_1(\lceil n/m \rceil)$ since $T_m(n)$ is obviously $T_1(\lceil n/m \rceil)$. Notice that this problem can not be divided to more than n elementary problems, which in this case, are indivisible. The elementary problem here is that of matching a number against exactly one number, and is clearly indivisible. Therefore, the speedup ratio increases as m increases until m exceeds n , because n processors are enough to divide the original problem to elementary subproblems. Further increase in m will not introduce further "parallelism" and, therefore, will not improve speed. Thus, $T_\infty(n) = T_1(1)$. The maximum speedup ratio is therefore achieved when $m \geq n$ and is equal $T_1(n)/T_1(1)$. In this example all overhead is neglected for the sake of convenience.

The above example is illustrative but is rather trivial and simplified. The partitioning scheme is straightforward and the overhead associated with the partitioning as well as obtaining the final result is neglected.

Unfortunately, this, in general, can not be neglected without sacrificing performance. Certain operations need to be carried out to combine the results of the subproblems to achieve the final goal; these operations are referred to here as the *recombination overhead*. Other operations may be generated to achieve the partitioning itself; this one is called the *partitioning overhead*. The overhead operations not only increase the number of steps that are required to solve the problem, but may complicate the partitioning scheme. Optimal

performance of a parallel algorithm may be achieved only if a specific partitioning scheme based on this overhead consideration is followed. An example to illustrate this terminology will follow in the next section.

2.2 Recursive partitioning of problems with inherent parallelism

Many computational problems can be decomposed to smaller problems because of their inherent parallel nature. Though the details of the decomposition schemes may vary from one type of problems to another, most of them can still be described as the decomposition of the given problem into two or more smaller independent problems whose solutions can be combined to give the solution of the original problem.

The problem should be partitioned in such a manner that the solution of the problem can be found as quickly as possible using that particular decomposition technique. Notice that the partitioning rule or scheme is *not* the same as the decomposition rule. The decomposition rule takes advantage of an inherent property of a problem to decompose it to several independent subproblems. The partitioning scheme is the rule which assigns the relative sizes of the subproblems using the decomposition rule.

Decomposing a problem directly into a large number of subproblems is a rather difficult task. Therefore, in general, the decomposition is accomplished in many stages. At each stage, a problem received from the preceding stage is further decomposed into a fixed number of smaller problems according to a predetermined rule. This process is continued *recursively* until the subproblems can not be decomposed further. This ensures complete exploitation of

parallelism. Two particular characteristics of the partitioning scheme are of importance: the number of children a parent may have and the sizes of the children subproblems. This chapter deals with the problem of finding these parameters to achieve the best possible performance.

2.2.1 Preliminaries

Let $P(n)$ be a problem of size n whose inherent nature allows it to be decomposed into smaller independent subproblems using some predetermined partitioning rule. If the subproblems of $P(n)$ have the same nature and characteristics as $P(n)$, then they are referred to as the children of $P(n)$. Since these "children" $P(r)$'s, where $r < n$, are similar to $P(n)$, the same partitioning rule can be applied to them to decompose them to even smaller subproblems. Thus, the partitioning may be recursively applied until the problem $P(n)$ is decomposed into many subproblems none of which is larger in size than a certain "elementary" problem.

Thus, recursive partitioning gives a "subproblem hierarchy" which resembles a tree. The original problem is at the root of the tree where the recursion level is zero. The children of $P(n)$ are at level 1 in the subproblem tree, their children are at level 2 and so on. The subproblem which is most removed from the root determines the recursion depth. The leaves of the tree are subproblems which are deemed to be "elementary" by some design criterion. The leaves may be degenerate or trivial forms of the original problem. For example, if $P(n)$ is an evaluation of polynomial of degree n , then the leaves may be polynomials of degree 0 which require no computation. In this case the only computations which are required to solve $P(n)$ are the computations required to

carry on the partitioning and the recombination overheads. In general, however, the leaves may be computational problems in their own right and may require some computation in addition to the computation required by the overhead. It is necessary to emphasize that the elementary problem may be *chosen* to be indivisible. Figure 2-1 shows the hierarchy of a typical problem.

Since the main objective of parallel algorithms is to improve speed by engaging all the available processors, it is important to employ elementary subproblems that are as small as possible. This generates smaller and more numerous leaves to the tree resulting in greater parallelism among smaller units and, consequently, a reduced execution time.

The speed-up ratio of a parallel algorithm depends upon p , the chosen number of "children" of a problem according to the decomposition rule. For convenience, p is chosen to be equal to two throughout most of the following analysis. This is known as *recursive doubling*. The methodology presented, however, is general and can be extended to any p .

2.2.2 Recursive doubling

Using recursive doubling, a computational problem $P(n)$ of size n is partitioned to two subproblems $P(n-r)$ and $P(r-\rho)$, where r is the *partitioning parameter* and ρ is a constant dependent on the nature of P_n . The same rule is applied recursively until all the tree leaves are elementary problems. These leaves are then solved in parallel and their solutions are combined to produce the required solution of the problem at the root.

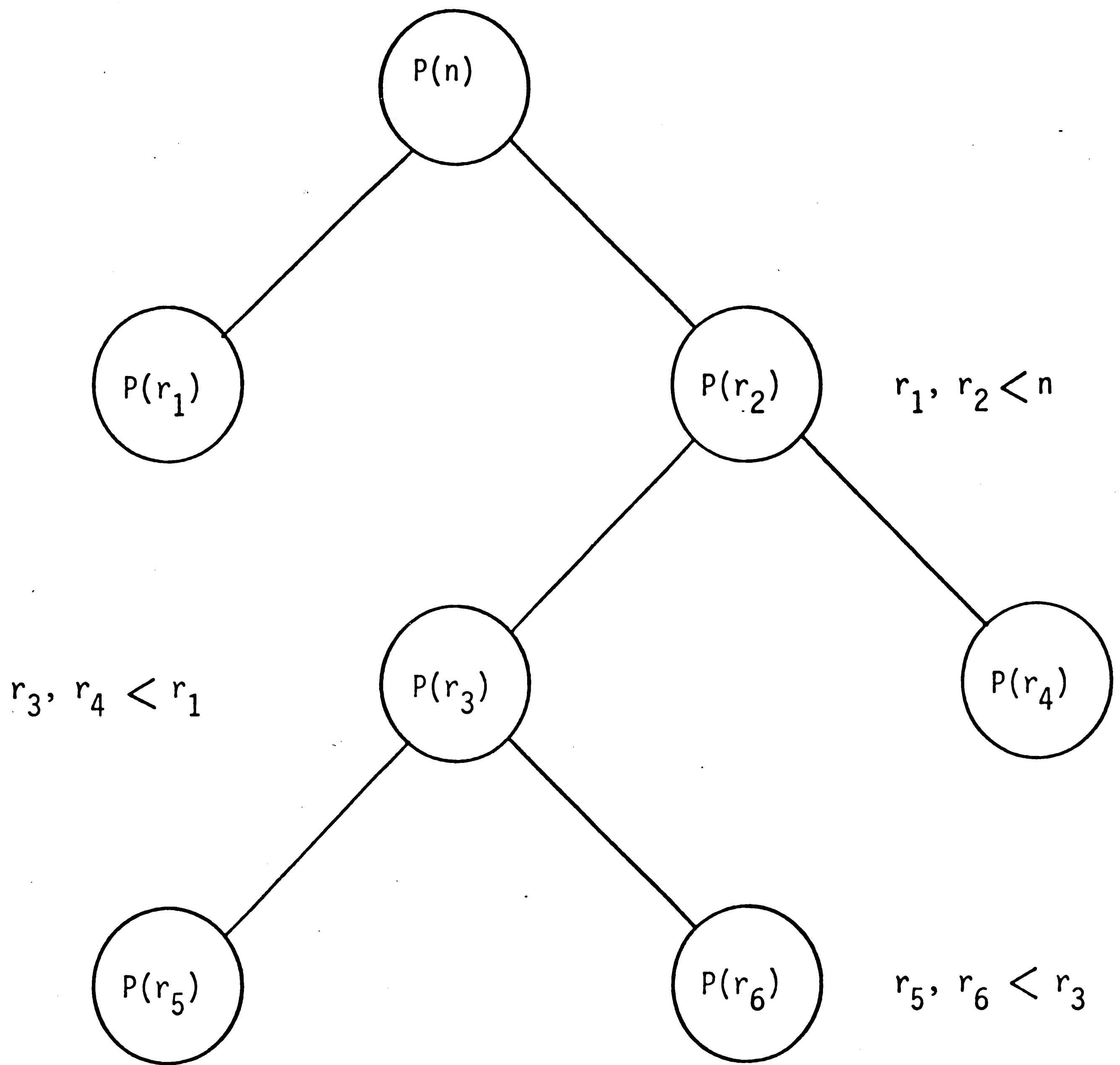


Figure 2-1: An example showing a problem heirarchy.

An important parameter which one may vary is the method of determining the relative sizes of the children. This is equivalent to finding the parameter r

at every node in the tree. If r is fixed throughout all the nodes, one gets what may be called "statically recursive" partitioning. On the other hand, if r is a function of the order of the subproblem at each node, one obtains a "dynamically recursive" partitioning. Dynamic recursion is more difficult to implement than static recursion, however, in general, it yields more optimal results since the parameter r is not determined independently of the problem size.

Let the problem $P(n)$ require exactly $T(n)$ steps to be computed. If the problem is decomposed into two smaller problems $P(n-r)$ and $P(r-\rho)$, the time required to solve $P(n)$ is

$$T(n) = \max\{T(n-r) + k_1, T(r-\rho) + k_2\} + \Lambda,$$

where k_1 , k_2 and Λ are the number of steps required to carry on the partitioning and recombination overheads respectively and are, in general, dependant on n and r .

Without a loss of generality, one may assume that $k_1 \geq k_2$. Then, letting k stand for the smaller of the two k 's and λ for $\Lambda + \max\{k_1, k_2\} - \min\{k_1, k_2\}$, the above expression can be written as

$$T(n) = \max\{T(n-r) + k, T(r-\rho)\} + \lambda. \quad (2.1)$$

From now on, k would be called the *partitioning overhead* and λ the *recombination overhead*.

Typically, the two subproblems $P(n-r)$ and $P(r-\rho)$ would be "solved" by independent sets of processors working in parallel, thus deriving benefit from a multiprocessor environment. The time relationship expressed by equation (2.1)

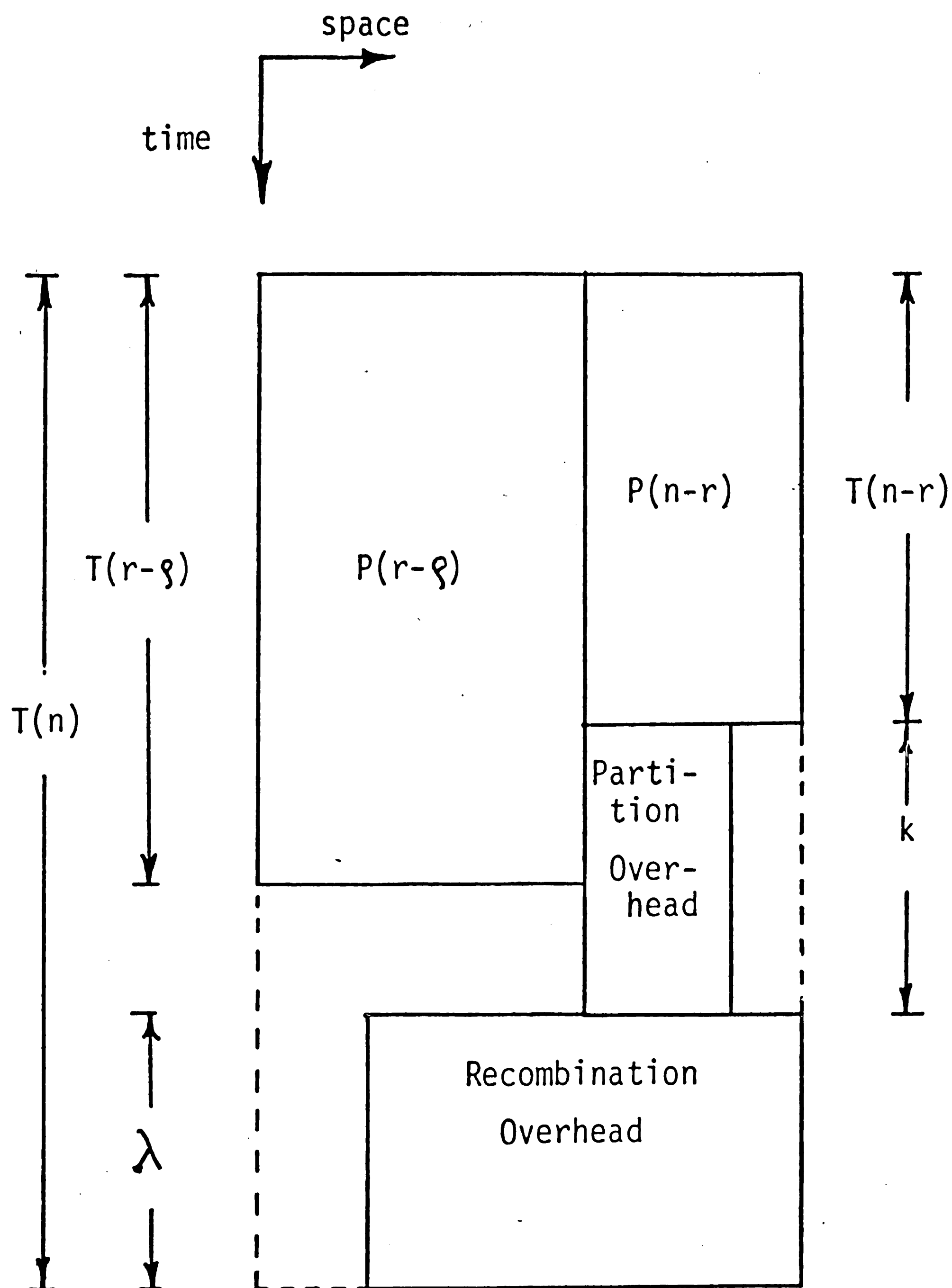


Figure 2-2: Timing involved in decomposing a problem to two problems.

is shown in figure 2-2 which shows the time and space complexity involved in partitioning a problem. The space complexity refers to the number of processors which the algorithm requires. The model above takes only the time complexity

into account and assumes an infinite number of processors. Since in reality the number of processors is limited, a tradeoff between time and space may result [6].

As an example, consider the evaluation of the polynomial P_n of degree n . The polynomial can be expressed as

$$P_n = x^r \cdot P_{n-r} + P_{r-1}.$$

The multiplication between x^r and P_{n-r} is a partitioning overhead and the addition of the two terms is a recombination overhead. If the operations of multiplication and addition are assumed to take the same time, then $k = \lambda = 1$. A group of processors work independently to evaluate both the terms before they can be finally combined (added) to give the final result. Note that the computation of x^r is neglected for reasons which are explained in Chapter 4.

Of particular interest here are the values of r which allow the computation to be completed in as few steps as possible assuming that the subproblems themselves are computed optimally. Another parameter which is important to evaluate or estimate is the number of steps $T(n)$ that are required to completely solve the problem.

To simplify the analysis, we partition the set of problems into distinct classes. A problem $P(n)$ is said to belong to a class C_m if and only if it requires exactly m steps to be solved i.e. if $T(n) = m$. Let $|C_m|$ denote the number of elements (cardinality) of this class. A class C_m is defined to be *empty* if $|C_m|$ is zero. This means that no problem of any order requires

exactly m steps to be completed.

One should be cautioned that the class of a problem is determined by finding the number of steps required to evaluate the problem in the "most general" setting. Degenerate cases of the problem can always be evaluated in a fewer number of steps. For example, the second degree polynomial $ax^2 + bx + c$ can be evaluated in two steps if $a = 1$ and $b = 2\sqrt{c}$; (this is done by computing $x + \sqrt{c}$ and then squaring it). But, as is shown later, a general second degree polynomial requires at least 3 steps and thus for polynomial problems $P(2)$ belongs to C_3 .

In this work, $T(n)$ is assumed to be larger than or equal to $T(n-1)$. This is true because if $T(n)$ is smaller than $T(n-1)$ then $P(n-1)$ can be solved as a degenerate case of $P(n)$ in less time, contradicting our assumption that the computational algorithm is optimal.

2.3 Constant overhead

The expression for $T(n)$ can not be analysed in general. However, some special cases are very important and illustrative. The simplest case arises when the overhead is independent of the size of the problem, i.e. when k and λ are constants. In the next chapter, a fast algorithm to evaluate polynomials is shown to generate constant overhead.

A class C_m contains all the problems which require m steps. The largest problem which belongs to this class is of particular interest. This problem is at the "upper boundary" of class C_m . Let the size of this problem be η_m . Then,

from equation (2.1), one gets

$$T(\eta_m) = \max\{T(\eta_m - r) + k, T(r - \rho)\} + \lambda. \quad (2.2)$$

2.3.1 Optimal dynamically recursive partitioning

As mentioned earlier, the relative sizes of the children at each node, are critically important for good performance of the recursive algorithm. The partitioning parameter r at every partitioning level must be determined according to an appropriate predefined rule. It can be chosen a priori as a constant or as a simple function of the order of every subproblem at every node in the tree. However, in general, this is not the optimal partitioning scheme. Better performance is achieved if for every subproblem of an arbitrary size, r is chosen to give the best possible performance for that specific size of the subproblem.

Intuitively, one may be tempted to assume that such an "optimal r " is close to one half the order of the subproblem. Although this is true in some cases, it is not true in general. For example parallel evaluation of a polynomial is faster for certain values of r which are not close to half the degree of the polynomial; this lack of symmetry is due to overhead as will be shown in the following argument.

The computational load of the parent problem should be equally distributed among all its children for the best possible performance. However, this load includes the overheads and this implies that the problem cannot be partitioned equally. We will now formalize this statement.

The key point in the analysis of equation (2.1) is to find the upper boundary of every class since stepping over these boundaries causes an increase in the number of steps. Note, however, that some classes may be empty and caution must be exercised while defining the upper boundaries of these classes. For reasons which will be clear later, it is convenient to define the upper boundary of an empty class to be the upper boundary of its preceding class. Since the preceding class itself may be empty, the upper boundary of an empty class is numerically equal to the upper boundary of the lower nonempty class which is closest to it with the understanding that it corresponds to a "real" nonempty class only if its boundary is greater than the boundary of its preceding class.

As mentioned earlier, let η_m be the upper boundary of class C_m . $P(\eta_m)$, the largest problem in class C_m , is decomposed into the two subproblems $P(\eta_m - r(\eta_m))$ and $P(r(\eta_m) - \rho)$ where $r(\eta_m)$ represents any value of the parameter r which gives an "optimal" partitioning of the problem $P(\eta_m)$. The following lemma shows that these subproblems are themselves at the upper boundaries of some previous classes.

Lemma 2.1. If $\eta_{\max(m)}$ is at the upper boundary of a class C_m , then $P(r(\eta_m) - \rho)$ is at the upper boundary of a nonempty class $C_{m-\lambda-i}$ for the smallest nonnegative integer i and $P(\eta_m - r(\eta_m))$ is at the upper boundary of a nonempty class $C_{m-k-\lambda-j}$ for the smallest nonnegative integer j . Further, $r(\eta_m)$ is unique and has the value of $\eta_{m-\lambda} + \rho$.

Proof. Assume that $r(\eta_m) - \rho$ is *not* at the upper boundary of the class $C_{m-\lambda}$.

Then $r(\eta_m) - \rho$ and $r(\eta_m) - \rho + t$ belong to the class $C_{m-\lambda}$ or a lower class for some positive t . Then, $r(n)$ can be increased by the quantity t without increasing the total number of steps required to solve P_{η_m} ; this, however, means that η_m in $P(\eta_m - r(\eta_m))$ can also be increased by the same quantity without leaving class C_m , since the increase in η_m will be neutralized by the increase in $r(\eta_m)$ which is a contradiction since η_m is the upper boundary of class C_m by definition. Therefore it is clear that $r(\eta_m)$ is unique since

$$r(\eta_m) - \rho = \eta_{m-\lambda}.$$

$$\therefore r(\eta_m) = \eta_{m-\lambda} + \rho$$

Similarly, $\eta_m - r(\eta_m)$ is at the boundary of class $C_{m-k-\lambda}$ because, otherwise, η_m can be increased without increasing the total number of steps required to solve $P(\eta_m)$ thus creating a contradiction since η_m is the upper boundary of class C_m and can not be increased without leaving class C_m to the next class. *Q.E.D*

Lemma 2.1 is useful to develop a relation between the upper boundary of a class and the upper boundaries of previous classes. This relation, stated in the next theorem, is important for predicting the performance of the algorithm when applied to any problem $P(n)$.

Theorem 2.1. The upper boundary η_m of class C_m is related to the boundaries of previous classes according to the equation

$$\eta_m = \eta_{m-\lambda} + \eta_{m-\lambda-k} + \rho. \quad (2.3)$$

Proof. Recall that problem $P(\eta_m)$ is partitioned into the subproblems $P(\eta_m - r(\eta_m))$ and $P(r(\eta_m) - \rho)$. From lemma 2.1, for optimal results, the value of $r(\eta_m)$ is unique and given by

$$r(\eta_m) = \eta_{m-\lambda} + \rho.$$

Further, from the proof of the same lemma,

$$\eta_m - r(\eta_m) = \eta_{m-k-\lambda}.$$

Eliminations from these two equations give

$$\eta_m = \eta_{m-\lambda} + \eta_{m-k-\lambda} + \rho. \quad Q.E.D$$

The preceding theorem explicitly specifies the class boundaries. We can also estimate the sizes of the classes that may be encountered through the following theorem.

Theorem 2.2. The cardinality of any class C_m can be expressed as

$$|C_m| = |C_{m-\lambda}| + |C_{m-\lambda-k}|. \quad (2.4)$$

Proof. The number of elements in a class C_m is

$$|C_m| = \eta_m - \eta_{m-1}$$

Using equation (2.3), this expression can be rewritten as

$$|C_m| = \eta_{m-\lambda} + \eta_{m-\lambda-k} + \rho - \eta_{m-\lambda-1} - \eta_{m-\lambda-k} - \rho$$

Regrouping the above terms,

$$|C_m| = \eta_{m-\lambda} - \eta_{m-\lambda-1} + \eta_{m-\lambda-k} - \eta_{m-\lambda-k-1}$$

$$\therefore |C_m| = |C_{m-\lambda}| + |C_{m-\lambda-k}|. \quad Q.E.D$$

Expressions (2.3) and (2.4) show the effect of overhead. The overheads k and λ not only increase the number of steps, but play the main role in determining the size of a class relative to its predecessors. Since the class sizes

increase rapidly, an increase in overhead by just one step slows down the algorithm immensely since it may reduce the number of members of a class in a detrimental manner. Since a problem of a high order calls on problems of smaller orders, increasing overhead is "recursively degenerative". Thus, a great improvement in performance may be achieved by decreasing the overhead by a few steps. Figure 2-3 shows the effect of increasing k and λ by 2, for a hypothetical algorithm having $k=3$, $\lambda=5$ and $\rho=0$. As expected, increasing λ is more detrimental than increasing k .

A consequence of equation (2.1) is that the nonempty classes are generated by adding either λ or $k+\lambda$ to some previous nonempty class. Let problem $P(1)$ be the elementary problem requiring $T(1)$ steps to be completed. Notice that $\rho \leq r \leq n-1$. Thus, assuming that $\rho=0$, $P(2)$ can be partitioned only in one way ($r=1$) and it requires $T(2)$ steps given by

$$T(n) = \max\{T(n-1) + k, T(1)\} + \lambda = T(1) + k + \lambda.$$

$T(3)$ can be found using the same method, the only difference being that more values of r are available. Therefore, in general, $T(n)$ can be expressed as

$$T(n) = \min\{\max\{T(n-r) + k, T(r-\rho)\} + \lambda\} \quad r = \rho, \rho+1, \dots, n$$

Thus, $T(n)$ can be expressed as $T(1) + c_1k + c_2\lambda$ where c_1 and c_2 are integers which depend on n and such that $c_1 \leq c_2$.

The above discussion yields following corollary.

Corollary 2.1. If $k = \lambda$, then the nonempty classes are always separated by

k .

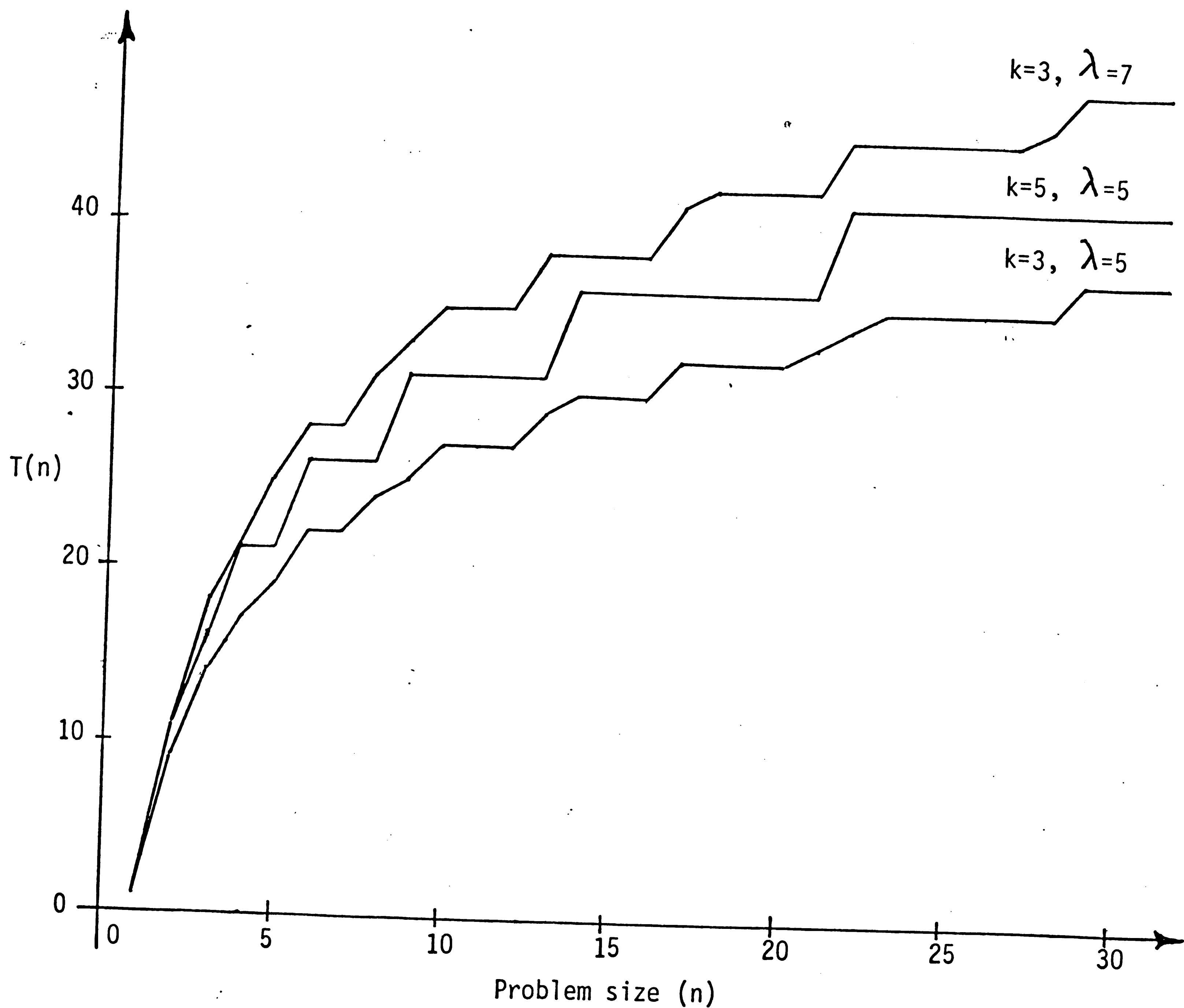


Figure 2-3: The effect of altering k and λ on a parallel algorithm

The asymptotic behaviour of the algorithm, on the other hand, is not determined by the number of steps required to solve the elementary problem, but rather, by the overhead. This is an important observation and it implies that an improvement in the solution of large problems is possible only through

a better partitioning scheme requiring lower overhead.

This asymptotic behavior of the algorithm can also be deduced from equation (2.3) which is a linear difference equation. Let X denote a function of z with η_m as the coefficient of z^m .

Equation (2.3) can then be expressed as

$$X = z^{k+\lambda}X + z^\lambda X + \frac{\rho}{1-z}$$

or

$$(z^{k+\lambda} + z^\lambda - 1)X = \frac{\rho}{z-1},$$

which gives

$$X = \frac{\rho}{(z^{k+\lambda} + z^\lambda - 1)(z-1)}.$$

Using partial fraction expansion for the last equation yields

$$X = \frac{c_1}{1-zz_1} + \frac{c_2}{1-zz_2} + \dots + \frac{c_{k+\lambda}}{1-zz_{k+\lambda}} - \frac{\rho}{1-z},$$

where $c_1, c_2, \dots, c_{k+\lambda}$ are constants which depend upon the initial conditions $T(1), T(2), \dots, T(k+\lambda)$ which themselves are determined from $T(1), k$ and λ ; $z_1, z_2, \dots, z_{k+\lambda}$ are the roots of the equation

$$z^{k+\lambda} + z^\lambda - 1 = 0. \tag{2.5}$$

The solution to the difference equation is

$$\eta_m = c_1 z_1^m + c_2 z_2^m + \dots + c_{k+\lambda} z_{k+\lambda}^{k+\lambda} + \rho.$$

Figure 2-4 shows the four possible plots of equation (2.5). It can be seen that regardless of k and λ , equation (2.5) has exactly one positive root.

In [7], it is proven that all the roots of a polynomial having only one

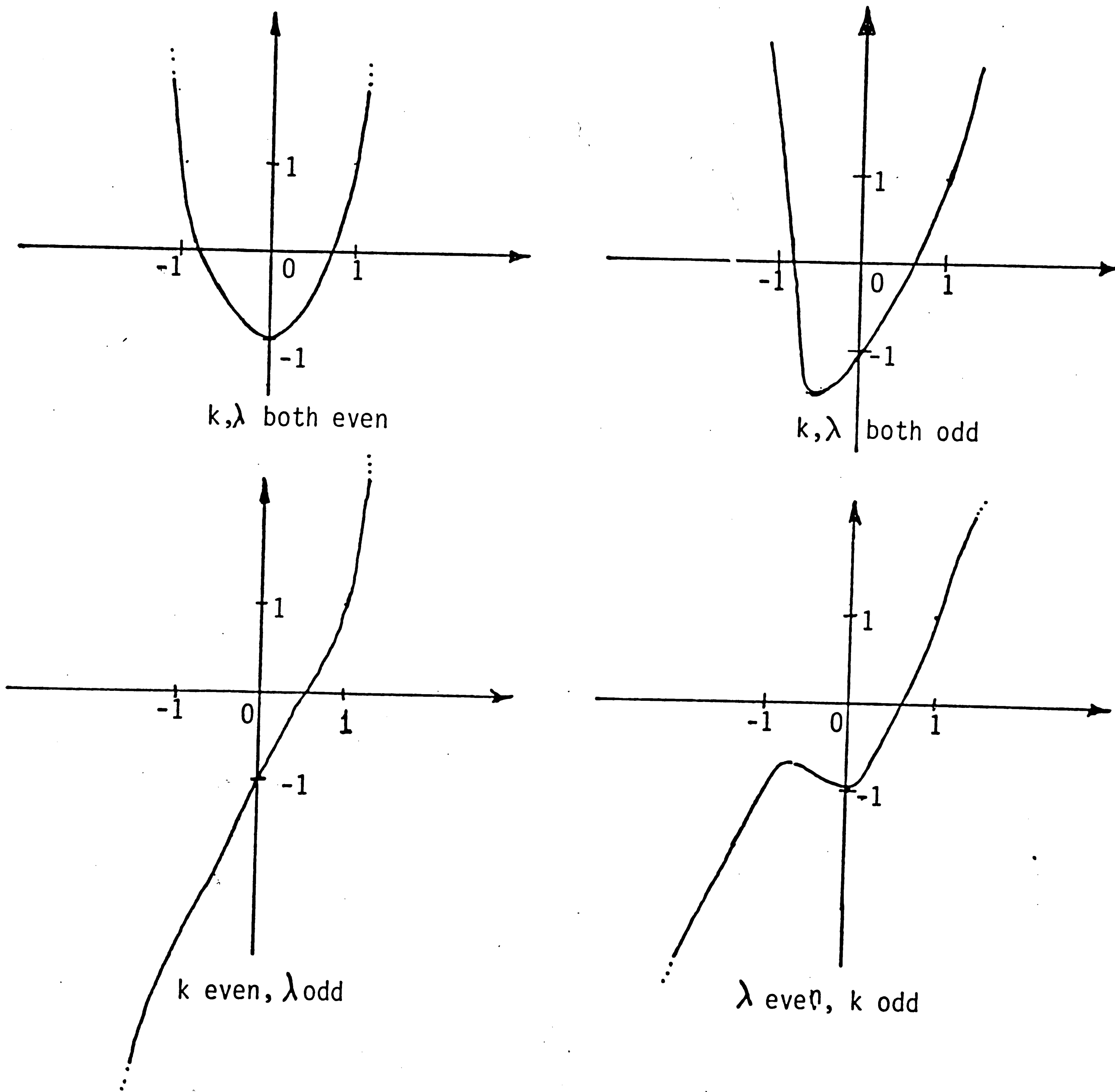


Figure 2-4: The four possible plots for the polynomial $z^{k+\lambda} + z^\lambda - 1$, where $k+\lambda$ and λ can be even or odd.

positive root lie in the circle $|z| \leq r$, where r is that positive root. Further, all roots lie in the circle $1 + \max\{|a_i/a_{k+\lambda}|\}$. Since the maximum ratio of coefficients in equation (2.5) is unity, all the roots must lie in the circle $|z| \leq 2$. Hence $\eta_m \leq 2^m$ as m approaches infinity.

We now determine the optimal values of r which would yield the best possible (fastest) algorithm in the dynamic recursion under consideration.

Let $r(n)$ represent a value of r which gives the minimum number of steps to completely solve a problem $P(n)$ (assuming that the subproblems themselves have been completed optimally).

Theorem 2.3. For any member n in class C_m , $r(n)$ must satisfy the inequality

$$n - \eta_{m-\lambda-k} \leq r(n) \leq \eta_{m-\lambda} + \rho. \quad (2.6)$$

Proof. Since $n \in C_m$, $n = \eta_m - h$ for some nonnegative h . $T(n)$ may then be expressed as

$$m = T(\eta_m - h) = \max\{T(\eta_m - h - r) + k, T(r - \rho)\} + \lambda,$$

Thus,

$$m \geq T(\eta_m - h - r) + k + \lambda \quad (2.7)$$

and

$$m \geq T(r - \rho) + \lambda. \quad (2.8)$$

From equation (2.7),

$$T(\eta_m - h) \leq m - k - \lambda, \text{ or}$$

$$\eta_m - h - r \leq \eta_{m-\lambda-k}$$

Substituting $\eta_m - n$ for h , we get

$$r(n) \geq n - \eta_{m-\lambda-k}$$

Similarly, from equation (2.8),

$$T(r-\rho) \leq m - \lambda, \text{ or}$$

$$r - \rho \leq \eta_{m-\lambda}$$

Rewriting this condition gives

$$r(n) \leq \eta_{m-\lambda} + \rho.$$

Combining the two constraints on $r(n)$ completes the proof.

Q.E.D.

Table 2-1 shows the performance of a parallel algorithm whose recombination overhead is 5, partitioning overhead is 3, $\rho = 0$ and $T(1) = 1$. Table 2-2 shows the performance of the same algorithm with $\rho = 1$. Notice that increasing ρ improves the performance of the algorithm by increasing the upper boundaries of the nonempty classes but no new nonempty classes are generated.

Corollary 2.2. The problem $P(n)$ with a constant λ can be decomposed optimally with $r = (n+\rho)/2$ if and only if $k = 0$.

Proof. Let n belong to class C_m . If $k=0$, then from theorem 2.1

$$\eta_m = 2\eta_{m-\lambda} + \rho. \tag{2.9}$$

Theorem 2.3, for this k , specifies the range of r as

$$n - \eta_{m-\lambda} \leq r \leq \eta_{m-\lambda} + \rho.$$

Using equation (2.9) in the last equation

Table 2-1: A hypothetical parallel algorithm with $k = 3$, $\lambda = 5$, and $\rho = 0$.

n	T(n)	minimum r	maximum r
1	1	-	-
2	9	1	1
3	14	2	2
4	17	2	2
5	19	3	3
6	22	3	4
7	22	4	4
8	24	5	5
9	25	5	5
10	27	5	7
11	27	6	7
12	27	7	7
13	29	8	8
14	30	7	9
15	30	8	9
16	30	9	9
17	32	9	12
18	32	10	12
19	32	11	12
20	32	12	12
21	33	12	12
22	34	13	13
23	35	11	16
24	35	12	16
25	35	13	16
26	35	14	16
27	35	15	16
28	35	16	16
29	37	16	20
30	37	17	20
31	37	18	20
32	37	19	20
33	37	20	20
34	38	18	21

Table 2-2: A hypothetical parallel algorithm with $k = 3$, $\lambda = 5$, and $\rho = 1$.

n	number of step	minimum r	maximum r
1	1	-	-
2	9	1	1
3	9	2	2
4	14	3	3
5	14	4	4
6	17	3	4
7	17	4	4
8	19	5	6
9	19	6	6
10	22	5	8
11	22	6	8
12	22	7	8
13	22	8	8
14	24	9	10
15	24	10	10
16	25	9	10
17	25	10	10
18	27	9	14
19	27	10	14
20	27	11	14
21	27	12	14
22	27	13	14
23	27	14	14
24	29	15	16
25	29	16	16
26	30	13	18
27	30	14	18
28	30	15	18
29	30	16	18
30	30	17	18
31	30	18	18
32	32	17	24
33	32	18	24
34	32	19	24

$$n - \frac{\eta_m - \rho}{2} \leq r \leq \frac{\eta_m + \rho}{2}. \quad (2.10)$$

It is easy to verify that $r = \frac{n + \rho}{2}$ does indeed lie within this range of optimal r .

To prove the second (necessary) part of the corollary, note that from theorem 2.3, for any n in class C_m ,

$$n - \eta_{m-\lambda} \leq r(n).$$

Substituting $(n + \rho)/2$ for $r(n)$ at $n = \eta_m$, gives

$$\eta_m \leq 2\eta_{m-\lambda-k} + \rho.$$

which, using theorem 1 gives

$$\eta_{m-\lambda} + \eta_{m-\lambda-k} + \rho \leq 2\eta_{m-\lambda-k} + \rho.$$

or

$$\eta_{m-\lambda} \leq \eta_{m-\lambda-k}.$$

Clearly, this can be true only if $k = 0$.

Q.E.D.

Corollary 2.2 is important to determine the cases when the optimal r can be chosen rather easily from n . It should be mentioned here that generally in "divide and conquer" strategies, r is taken as $n/2$. But from the above corollary, such r 's are optimal only if both k and ρ are zero.

2.3.2 Statically recursive partitioning

A special case occurs when the partitioning parameter, r , is fixed throughout all the recursive partitioning levels. The scheme is not optimal but its simplicity may be advantageous under some circumstances. In particular, if only a few processors are available, dynamically recursive partitioning does not necessarily perform better or at least not significantly better. Statically recursive partitioning has a modular and predictable structure which is particularly useful for direct hardware implementation.

Static recursion proceeds as follows. The elementary problem size, L , is chosen prior to any partitioning. Then at each stage, r is chosen such that the problem $P(n)$ is recursively partitioned into two subproblems $P(n-r)$ and $P(r-\rho)$, with $L = r-\rho$. The latter is not be partitioned further and is the basic building block of the algorithm. $P(n-r)$, however, is partitioned further (if $n-r > L$) using the same procedure.

An analysis of this procedure gives following result.

Theorem 2.4. For static recursion with elementary problem of size L ,

$$\eta_m = \eta_{m-\lambda-k} + L + \rho.$$

Proof. From equation (2.1) one gets,

$$T(\eta_m) = \max\{T(\eta_m - L - \rho) + k, T(L)\} + \lambda.$$

The first term in the above braces is greater than or equal to the second term.

Thus,

$$T(\eta_m) = T(\eta_m - L - \rho) + k + \lambda \quad (2.11)$$

Since $T(\eta_m) = m$ (by definition), then $T(\eta_m - L - \rho) = m - k - \lambda$

and thus $\eta_m - L - \rho$ belongs to class $C_{m-k-\lambda}$.

Moreover, it is at the upper boundary of this class because, otherwise

$$T(\eta_m - L - \rho) = T(\eta_m - L - \rho - 1)$$

which gives

$$T(\eta_m + 1) = T(\eta_m + 1 - L - \rho) + k + \lambda.$$

This leads to

$$T(\eta_m + 1) = T(\eta_m - L - \rho) + k + \lambda = T(\eta_m),$$

which contradicts the fact that η_m is at the upper boundary of class C_m . Thus,

$$\eta_m = \eta_{m-\lambda-k} + L + \rho. \quad \text{Q.E.D.}$$

Since L is typically a small number, it is clear from a comparison of theorems 2.1 and 2.4 that dynamic recursion performs much better especially for large problems if many processors are available to take full advantage of existing parallelism.

2.3.3 Partitioning according to a simple function of order

A partitioning scheme which falls between static and dynamic recursive doubling is to choose r as a simple function of the size of the subproblem under consideration. A good guess is to choose r to be half the value of n . For many types of problems, this value may in fact belong to the range of optimal $r(n)$ found in section 2.3.1.

To analyse this partitioning scheme, substitute $\lfloor n/2 \rfloor$ for r in equation (2.1) to get

$$T(\eta_m) = \max\{T(\eta_m - \lfloor \eta_m/2 \rfloor) + k, T(\lfloor \eta_m/2 \rfloor - \rho)\} + \lambda.$$

Now, since $\eta_m - \lfloor \eta_m/2 \rfloor$ equals $\lfloor \eta_m/2 \rfloor$,

$$T(\eta_m) = \max\{T(\lfloor \eta_m/2 \rfloor) + k, T(\lfloor \eta_m/2 \rfloor - \rho)\} + \lambda. \quad (2.12)$$

For all nonnegative values of ρ , the first term in the braces of Equation (2.12) is larger than the second term. Thus, equation (2.12) reduces to

$$T(\eta_m) = T(\lfloor \eta_m/2 \rfloor) + k + \lambda.$$

The above equation can be further simplified by observing that η_m is even because otherwise $T(\eta_m) = T(\eta_m + 1)$ which is a contradiction. This gives

$$T(\eta_m) = T(\eta_m/2) + k + \lambda. \quad (2.13)$$

Since $T(\eta_m) = m$, equation (2.13) shows that $\eta_m/2$ belong to class $C_{m-k-\lambda}$.

Further, it should be the maximum of that class because otherwise,

$$T(\eta_{m+2}) = T(\eta_{m/2} + 1) + k + \lambda = T(\eta_{m/2}) + k + \lambda = T(\eta_m),$$

showing that η_m is not at the upper boundary of class C_m which is a contradiction to its definition.

$$\text{Thus, } \eta_m = 2\eta_{m-k-\lambda}. \quad (2.14)$$

2.3.4 Relative performance of different partitioning schemes

Statically recursive partitioning is the simplest and the least adaptive among the schemes reviewed above. It uses a fixed value of r throughout the partitioning tree regardless of the sizes of problems at different stages. If, on the other hand, finding the value of r at a particular stage in the tree takes into consideration the size of the subproblem at that stage, then the recursion is dynamic. Dynamic recursion partitioning can be done in several ways each of which may give a different performance level. In general, performance improves as r approaches the optimal $r(n)$ at every stage in the partition scheme. Thus,

the worst performance is achieved by static recursion and the best by optimal dynamic recursion. The choice of $r = \lfloor n/2 \rfloor$ gives an average performance but is sometimes advantageous since the choice of r may be made fairly quickly.

2.4 Limitations of the chosen model

Solving any computational problem involves using operations which manipulate operands. Usually, more than one operation is necessary to solve any problem. For example, evaluating a polynomial requires at least addition and multiplication; problems involving matrix inversion require addition, multiplication as well as division. Primary and secondary memory access times are neglected in complexity analysis but may also have to be included. This thesis assumes that all operations require the same time, although it is often quite unrealistic. Multiplication usually takes at least twice as much time as addition and in searching and sorting problems, access to RAM is much faster than access to bulk memory.

The difference in the time requirements of different operations often make a drastic change in the partitioning scheme if optimality is to be maintained. In the model developed above, different operation time requirements can be accommodated by choosing a basic unit of time (generally the greatest common divisor of all the operation times) and then expressing each operation time as its multiple. This affects the partitioning scheme only by changing the values of the parameters k and λ . This approach simple modification of the basic model can provide accurate results in a variety of realistic problems.

Chapter 3

EVALUATION OF POLYNOMIALS ON MULTIPROCESSOR ARCHITECTURES

3.1 Introduction

Polynomial evaluations arise in applications requiring the computation of transcendental functions and interpolating polynomials. Parallel evaluation of polynomials is a classical problem which has been tackled since the introduction of the idea of multiprocessing [4, 9-13]. The evaluation of polynomials has been investigated thoroughly because it is a typical problem which is inherently structured so that extensive parallelism is available. Some existing vector processors such as the IBM 3838 attached back-end processor include polynomial evaluation as one of their functions [8].

3.2 Computational models for parallel polynomial evaluation

In this work, a polynomial of degree n is designated as P_n . To compute P_n in optimal or near optimal time using a multiprocessor architecture, P_n is split into many smaller polynomials which may be evaluated in parallel using several processors. We start by splitting P_n into two polynomials as

$$P_n = x^r \cdot P_{n-r} + P_{r-1} \quad (3.1)$$

Expression (3.1) shows that P_n may be decomposed into two independent polynomials which can be computed in parallel. The term x^r must also be computed in parallel to with P_{r-1} and P_{n-r} . The two polynomials on the right hand side of equation (3.1) can in turn be decomposed into smaller polynomials by applying the same or a similar rule. The decomposition is thus recursive in nature as was explained in chapter 3. Notice that the polynomial could have

been split to three polynomials rather than two with minor modifications to the model.

Let $T(n)$ denote the minimum number of steps (and thus the minimum time measured in steps) required to compute P_n using a parallel algorithm with a specific partitioning scheme.

As mentioned in chapter 3, the decomposition model, expressed in equation (3.1), can be used to partition the polynomial problem in several ways by choosing different methods of specifying r at each recursion level. The simplest method of static recursion fixes r a priori to some constant throughout all the recursion levels. The optimal dynamic recursion method requires to choose at each node in the recursion tree an r which optimizes the completion of the subproblem associated with that specific node, i.e. an r which optimizes the evaluation of the polynomial with the specific degree at that node. Many other methods of choosing r are conceivable and feasible, but in this work we mainly concentrate on the two methods mentioned above. In general, optimal dynamic recursion has the best performance, especially on a MIMD computer. However, some other method may be more suitable and possibly even faster for a dedicated hardware architecture.

3.3 Statically recursive partitioning

In this simple method, radix r is chosen to be a constant throughout the polynomial evaluation process, leading to the computational process expressed as

$$P_n = x^r(x^r(x^r(\dots(x^r.P_{n-kr} + P_{r-1}) + P_{r-1}) + P_{r-1}) \dots + P_{r-1}). \quad (3.2)$$

Horner's rule is seen as a special case of static recursion with $r=1$. Each step in this evaluation is:

$$P_n = x^r.P_{n-r} + P_{r-1},$$

where P_{r-1} collects all the P_n terms having a degree less than r , and P_{n-r} collects all the remaining terms.

Since the degree of P_{r-1} is less than r , P_{r-1} can not be decomposed further. But, P_{n-r} can in general be decomposed further by applying the method recursively until all the degrees involved are less than r . When polynomial P_n is completely decomposed, it has the form expressed in equation (3.2).

The recursion depth, k , is the largest integer such that $n \geq kr$. The $r+1$ degree polynomials can be evaluated using any available method. The simplest method is to calculate them by direct evaluation. In this case, one may take advantage of the fact that the same powers of x are used in the calculation of each of the $r-1$ degree polynomials, and therefore they may be precomputed and then used repeatedly.

3.3.1 Implementation of static recursion for polynomial evaluation

In static recursion, r is fixed so the polynomial P_n is evaluated essentially by computing many P_{r-1} polynomials. As mentioned earlier, these polynomials may be evaluated using any satisfactory method. It will be shown later in chapter 5 that Horner's rule may be used to evaluate these polynomials in a fashion that lays out a pipelined architecture to compute polynomials using statically recursive partitioning. Horner's rule is the most efficient and quickest method to compute a polynomial on a single processor Von Neuman Architecture. But in this case the powers of x are calculated once and used for all the P_{r-1} polynomials, so Horner's rule requires almost as many operations as the direct evaluation where each power of x is multiplied by its corresponding coefficient, possibly in parallel. In contrast, Horner's rule can not be parallelized but may be chosen to reduce the hardware complexity in case of implementation through dedicated hardware. On a MIMD computer, evaluating the P_{r-1} polynomials using direct evaluation leads to a superior performance with only a small penalty in terms of additional overhead to compute the powers of x .

This section presents an implementation of polynomial evaluation using static recursive partitioning with direct evaluation of P_{r-1} polynomials. It is assumed that the multiplication of an x power and its corresponding coefficients is done in parallel with other operations.

Since the powers of x involved in the calculation of each P_{r-1} polynomial are precalculated, each one of the P_{r-1} polynomials requires exactly $r-1$

multiplications and $r-1$ additions and may be done in r time slots if an infinite number of processors are available. The k P_{r-1} polynomials in equation (3.2) contribute $k(r-1)$ multiplications and the polynomial P_{n-kr} requires $n-kr$ multiplications. At each level there is also a multiplication with x^r giving k more multiplications. Finally, the powers of x which are calculated prior to completing any level require $r-1$ multiplications. Thus, the total number of multiplications is $n-kr+k(r-1)+k+r-1 = n+r-1$. By a similar argument, the total number of additions is found to be n . Thus this implementation implies an additional $r-1$ multiplications over Horner's rule which require only n multiplications.

The simplicity of this algorithm makes it particularly suitable for direct hardware implementation, even though it does not fully exploit the potential parallelism available in polynomial evaluation.

3.4 Dynamically recursive partitioning

If at each stage of computation, the problem is split optimally, the overall problem is solved in the minimum number of steps achievable on such a model. This section shows that polynomials can be partitioned dynamically according to the model presented in chapter 3.

Note that in equation (3.1), the multiplication between P_{n-r} and x^r is a partitioning overhead and the addition of the two terms is a recombination overhead. Both these overheads have the value of one since we consider that all the operations take the same execution time. Thus, one gets the computational model

$$T[P_n] = \max\{\max\{T[x^r], T[P_{n-r}]\} + 1, T[P_{r-1}]\} + 1. \quad (3.3)$$

This model does not necessarily follow the dynamically recursive model developed in chapter 3 since for the optimal values of r , the term x^r may require more steps than could be tolerated. To prove that the two models are equivalent, we first prove that at least for one value of optimal r , x^r should not be the dominating term in the \max expression. The following lemma prove that this x^r caculation can never be a bottleneck when partitioning dynamically.

$$\text{Lemma 3.1. } \mathcal{L}[x^r] \leq T[P_{r-1}] - 1.$$

Proof. Choose a particular $r-1$ degree polynomial $P_{r-1} = a_{r-1}x^{r-1} + 1$. If P_{r-1} requires m steps, then calculating the value $a_{r-1}x^{r-1}$ requires at most $m-1$ steps. For a particular $a_{r-1} = x$, however, this expression is x^r . Therefore, computing x^r requires one less step than computing P_{r-1} . *Q.E.D.*

The next theorem proves that the dynamic model of chapter 3 is a valid model for the dynamically recursive partitioning of polynomials.

Theorem 3.1. If one uses the partitioning shown in equation (3.1) with optimal r , then

$$T[P_n] = \max\{T[P_{n-r}], T[P_{r-1}]\} + 1.$$

Proof. The total number of steps, $T[P_n]$, required to compute P_n according to equation (3.3) is

$$\max\{\max\{T[x^r] + 1, T[P_{n-r}] + 1\}, T[P_{r-1}]\} + 1.$$

From lemma 3.1, it is known that $T[x^r] + 1$ is at most equal to $T[P_{r-1}]$. Thus,

either of these two terms can be removed without changing the outcome of the *max* function. Thus, $T[P_n] = \max\{T[P_n] + 1, T[P_{r-1}]\} + 1$. Q.E.D.

Since theorem 3.1 proves that the constant overhead model of chapter 2 is applicable in this case, the results developed there are directly applicable in the present case. The following sections describe the characteristics of the algorithm.

3.4.1 Complexity of the optimal algorithm

As mentioned earlier, partitioning a polynomial into two polynomials introduces two overhead operations for each level of partitioning. The partitioning overhead is a multiplication with x^r and the recombination overhead is an addition of two terms. Thus, both λ and k in section 2.3.1 are equal to unity. Notice also that ρ is unity. Therefore, all polynomials will be classified according to their degrees. Let class C_m contain all the polynomial degrees which require m steps to be evaluated in a multiprocessor environment. Let η_m designate the largest member of C_m . Equation (2.3) in this case gives

$$\eta_m = \eta_{m-1} + \eta_{m-2} + 1 \quad (3.4)$$

Polynomial P_n will be partitioned to many polynomials of degree 0 by optimal partitioning. But a polynomial of degree 0 is actually a coefficient of the original polynomial and thus the leaves in the recursion tree correspond to the coefficients of the polynomial P_n .

The optimal number of computational steps required to compute polynomials of degrees less than 26 are shown in table 3-1.

Table 3-1: The characteristics of the optimal dynamic algorithm for polynomials of degrees 1-25.

problem size	number of steps	minimum r	maximum r
1	2	1	1
2	3	2	2
3	4	2	3
4	4	3	3
5	5	3	5
6	5	4	5
7	5	5	5
8	6	4	8
9	6	5	8
10	6	6	8
11	6	7	8
12	6	8	8
13	7	6	13
14	7	7	13
15	7	8	13
16	7	9	13
17	7	10	13
18	7	11	13
19	7	12	13
20	7	13	13
21	8	9	21
22	8	10	21
23	8	11	21
24	8	12	21
25	8	13	21

The number of processors is assumed to be large enough so as not to be a constraint. Shown also in table 3-1 are the values of optimal r . The range of optimal r 's is contiguous as predicted by theorem 2.3. Notice that all classes are nonempty and that $|C_m|$ is a Fibonacci number (1, 2, 3, 5, 8, 13, etc). Moreover, the smallest value of n in any class C_m is also a Fibonacci number and is designated by F_m . That F_m is a Fibonacci number is not coincidental, rather, (3.4) mimics the well known property that a Fibonacci number is

generated by adding the two preceding Fibonacci numbers.

Applying theorem 2.3 gives the following range of optimal r .

$$n - \eta_{m-2} \leq r \leq \eta_{m-1} + 1, \quad \text{where } n \in C_m. \quad (3.5)$$

3.4.2 Number of operations

Any parallel algorithm works by partitioning the original computational problem into smaller and independent subproblems which can be solved in parallel. However, this can be done only at the expense of overhead which introduces some new operations which otherwise would have been unnecessary. Thus, the time savings are achieved at the cost of a decreased efficiency. Dynamically recursive partitioning introduces additional operations over the optimal serial evaluation of polynomials by Horner's rule which requires exactly n additions and n multiplications to evaluate a polynomial P_n of order n . To assess the efficiency of a parallel computation, we now compare the number of operations required by it with the number of operations required by Horner's rule.

The following theorem determines the total number of operations for the dynamic optimal recursion excluding the operations necessary to compute the required powers of x .

Theorem 3.2. Computing a polynomial P_n by optimal dynamically recursive partitioning requires n additions and n multiplications if the required powers of x are precomputed.

Proof. At every node in the recursion tree where (3.4) is applied, two

operations, a multiplication and an addition, are encountered. The sum of the degrees of the children polynomials is exactly one less than the degree of the parent polynomial. (This is obvious from (3.4). The degrees of the polynomial on the right are $n-r$ and $r-1$, respectively and that, on the left is $n-1$). Since all the polynomials at the leaves of the recursion tree have a degree of 0, it is clear that equation (3.1) must have been applied n times. Thus, if the polynomial is partitioned completely, n additions and n multiplications are generated. Q.E.D.

Thus, the number of operations resulting from overhead operations equals those required by Horner's rule. The only extra computations which the parallel evaluation requires are the multiplications required to compute the necessary powers of x . The powers of x which are generated for a particular polynomial depend on the values of r that are chosen at different stages throughout the recursion. To identify these powers, the values of r must be chosen according to a consistent rule such as using either the minimum or the maximum value of the range specified by equation (3.5) at *each node in the recursion tree*. As partitioning proceeds, new powers of x are generated as required. In general, each new power of x may require several multiplications. But as the following two theorems show, if throughout the recursion stages r is *consistently* chosen to be either the minimum or the maximum value of the range of r , then each new power is related rather simply to the powers already generated.

Lemma 3.2. Let η_m denotes the maximum of class C_m . A polynomial

P_{η_m} decomposes into the polynomials $P_{\eta_1}, P_{\eta_2}, P_{\eta_3}, \dots, P_{\eta_{m-2}}, P_{\eta_{m-1}}$ such that $\eta_1, \eta_2, \dots, \eta_{m-1}$ are the maximums of *all* the classes which precede the class C_m . Further, all the x powers used in the computation of P_{η_m} are the Fibonacci numbers $1, 2, 3, 5, \dots, F_m$.

Proof. As shown by (3.5), the value of r used to partition P_{η_m} is unique such that

$$r = \eta_m - \eta_{m-2}.$$

Using equation (3.4), we get

$$r = \eta_{m-1} + 1.$$

Notice that the first polynomial generated in this partitioning has a degree $\eta_m - r$.

But $\eta_m - r = \eta_m - \eta_{m-1} - 1 = \eta_{m-2}$.

The other generated polynomial has a degree of $r - 1$ which clearly equals η_{m-1} .

Finally, since η_{m-1} is one less than the Fibonacci number F_m and since $r = \eta_{m-1} + 1$, the value of r is a Fibonacci number F_m .

Thus, partitioning the polynomial P_{η_m} generates two polynomials whose sizes η_{m-1} and η_{m-2} and uses F_m 'th power of x . Recursive use of this argument proves the lemma. Q.E.D.

Theorem 3.3. If at each node in the recursion tree the minimum optimal r is used, the evaluation of polynomial $P_n \in C_m$ requires only powers $1, 2, \dots, F_{m-2}$ (i.e. all Fibonacci numbers 1 through F_{m-2}) of x . Further, a new power of x is always a product of two powers of x which have been already computed and therefore it requires exactly one more multiplication to be calculated.

Proof. Splitting P_n gives two polynomials P_{n-r} and P_{r-1} . Notice that if r is chosen at its minimum value of $n - \eta_{m-2}$, then $n - r$ equals η_{m-2} . Therefore by lemma 3.2, its evaluation requires those powers of x which are Fibonacci numbers 1 through F_{m-2} .

Since $r - 1 = n - \eta_{m-2} - 1$, it can be easily shown that $r - 1 \in C_m$ or $r - 1 \in C_{m-1}$. Thus, P_{r-1} will be split into two polynomials one of which is of degree η_{m-3} or η_{m-4} as per the above argument. Lemma 3.2 ensures that these polynomials will not introduce any new power of x since they require "Fibonacci powers" which have already been generated before. Therefore, the only possible new powers of x can be generated in the "rightmost" branch of the recursion tree. In case $(r - 1) \in C_{m-1}$, P_{r-1} is split according to an r_2 such that

$r_2 = (r - 1) - \eta_{m-3} = r - F_{m-2}$. But power F_{m-2} was generated earlier and therefore, x^r is obtained by multiplying this power with x^{r_2} . If $(r - 1) \in C_m$, then a similar argument shows that

$$r_2 = r - 1 - \eta_{m-3} + 1 = r - F_{m-3}.$$

This again shows that x^r is obtained through one multiplication.

Therefore, any power which is generated at any node in the recursion tree can be calculated through only one multiplication and using two powers which have already been generated before at two other nodes. Q.E.D.

Theorem 3.4. If at each node in the recursion tree the largest optimal r is used, evaluation of a polynomial $P_n \in C_m$ requires only the powers 1, 2, 3, 5, ..., F_{m-1} , F_m .

Proof. As shown in equation (3.5), the largest value of r for any $n \in C_m$ is

$\eta_{m-1} + 1$, which is the Fibonacci number F_m . Hence the theorem. Q.E.D.

The above two theorems combined with the operation count for the overhead operations provide a good estimate of the amount of operations required to complete polynomial evaluation on multiprocessor architectures using optimal recursion. If $n \in C_m$ ($F_m \leq n \leq F_{m+1}$), then evaluation of P_n requires n additions and $n + m - 1$ multiplications. Moreover, the partitioning scheme can take advantage of the multiprocessor and complete all the required operations in only $m = O(c \log(n))$ steps, where c is a constant.

Chapter 4

MATRIX OPERATIONS ON MULTIPROCESSOR ARCHITECTURE

4.1 Introduction

One of the most rewarding applications of parallel processing is matrix manipulation. Matrix operations arise in applications related to structural analysis, transforms, image processing, fluid mechanics and partial differential equations, to name a few. Further, many of these applications require manipulation of matrices having very large orders, sometimes up to 100,000. In addition, matrices have elegant structures and their operations can be easily decomposed. Since matrix operations are such good candidates for parallel processing, it is worthwhile to develop models and methods to help "parallelize" matrix manipulation algorithms.

We have seen earlier that due to decomposition overhead and various other considerations, partitioning computational problems in equal halves may not achieve optimal parallel execution time. Nevertheless, traditionally, many parallel algorithms are based on this heuristic approach. Parallel matrix algorithms have not been exceptions. Fortunately, in many cases involving matrices, dividing problems in equal halves may in fact be optimal or near optimal. In this chapter, we develop and analyse a model which can be directly applied to some matrix operations, notably, matrix inversion. The main purpose of the discussion here is not to develop a specific model, but rather to illustrate a methodology to develop useful models.

The model which will be analysed below is represented by the equation

$$T(n) = \min\{\max\{T(n-r), T(r)\} + k[\log(r)] + \lambda\}, \quad 1 \leq r \leq n-1, \quad n \geq 2,$$

where k and λ are integer constants such that $\lambda \geq k \geq 0$.

Compared to the model discussed in chapter 2, this equation has *variable overhead*. The reason for the \log term is that many matrix operations with r operands can be done in a binary multiprocessor tree of height proportional to $\log(r)$. In [4], it is shown that multiplication of two matrices of sizes $m \times n$ and $n \times p$ respectively require exactly $[\log(n)] + 1$ steps. Since decomposition of matrix operations requires matrix multiplication in many cases, the \log term usually arises in the above context, as will be shown in the example of next section.

4.2 Model for matrix inversion

A very important matrix operation is inversion. One of the methods to invert matrices is the LU decomposition followed by inversion of each triangular matrix and then their multiplication.

This and various other applications make triangular matrix inversion an attractive problem for parallel processing. Various techniques have been devised to compute the inverse of a triangular matrix in parallel [4, 5, 14, 15].

The algorithm which will be modeled here uses recursive doubling. Let \mathbf{A} be a lower triangular matrix of order $n \times n$ which is partitioned as

$$\mathbf{A} = \begin{array}{cc} \mathbf{A}_1 & \mathbf{0} \\ \mathbf{A}_2 & \mathbf{A}_3 \end{array}$$

where \mathbf{A}_1 , \mathbf{A}_2 and \mathbf{A}_3 are matrices of orders $r \times r$, $(n-r) \times r$, and $(n-r) \times (n-r)$ respectively.

Since the inverse of a triangular matrix is itself triangular, \mathbf{A}^{-1} can be expressed as

$$\mathbf{A}^{-1} = \begin{pmatrix} \mathbf{B}_1 & \mathbf{0} \\ \mathbf{B}_2 & \mathbf{B}_3 \end{pmatrix}$$

where \mathbf{B}_1 , \mathbf{B}_2 and \mathbf{B}_3 are matrices having the same orders as \mathbf{A}_1 , \mathbf{A}_2 and \mathbf{A}_3 respectively. This representations leads to the following identities:

$$\mathbf{A}_1 \mathbf{B}_1 = \mathbf{I}_1,$$

$$\mathbf{A}_2 \mathbf{B}_1 + \mathbf{A}_3 \mathbf{B}_2 = \mathbf{0} \text{ and}$$

$$\mathbf{A}_3 \mathbf{B}_3 = \mathbf{I}_2,$$

where \mathbf{I}_1^{-1} and \mathbf{I}_2 are identity matrices of appropriate order. These equations then lead to the three-step procedure to compute \mathbf{A}^{-1} .

$$\mathbf{B}_1 = \mathbf{A}_1^{-1},$$

$$\mathbf{B}_3 = \mathbf{A}_3^{-1} \text{ and}$$

$$\mathbf{B}_2 = -\mathbf{B}_3 \mathbf{A}_2 \mathbf{B}_1.$$

Notice that inverting \mathbf{A}_1 and \mathbf{A}_2 are independent operations and thus, can be performed in parallel. The computation of $\mathbf{A}_3^{-1} \mathbf{A}_2 \mathbf{A}_1^{-1}$ can be done by solving $\mathbf{A}_3 \mathbf{Y} = \mathbf{A}_2$ in parallel with the inversion of \mathbf{A}_1 and \mathbf{A}_3 to compute $\mathbf{A}_3^{-1} \mathbf{A}_2$ and then multiplying it with \mathbf{B}_1 as soon as \mathbf{B}_1 is computed [4]. Since the last operation (multiplication with $r \times r$ dimension \mathbf{B}_1) takes $\lceil \log(r) \rceil + 1$ time steps, the total time to complete the inversion is

$$T(n) = \min\{\max\{T(n-r), T(r)\} + \lceil \log(r) \rceil + 1\}, \quad 1 \leq r \leq n-1, \quad n \geq 2.$$

In this equation, $T(n-r)$ and $T(r)$ are the times required to evaluate the inverses \mathbf{A}_3^{-1} and \mathbf{A}_1^{-1} ; the time required to solve $\mathbf{A}_3 \mathbf{Y} = \mathbf{A}_2$ is not indicated since it

is less than $T(r)$. The values of $T(n)$ and optimal r for this case are shown in table 4-1

Table 4-1: The time and optimal r values for parallel triangular matrix inversion.

n	T(n)	r
2	2	1
3	3	1
4	4	1, 2
5	5	1, 2
6	6	1, 2, 3
7	7	1, 2, 3, 4
8	7	4
9	8	1, 4
10	9	1, 2, 4, 5
11	10	1, 2, 3, 4, 5, 6
12	10	4, 6
13	11	1, 4, 5, 6, 7, 8
14	11	6, 7, 8
15	11	7, 8
16	11	8
17	12	1, 8
18	13	1, 2, 8, 9
19	14	1, 2, 3, 4, 7, 8, 9, 10
20	14	4, 8, 10
21	15	1, 4, 5, 6, 7, 8, 9, 10, 11, 12
22	15	6, 7, 8, 10, 11, 12
23	15	7, 8, 11, 12
24	15	8, 12
25	16	1, 8, 9, 10, 11, 12, 13, 14, 15, 16
26	16	10, 11, 12, 13, 14, 15, 16
27	16	11, 12, 13, 14, 15, 16
28	16	12, 13, 14, 15, 16
29	16	13, 14, 15, 16
30	16	14, 15, 16
31	16	15, 16
32	16	16

4.3 Analysis of variable overhead models

The computational complexity model of section 4.2 can be generalized to the following form:

$$T(n) = \min\{\max\{T(n-r), T(r)\} + k[\log(r)] + \lambda\}, \quad 1 \leq r \leq n-1, \quad n \geq 2. \quad (4.1)$$

In this thesis we only consider cases where $\lambda \geq k \geq 0$

We first show that $T(n)$ is a monotonically increasing function for all values of n .

Theorem 4.1. For all n , $T(n+1) \geq T(n)$.

Proof. We use mathematical induction over n to prove this result. First, assume that the function $T(n)$ is monotonically increasing for all values of $n < u$, i.e:

$$T(n+1) \geq T(n), \quad n < u.$$

We now prove that $T(u+1) \geq T(u)$.

Let $1 \leq r_1 \leq u$ denote the r used to evaluate $T(u+1)$. Then

$$T(u+1) = \max\{T(u+1-r_1), T(r_1)\} + k[\log(r_1)] + \lambda. \quad (4.2)$$

If $r_1 = 1$, then equation (4.2) becomes

$$T(u+1) = \max\{T(u), T(1)\} + \lambda.$$

But since $T(n)$ is assumed to be monotonically increasing till u , $T(u) \geq T(1)$ and thus

$$T(u+1) = T(u) + \lambda \geq T(u).$$

If $r_1 \neq 1$, then $2 \leq r_1 \leq u$ giving $1 \leq r_1 - 1 \leq u - 1$. Thus, $r_1 - 1$ is one of the r values used in the optimization of $T(u)$ in equation (4.2). Therefore,

$$T(u) \leq \max\{T(u-(r_1-1)), T(r_1-1)\} + k[\log(r_1-1)] + \lambda. \quad (4.3)$$

But since $r_1 - 1 < u$, our assumption gives $T(r_1) \geq T(r_1 - 1)$. Keeping this in mind, a comparison of the terms of equations (4.2) and (4.3) gives

$$T(u+1) \geq T(u).$$

Thus, the theorem is true for $n = u$ if it is true for $n < u$. From equation (4.2), we have $T(2) = T(1) + \lambda \geq T(1)$ showing that the theorem is true for $n = 2$. Thus, the proof is complete by induction. *Q.E.D.*

We will show later that $\lfloor n/2 \rfloor$ is an optimal value of r for all n . But first, we obtain a few properties of $T(n)$ assuming that optimal $r = \lfloor n/2 \rfloor$.

Define a new function such that

$$T_0(n) = \max\{T_0(n-r), T_0(r)\} + k[\log(r)] + \lambda, \quad r = \lfloor n/2 \rfloor. \quad (4.4)$$

Notice that if $r = \lfloor n/2 \rfloor$ is optimal, then $T(n) = T_0(n)$. The expression for $T_0(n)$ can be simplified to either one of the two following equations, depending on whether n is even or odd,

$$T_0(n) = T_0(m+1) + k[\log(m)] + \lambda, \quad n = 2m + 1, \text{ and} \quad (4.5)$$

$$T_0(n) = T_0(m) + k[\log(m)] + \lambda, \quad n = 2m. \quad (4.6)$$

Following three theorems describe the jumps in $T_0(n)$ as n increases and enable us to obtain an expression for $T_0(n)$.

Theorem 4.2. If $n = 2^i$, where i is a positive integer, then

$$T_0(n+1) = T_0(n) + \lambda.$$

Proof. The theorem would be proved by mathematical induction over i . It is clearly true for $i=1$ since equation (4.6) gives

$$T_0(2) = T_0(1) + \lambda.$$

Assume that it is true for all $i < l$. To show that it is true for $i=l$, note that equation (4.6) gives

$$T_0(2^l) = T_0(2^{l-1}) + (l-1) + \lambda.$$

However, since we assumed that the theorem statement is true for $i \leq l$, we get

$$\begin{aligned} T(2^l) &= T_0(2^{l-1}+1) + (l-1) + \lambda - \lambda \text{ (from the above assumption)} \\ &= T_0(2^l+1) - \lambda \text{ (from equation (4.5)).} \end{aligned}$$

This shows that the relation is valid for $i=l$.

Q.E.D.

Theorem 4.3. For all the values of n which are expressible as $n = 2^i + 2^j$, where i and j are two distinct nonnegative integers,

$$T_0(n+1) = T_0(n) + k.$$

Proof. We prove this theorem using mathematical induction over n . The relationship stated above is true for $n=3$ since

$$T_0(4) = T_0(2) + \lambda + k \text{ from equation (4.6),}$$

which gives using equation (4.5),

$$T_0(4) = T_0(3) + k.$$

We now assume that the theorem statement is true for all $n < u$ and prove it for $n = u$.

Let $u = 2^i + 2^j$, $i \neq j$. If neither i nor j is zero, then u is even and thus, from equation (4.6), $T_0(u)$ can be expressed as

$$T_0(u) = T_0(2^{i-1}+2^{j-1}) + k[\log(2^{i-1}+2^{j-1})] + \lambda.$$

Our assumption gives

$$T_0(2^{i-1}+2^{j-1}) = T_0(2^{i-1}+2^{j-1}+1) - k, \text{ which gives}$$

$$T_0(u) = T_0(2^{i-1}+2^{j-1}+1) + k[\log(2^{i-1}+2^{j-1})] + \lambda - k.$$

The first three terms on the right hand side can be combined using equation (4.5) to give the value of $T_0(u+1)$.

$$\text{Thus, } T_0(u+1) = T_0(u) + k.$$

On the other hand, if (say) $i=0$, then $u = 2^j + 1$ and using equation (4.5), we get

$$T_0(u) = T_0(2^{j-1}+1) + k(j-1) + \lambda,$$

and from equation (4.6)

$$T_0(u+1) = T_0(2^{j-1}+1) + k(j) + \lambda.$$

Clearly, in this case also

$$T_0(u+1) = T_0(u) + k.$$

Q.E.D.

Theorem 4.4. If $n \neq 2^i + 2^j$, for any two nonnegative integers i and j , then

$$T_0(u+1) = T_0(u).$$

Proof. The theorem is true for $n=7$ (the first such n value) since from equations (4.5) and (4.6), $T_0(7) = T_0(8) = T_0(4) + 2k + \lambda$. If the theorem is true for all $n < u$, then it can be shown to be true for $n = u$, and by mathematical induction, the proof would be complete.

If u is even ($u = 2m$), then from equations (4.6) and (4.5), we have

$$T_0(u) = T_0(m) + k[\log(m)] + \lambda, \text{ and}$$

$$T_0(u+1) = T_0(m+1) + k[\log(m)] + \lambda.$$

But $T_0(m+1) = T_0(m)$ since m is not a sum of two powers of 2 and $m < u$.

Thus,

$$T(u+1) = T(u).$$

If u is odd ($u = 2m + 1$), then from equations (4.5) and (4.6),

$$T_0(u) = T_0(m) + k[\log(m)] + \lambda, \text{ and}$$

$$T_0(u+1) = T_0(m+1) + k[\log(m+1)] + \lambda.$$

Clearly, $[\log(m)] = [\log(m+1)]$, else m is a power of 2, say 2^l . This is not possible since it implies that $u = 2^{l+1} + 2^0$, which is a contradiction since $u \neq 2^i + 2^j$ for any i and j . Q.E.D.

The last three theorems are of paramount importance in predicting $T_0(n)$ for any n . They show that jumps of known magnitude in $T_0(n)$ occur only at n 's of the form $2^i + 2^j$. $T_0(n)$ can thus be found by examining all the integers which have the form $2^i + 2^j$ and are strictly smaller than n . In particular, the following corollary gives an exact expression for $T_0(n)$.

Corollary 4.1. For all n ,

$$T_0(n) = ([\log(n)] + 1)\lambda + k[\log(n)]([\log(n)] - 1)/2 + k[\log(n - 2^{[\log(n)]})] + T(1), \quad n \neq 2^i \quad (4.7)$$

$$= [\log(n)]\lambda + k[\log(n)]([\log(n)] - 1)/2 + T(1), \quad n = 2^i, \quad i \geq 1. \quad (4.8)$$

Proof. To find $T_0(n)$, one can examine the jumps which are encountered by the function $T_0(m)$ as m increases from -say- 1 to n . Theorem 4.2 states that at any value of n which can be expressed as 2^i , $T(n)$ encounters a jump of

magnitude λ . Similarly, theorem 4.3 states that at any n having the form $2^i + 2^j$, $i \neq j$, $T_0(n)$ encounters a jump of magnitude k . On the other hand, theorem 4.4 states that no jumps occur at those n 's which do not have the form 2^i or $2^i + 2^j$. This suggests that one can compute $T_0(n)$ by finding the number of jumps caused by the integers having the form 2^i and $2^i + 2^j$, multiplying them with λ and k respectively and adding them to -say- $T_0(1)$.

Let $l = \lfloor \log(n) \rfloor$. Define S_n as the set of integers which are strictly less than n and have the form 2^i . Similarly, define Z_n as the set of integers which are strictly less than n and have the form $2^i + 2^j$, $i \neq j$. Then,

$$T(n) = \lambda|S_n| + k|Z_n| + T(1).$$

Notice that

$$\begin{aligned} |S_n| &= (l+1), \quad n \neq 2^l \\ &= l, \quad n = 2^l. \end{aligned}$$

Finding $|Z_n|$ involves finding the number of all the possible weight-2 vectors among a vector with l components. Thus,

$$\begin{aligned} |Z_n| &= \sum_{j=1}^{l-1} j + \lfloor \log(n-2^l) \rfloor, \quad n \neq 2^l \\ &= \sum_{j=1}^{l-1} j, \quad n = 2^l. \end{aligned}$$

Substituting $l(l-1)/2$ for $\sum_{j=1}^{l-1} j$ and $\lfloor \log(n) \rfloor$ for l completes the proof. *Q.E.D.*

4.4 Optimal partitioning for variable overhead

We now show how to optimally partition the problem, which is achieved by determining the values of optimal r 's. In particular, it is shown that optimal partitioning is achieved by choosing r at any stage of computation as $r = \lfloor n/2 \rfloor$ where n is the size of the problem at that particular stage.

Theorem 4.5. For $T(n)$ defined in equation (4.1), $\lfloor n/2 \rfloor$ is an optimal value of r , i.e.

$$T(n) = T_0(n).$$

Proof. The theorem would be proved by mathematical induction over n . Notice that for $n=2$, the optimal value of r is 1. Assume that the theorem statement is true for all $n \leq u$. Next, we prove that this infers that it is valid for all $n \leq u+1$.

Since we assume that $T(n) = T_0(n)$ for all $n \leq u$, theorems 2, 3 and 4 are valid for $T(n)$ if $n \leq u$.

First, consider the case when u is even, and let $u = 2m$. Using equation (4.6), $T(u)$ is expressed as

$$T(u) = T(m) + k[\log(m)] + \lambda.$$

On the other hand, equation (4.5) gives for

$$T_0(u+1) = T(m+1) + k[\log(m)] + \lambda.$$

To compare the above two equations, three distinct cases must be identified; $m \neq 2^i + 2^j$, $m = 2^i$ and $m = 2^i + 2^j$, when $i \neq j$.

Case 1. If $m \neq 2^i + 2^j$, then according to theorem 4, $T(m) = T(m+1)$. This clearly means that $T_0(u+1) = T(u)$, which implies that $T(u+1) = T_0(u+1)$ since $T(u+1)$ can not be less than $T(u)$.

Case 2. If $m = 2^i$, then according to theorem 2, $T(m+1) = T(m) + \lambda$. Thus, the expression for $T_0(u+1)$ becomes

$$\begin{aligned} T_0(u+1) &= T(m) + k[\log(m)] + 2\lambda. \\ &= T(u) + \lambda. \end{aligned}$$

If $T(u+1) \geq T(u) + \lambda$, then $T(u+1) = T_0(u+1)$. Since $u+1 = 2m+1$, we have

$$T(u+1) = T(2^i+1-r) + k[\log(r)] + \lambda.$$

We want to prove that for any r , $T(u+1) \geq T_0(u+1)$ since this implies that $r = \lfloor n/2 \rfloor$ is optimal.

Consider the case when $r \leq m$, such that $r = m - p$, $p \geq 0$. Then

$$T(u+1) = T(2^i+p+1) + k[\log(2^i-p)] + \lambda. \quad (4.9)$$

For $p=0$, we clearly have $r=m$. If we want to find an r which yields a value of $T(u+1)$ which is less than $T(u) + \lambda$, then p must be increased sufficiently to decrease the \log term in equation (4.9). However, as p increases, the decrease in the \log term is achieved at the possible expense of an increase in $T(2^i+p+1)$, the first term in the right hand side of equation (4.9). We intend to prove that the decrease in the \log term is more than offset by an increase in the first term, so that $T(u+1)$ can never decrease below $T(u) + \lambda$. Decreasing the \log term by l , requires that we half its argument l times. This can be achieved with a p having the form

$$p = 2^{i-1} + 2^{i-2} + \dots + 2^{i-l}.$$

Thus, the expression for $T(2^i+p+1)$ becomes

$$T(2^i+p+1) = T(2^i+2^{i-1}+2^{i-2}+\dots+2^{i-l}).$$

For the value of p above, corollary 4.1 and equation (4.9) give

$$T(u+1) = (i+1)\lambda + ki(i+1)/2 + T(1) + k(i-l) + \lambda, \quad l < i \quad (4.10)$$

$$= (i+2)\lambda + ki(i+1)/2 + T(1) + \lambda, \quad l = i \quad (4.11)$$

Since $u = 2^i$, corollary 4.1 can be easily applied to find the value of $T(u)$ as

$$T(u) = (i+1)\lambda + ki(i+1)/2 + T(1). \quad (4.12)$$

Comparing equation (4.12) against equations (4.10) and (4.11) show that

$T(u+1) \geq T(u) + \lambda$, and therefore that $T(u+1) = T_0(u+1)$.

Case 3. If $m = 2^i + 2^j$, $i \neq j$, then according to theorem 3, we have

$T(m+1) = T(m) = k$. Thus, the expression for $T_0(u+1)$ becomes

$$\begin{aligned} T_0(u+1) &= T(m) + k[\log(m)] + k + \lambda \\ &= T(u) + k. \end{aligned}$$

If $T(u+1) \geq T(u) + k$, then $T(u+1) = T_0(u+1)$. Since $u+1 = 2m+1$, we have

$$T(u+1) = T(2^{i+1}-r) + k[\log(r)] + \lambda.$$

We want to prove that for any r , $T_0(u+1) \geq T(u+1)$ since this implies that $r = \lfloor n/2 \rfloor$ is optimal.

Consider the case when $r \leq m$, such that $r = m - p$, $p \geq 0$. Proceed as in case 2 above, with the exception that we intend to reduce the \log term in equation (4.9) by $l+1$, so we add 2^j to p . Therefore, p is expressed as

$$p = 2^{i-1} + 2^{i-2} + \dots + 2^{i-l} + 2^j.$$

Thus, the expression for $T(2^i + 2^j + p + 1)$ becomes

$$T(2^i + p + 1) = T(2^i + 2^{i-1} + 2^{i-2} + \dots + 2^{i-l} + 2^j + 1). \quad (4.13)$$

For the value of p above, corollary 4.1 and equation (4.9) give

$$T(u+1) = (i+1)\lambda + ki(i+1)/2 + T(1) + k(i-l) + \lambda, \quad (j+1) < (i-l) \quad (4.14)$$

$$= (i+2)\lambda + ki(i+1)/2 + T(1) + k(i-l) + \lambda, \quad (j+1) = (i-l) \quad (4.15)$$

$$= (i+2)\lambda + ki(i+1)/2 + kj + T(1) + k(i-l) + \lambda, \quad (j+1) > (i-l) \quad (4.16)$$

Notice that when $j+1 \geq i-l$, the term 2^{j+1} is repeated twice in the equation (4.13); this means that the two terms add and produce a "carry", which "propagates" so that all the two powers 2^{j+1} through 2^i add up to 2^{i+1} . Since $u = 2^i + 2^j$, corollary 4.1 can be easily applied to find the value of $T(u)$ as

$$T(u) = (i+2)\lambda + ki(i+1)/2 + kj + T(1). \quad (4.17)$$

Comparing equation (4.17) against equations (4.14) through (4.16) show that $T(u+1)$ can not be less than $T(u) + k$. Notice that dropping the term 2^j from the expression for p , would not affect the above argument or its conclusion. This implies that $T(u+1) = T_0(u+1)$. A similar argument can be used to show that the same result is inferred for any $r > m$.

Thus, in all three cases, the assumption that $T(n) = T_0(n)$ for all $n \leq u$, where u is even leads to the conclusion that $T(u+1) = T_0(u)$. A similar argument can be employed for an odd u to reach the same conclusion.

Therefore, by mathematical induction, $T(n) = T_0(n)$. *Q.E.D.*

The above theorem is very important in that it shows that a fairly simple partitioning scheme is optimal for the variable recombination overhead of the form $k[\log(r)] + \lambda$ when $\lambda \geq k \geq 0$ (if $\lambda = 0$, then optimal partitioning is achieved with $r = 1$). This result is not valid if $k > \lambda$ but a similar discussion in this case is beyond the scope of this thesis. This result should be compared with corollary 2.2.

Theorems 4.4 and 4.5 together imply that the optimal complexity in most matrix problems would be $O(k \log_2^2(n))$.

Chapter 5

HARDWARE IMPLEMENTATION OF PARALLEL ALGORITHMS

5.1 Introduction

The rapid advent of Very-Large-Scale-Integrated (VLSI) technology has created new architectural possibilities in implementing parallel algorithms directly in hardware. The current technology has made possible the fabrication of more than 250 000 transistors on a single chip. Such a technology can be used effectively in designing high-performance processors dedicated towards one type of computational problems. Such dedicated hardware "functional units" usually run under a more general-purpose processing or control unit which acts as a task arbiter.

The design of functional units poses some challenges. The hardware must be modular and cost effective. A functional unit implements in hardware an algorithm which solves the computational problem to which it is dedicated. Careful consideration must be given to the algorithm used since it determines the hardware complexity, speed and interconnectivity of the hardware unit. It generally results in a trade off between hardware complexity and speed.

When implementing parallel algorithms directly in hardware, it is important to differentiate between temporal parallelism and spatial parallelism. Let a processor be defined as hardware unit which performs a certain process which essentially is a group of sequential operations on some input operands. A

process frame is the process pertaining to a specific input.

Temporal parallelism is achieved by decomposing the process into sequential "subprocesses". In this case, a hardware "subprocessor" is assigned to every subprocess. Subprocessors are actually specialized hardware segments controlled in such a manner that several segments may be busy simultaneously. One subprocessor may be assigned to more than one subprocess if their structures are the same. The subprocessors are then pipelined with possible feedback and feedforward paths, so that a subprocessor may be engaged in a subprocess without necessarily waiting for the whole processor to complete the execution of a process frame. Thus, a subprocessor may start working on subprocess as soon as it completes the previous one independently of the whole process. In practice, a subprocessor may not be completely engaged at all times because of timing and synchronization constraints, nevertheless, its throughput is usually significantly improved. Thus, essentially temporal parallelism is pipelining consecutive modules of hardware rather than duplicating them, so that a process flows from segment to segment in a lock-step synchronous manner. On the other hand, spatial parallelism is achieved by duplicating hardware such that several processors run distinct process frames concurrently. Obviously, each processor in the pool of available processors may be internally pipelined, as well as being able to function in a pipeline. Henceforth, parallelism is used to indicate spatial parallelism.

If pipelining alone can not achieve the required performance level, use of spatial parallelism is indicated. However, it is important to use temporal

parallelism as often as possible, since pipelining is a technique which significantly increases hardware throughput at little or no extra cost.

In this chapter, a processor dedicated for polynomial evaluation will be developed. The processor will implement in hardware the statically recursive parallel algorithm described earlier.

5.2 Polynomial evaluation by statically recursive doubling

In the case of polynomial evaluation, Horner's rule poses itself as the most suitable algorithm for hardware implementation because of its simplicity and modularity. However, Horner's rule is sequential. If high performance is a premium, a parallel algorithm must be used even though it will result in loss of efficiency and will increase the hardware complexity. Any of the parallel algorithms described in the previous sections can be used. Dynamically recursive parallel algorithms demand extreme flexibility and are rather cumbersome to implement in hardware. On the other hand, statically recursive parallel polynomial evaluation is as modular as Horner's rule since it is a generalization of it.

This chapter starts with a hardware implementation of Horner's rule not only because it is useful in its own right for direct polynomial evaluation, but also because it is the building block for a parallel pipelined processor for polynomial evaluation.

A polynomial P_n can be expressed as

$$x(x(\dots(a_n x + a_{n-1}) + a_{n-2}) + \dots + a_1) + a_0.$$

Horner's rule evaluates a polynomial by computing the value of each term in the expression above starting with the innermost term and subsequently the next expression and so on until the polynomial P_n is evaluated. Thus, Horner's rule can be viewed as a sequence of first degree polynomial evaluations where the first degree coefficient of any polynomial (except the innermost one) is a result of the preceding polynomial evaluation. In hardware, this translates into a highly modular design, where each module evaluates a first degree polynomial using a result computed by the preceding module.

Evaluation of first degree polynomials involves one multiplication and one addition. Thus, a hardware module which evaluates a first degree polynomial $ax+b$, consists of processing elements which can do both operations. The multiplication operation is, of course, far more complicated than the addition operation and it will dominate the hardware. Any known word-oriented multiplier can be used. Let P_1 designate a hardware module which computes a first degree polynomial. Let the delay of a P_1 module be d_1 , and δ_i be a delay operator with a delay of $i.d_1$.

Figure 5-1 shows the basic outline of a processor which evaluates a polynomial of degree n using Horner's rule. Notice that throughout the processor, only one type of hardware module is used, namely P_1 . The modules can be easily pipelined for higher throughput. A more careful consideration of the circuit reveals that only one P_1 module is needed, since we can easily feedback its output to its input. Such a processor use *only one* P_1 module rather than n modules. Any polynomial P_ρ of an arbitrary degree ρ can be

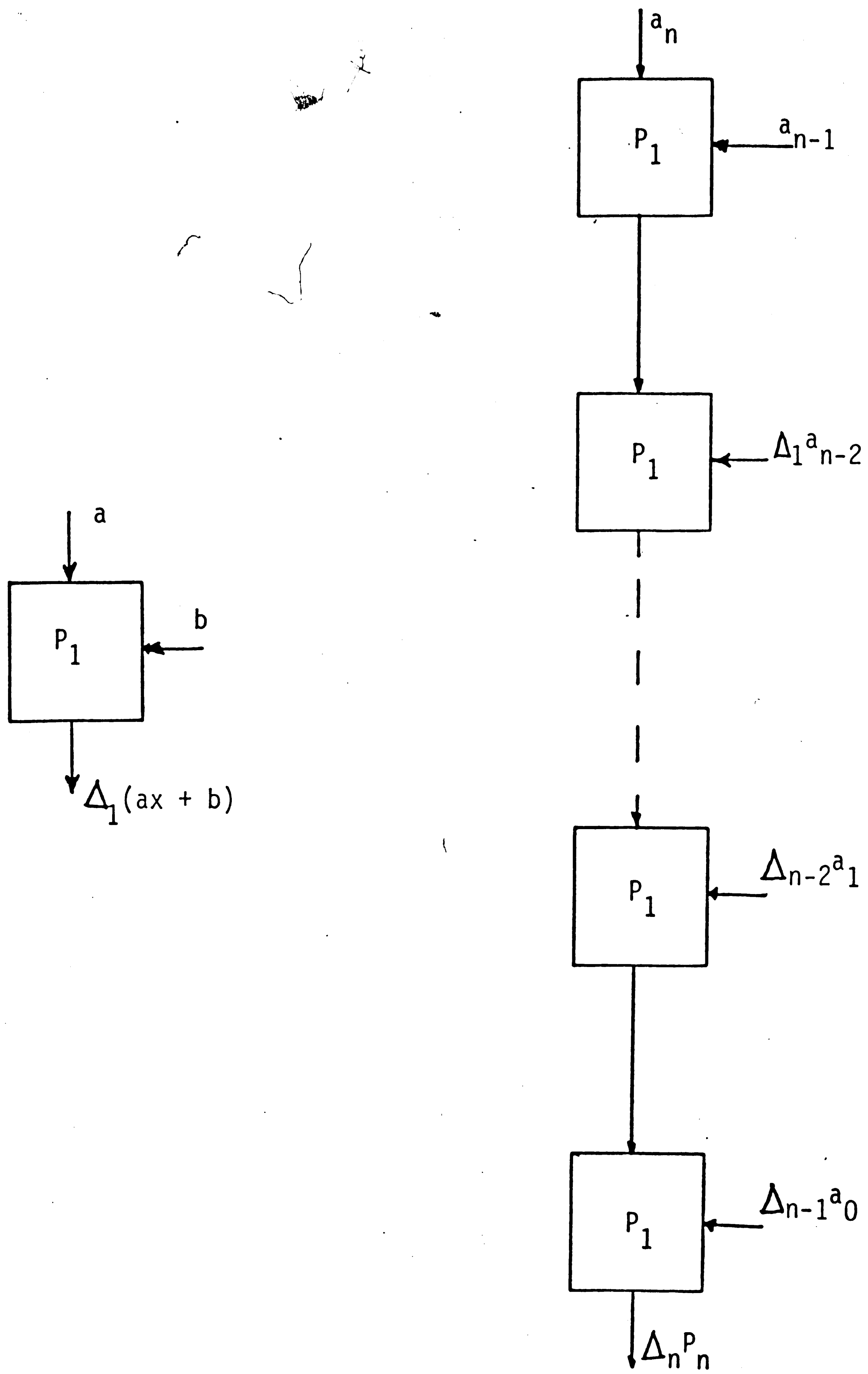


Figure 5-1: A direct implementation of Horner's rule to evaluate $P_n = a_n x^n + a_{n-1} x^{n-1} + \dots + a_0$

evaluated simply by clocking the module ρ times. Such a processor is designated by P_ρ . Figure 5-2 shows such a processor with the proper sequence of inputs.

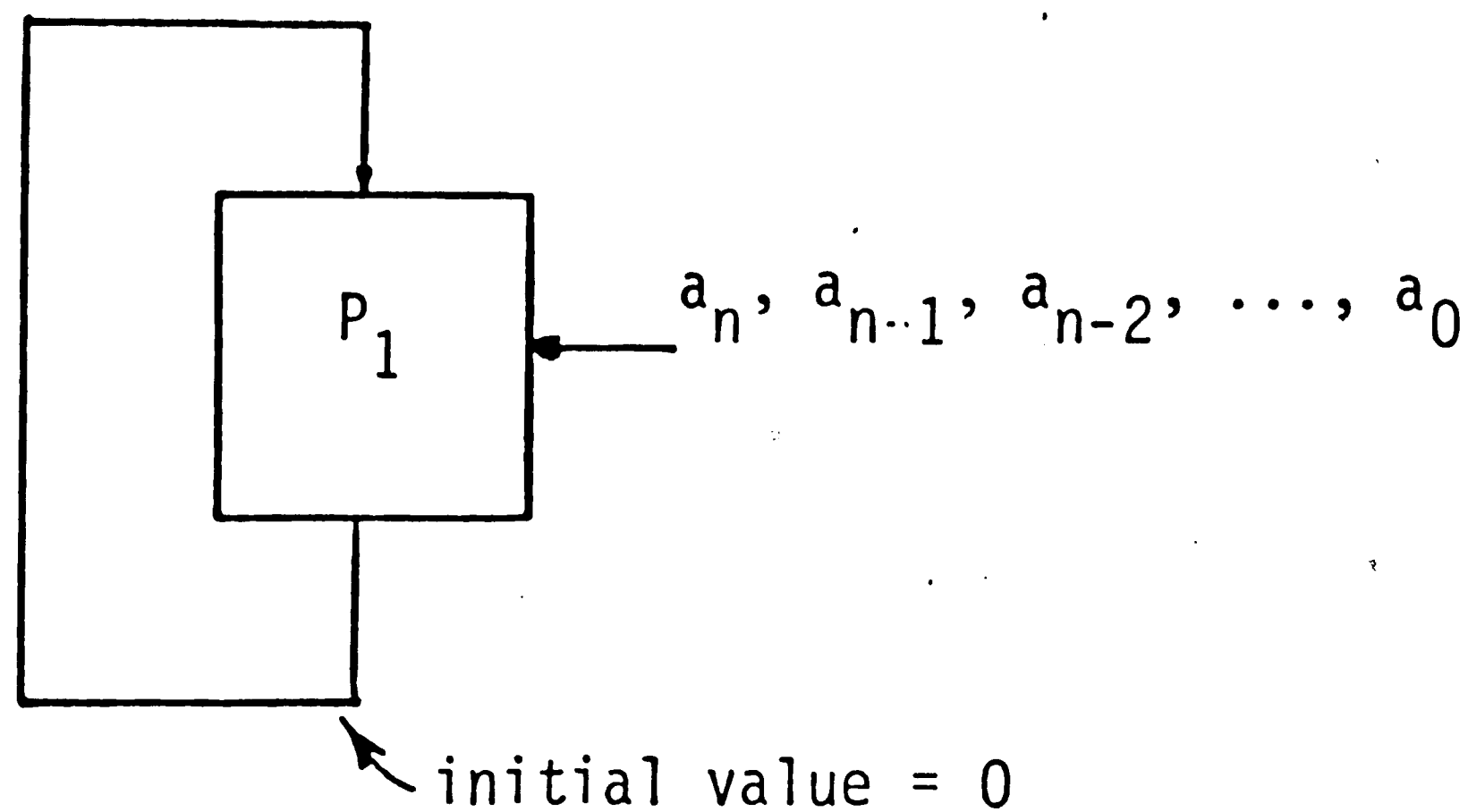


Figure 5-2: A processor which evaluates a polynomial P_ρ using Horner's rule. Notice the input sequence at every initialization of the module.

The evaluation of a polynomial of degree ρ requires $d_\rho = \rho d_1$ time units regardless of the number of the P_1 modules in the processor. However, a multi-module processor can achieve a better *average* throughput if many polynomials are to be evaluated. If the application requires the evaluation of a large number polynomials in a burst-like fashion, then many P_ρ processors can be used in parallel, each evaluating a different polynomial. For this reason, Horner's rule is best implemented using one P_1 module.

Since the use of several processors independently does not speed up the evaluation of a single polynomial, one may have to use a parallel polynomial evaluation algorithm.

As shown in (3.2), a polynomial P_n can be expressed as

$$P_n = x^r(x^r(x^r(\dots(x^r.P_{n-kr} + P_{r-1}) + P_{r-1}) + P_{r-1}) \dots + P_{r-1}.$$

Thus, a polynomial of degree n can be evaluated by evaluating in parallel $k = \lfloor n/r \rfloor$ polynomials each of degree $r-1$, and combining the results of these calculations in the right sequence. Figure 5-3 shows the outline of a processor which performs these steps. The processor has k stages, where each stage has one P_1 module and one P_{r-1} module except the first stage which has two P_{r-1} modules. Straightforward use of this processor allows the evaluation of polynomials having a degree up to $(k+1)r-1$. The modules which constitute the processor are pipelined, so that each module may be used as soon as it completes one evaluation to work on the next polynomial.

Since all stages of the processor shown in figure 5-3 are basically the same, it is also possible to feedback the output of the processor to its input, to increase the effective number of processor stages without increasing them physically. To ensure collision free scheduling, the output can not be fed back to the input unless the first stage is free. But since the very first P_1 block starts processing at time d_{r-1} and the first P_{r-1} is free at that time, the output of the first P_1 may be fed to the input of the first P_{r-1} . Thus, static recursion does not yield a highly parallel architecture.

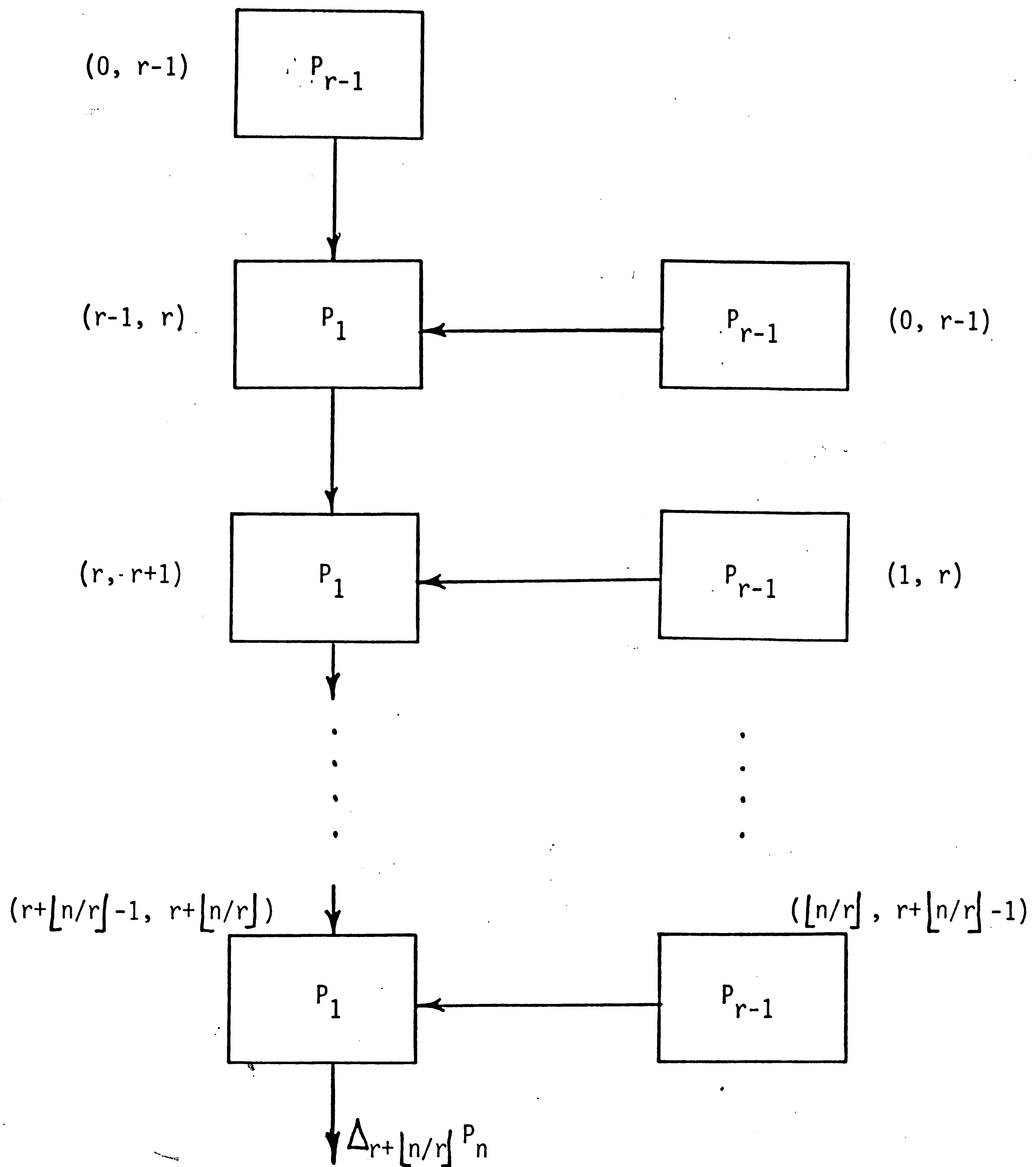


Figure 5-3: Direct hardware implementation of static parallel polynomial evaluation. The timing for each module is shown in the 2-tuple written besides each module. The first parameter shows the time when the input to the module must be presented and the second parameter shows the time when the output is available.

5.3. Implementation of dynamically recursive partitioning

Dynamically recursive partitioning typically produces algorithms which require a high level of interprocessor communication. For example, in parallel evaluation of polynomials, if processors are assigned to specific operations, they often need operands computed by other processors. If the time required to fetch an operand from another processor is significant, then parallelizing the computation may not achieve any benefits. Thus, high bandwidth communication must be used in order to avoid communication bottlenecks in these architectures.

Figure 5-4 shows an outline of an architecture suitable for dynamically recursive parallel computing. Interprocessor communication can be achieved using common memory modules, a high bandwidth bus with some arbiter or a cross switch. Any communication scheme is likely to be complex because of the performance rates required. Notice that such a system, with its high speed and flexible communication, is essentially a MIMD computer. Since the communication scheme constitutes a major part of the system hardware, dedicating the whole system to one type of application is not economically justifiable. Such an architecture is inherently flexible and thus, it may as well be used for several applications.

Nevertheless, for some applications, it may be feasible to implement some complex parallel algorithms in hardware. Hwang has proposed a parallel VLSI architecture to invert matrices based on LU decomposition and triangular matrix inversion, as a major hardware extension to supercomputers [15] which the

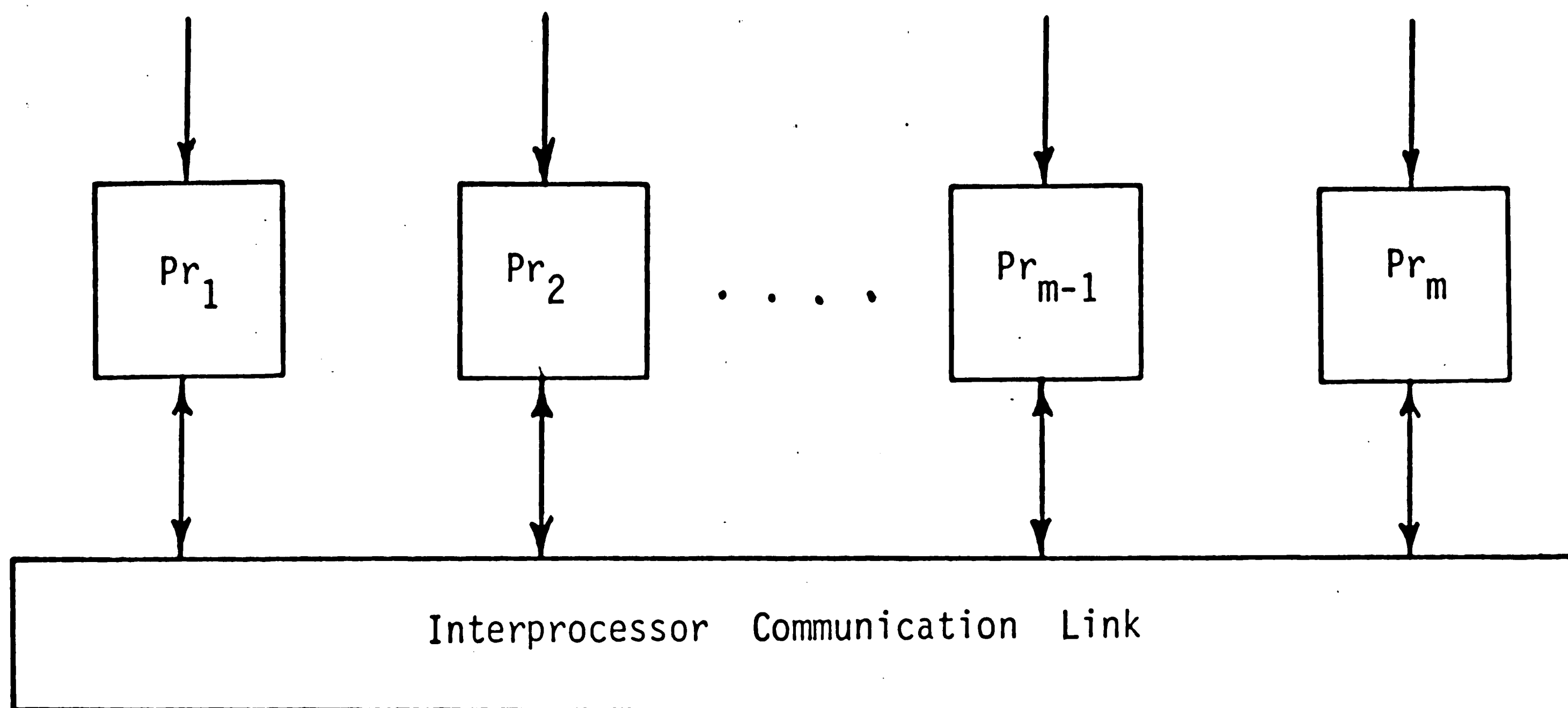


Figure 5-4: The layout of an architecture suitable for dynamically recursive parallel algorithms.

readers will find interesting.

5.4 Bit-Sequential polynomial evaluation

Bit-sequential processing can be used to further increase the throughput without increasing the hardware complexity. Pipelining Bit-sequential modules means that a module does not have to wait for a previous module to complete its task, but rather, can immediately start working using each input bit as it comes out from the previous module.

5.4.1 A Hardware implementation of Horner's rule

As mentioned earlier, the basic block of Horner's rule evaluation is simply the computation of $ax + b$. The coefficient a is, in general, another polynomial of the same form, and thus it is available one bit at a time.

Figure 5-5 shows a bit sequential circuit which computes $ax + b$. The multiplication of a and x dominates the hardware. The first row of full adders implement a multiplier which takes two inputs. The first input is a vector representing x and the second is a bit sequential input a_i which represents the first coefficient of the polynomial P_1 . In general, a_i comes from a previous module as is explained above. The product comes out from the first adder, one bit every clock period, starting with the least significant bit.

This multiplier has as many adders as the number of significant bits of x (which henceforth will be referred to as the multiplicand), and can process an arbitrarily long multiplier.

Let us assume that both the multiplier and the multiplicand are numbers represented in two's complement using ν bits. For correct multiplication of negative numbers, the operands have to be sign extended to as many bits as the number significant bits of the product. This will almost double the hardware complexity of the multiplier, since the number of full adders are determined by the number of bits of the multiplicand, x . However, since x is known in advance, its sign can be used to derive the complement of the product as it computed one bit every clock period, removing the necessity of

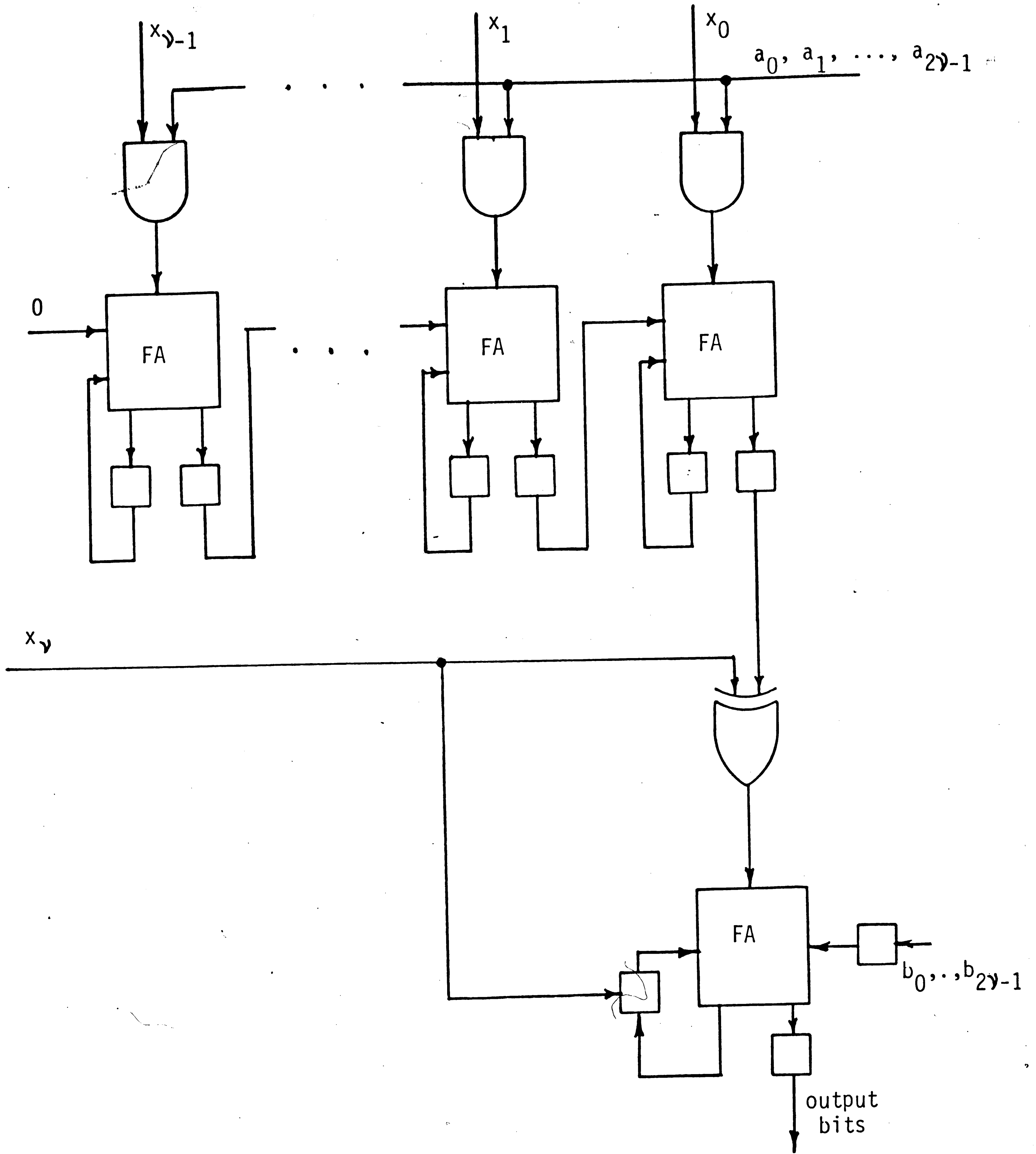


Figure 5-5: A bit-sequential processor for the computation of first degree polynomials

accommodating a negative multiplicand. This solution cuts the number of adders in half. The multiplicand, x , is originally represented in ν bits, however, since only the absolute value of x is used, we need only $\nu-1$ bits to represent x and therefore, $\nu-1$ adders. The multiplier has ν significant bits, thus, the product has $2\nu-1$ significant bits. Since negative operands need to be sign extended to as many bits as the number of significant bits in the product, the multiplier must be sign extended to $2\nu-1$ bits. The previous solution can not be used to avoid sign extending the multiplier, since its sign will not be known until ν clocks would have elapsed. Extending the multiplier does not incur any additional hardware, but it causes the multiplication to require twice as many clock periods.

Since inverting the sign of a binary number in two's complement can be done by complimenting each bit and adding one, the sign bit of x is used to determine whether to compliment the product bits as they come out of the multiplier; this is accomplished by XORing the product bit with the sign bit of x . If the sign bit of x is 1, the product must also be incremented. This can be done in an additional full adder after the XOR gate. Fortunately, the last adder serves another purpose as well. If the sign of x is used to set (or reset) the carry Flip-flop of the last adder, this full adder is quite sufficient to add b , the second coefficient of the polynomial, to the product ax . The second coefficient is input bit-sequentially delayed one clock period relative to the first bit of the first coefficient. In order to synchronize a and b , the second coefficient b is latched. The input b must be sign extended to $2\nu-1$ bits to be compatible with the product ax . The output of the last adder is the two's

complement binary representation of $ax+b$ expressed in 2ν bits coming one at a clock period.

Table 5-1: A simulation of the module P_1 , with $x=7$, $a=-8$, $b=-8$ and $\nu=4$.

Clock	Adder	a_i	b_i	Output	
0	: 0000 0000	0	0	0	
1	: 0000 0000	0	0	0	
2	: 0000 0000	0	0	0	First bit of output
3	: 0000 0000	1	1	0	
4	: 0000 0111	1	1	0	
5	: 0000 0100	1	1	0	
6	: 0000 0110	1	1	0	
7	: 0000 0111	0	0	0	
8	: 0000 0000	0	0	1	
10	: 0000 0011	0	0	1	Last bit of output
11	: 0000 0001	0	0	1	
12	: 0000 0000	0	0	1	
13	: 0000 0000	0	0	0	

Since this multiplier will be used as a module in a pipelined circuit, it is important to determine the timing relationships between the input and the output. Let the clock period when the the first bit of a enters the circuit be a reference point in time. The circuit timing can be analysed by referring all other occurances in time as an offset to that period. The first bit of the polynomial comes out at an offset of 2 and the last at an offset of $2\nu+1$.

In order not to let the number of bits grow geometrically, the precision of the output has to be limited to ν bits. Thus, the least significant ν bits of the output representing $ax+b$ must be truncated, so only ν bits are used and fed to the next and identical module. Let the circuit which produces such an output

be designated as module P_1 . Notice that if the output of a module is to be used as an input to another module, the result must be treated as if it is the first coefficient of the next first degree polynomial to be evaluated, and so it must be sign extended from ν to $2\nu-1$ bits.

Figure 5-6 shows a pipeline connecting several P_1 modules. The same figure also shows the timing relationships between various signals. It may be observed that the output of the second module starts coming at a time when the first module is available for another computation. Thus, a pipeline of more than two P_1 modules is redundant. A pipeline of two P_1 modules can be used to evaluate polynomials of an arbitrary degree by feeding back its output to its input a certain number of times. Actually, a controller can be used to determine the necessary number of feedback instances around the pipeline, depending on the degree of the polynomial which is to be evaluated.

A module composed of two P_1 modules computes a second degree polynomial ax^2+bx+c without feedback. Such a " P_2 " module receives four inputs; a vector representing x , and three bit-sequential inputs representing a , b and c . The inputs a and b are synchronous, while c is delayed $\nu+2$ clock periods.

As is explained in the previous section, only P_1 modules are enough to implement statically recursive evaluation of polynomials. Thus, the above observation shows that only two P_1 modules are needed to implement the P_{r-1} modules for any value of r by employing proper feedback.

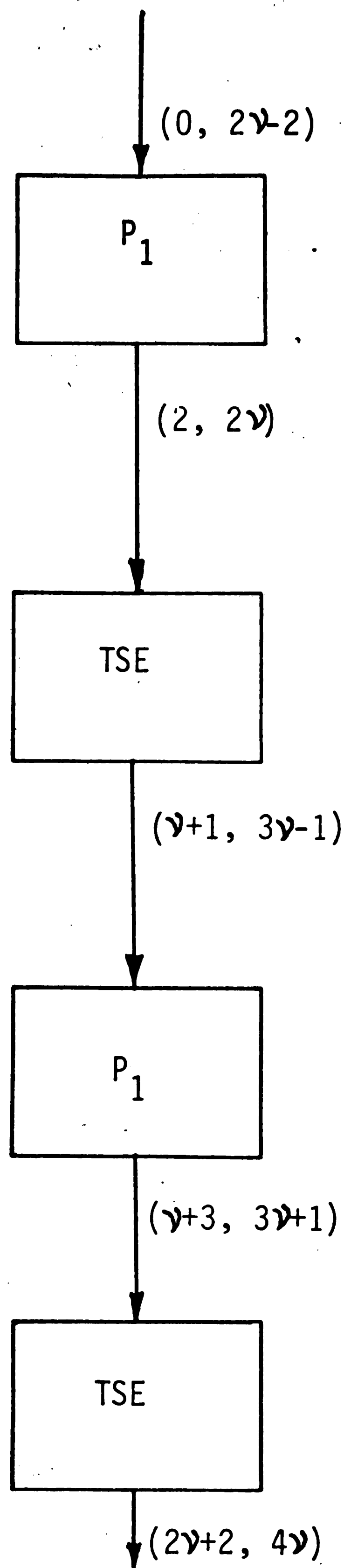


Figure 5-6: A pipe containing segments of P_1 modules.
 The two-tuple numbers shows the time when the first and last bits of different modules are available.
 The module "TSE" truncates the $2\nu - 1$ bit result to most significant ν bits and sign extends it to $2\nu - 1$ bits.

Chapter 6

CONCLUSION

This thesis discusses the importance of partitioning in the development of parallel algorithms using recursive doubling. It is shown that partitioning a problem into two equal halves is not optimal except under certain conditions. A model to study the complexity of a parallel algorithm using recursive doubling is developed and used to study the effects of various partitioning schemes.

Since partitioning introduces overhead whose exact form depends on the nature of the problem, the model could not be studied in complete generality. This thesis considers important classes of overhead, namely constant overhead and variable logarithmic overhead. These two cases cover a large segment of practical problems.

In the case of constant overhead, expressions are developed to describe the performance, and the partition range which give optimal performance is specified. It is shown that partitioning-by-half is optimal only when there is no partitioning overhead and the sizes of the subproblems add up to the size of the original problem.

The results of the constant overhead model are used to develop a parallel algorithm for the evaluation of polynomials. It is shown that decomposing a polynomial according to the golden numbers (Fibonacci numbers) yields an optimal partitioning scheme with the order of complexity equal to $O(c \log(n))$.

In the case of variable overhead, a model containing a logarithmic term in the form of $k[\log(r)] + \lambda$, where k and λ are constant integers and $\lambda \geq k \geq 0$, is used since it is typical of parallel algorithms for matrix manipulation. In particular, triangular matrix inversion is shown to conform to such a model. It is shown that in this case, partitioning a problem into two equal (or near equal) problems is optimal. An exact expression is developed to give the computational complexity of the algorithm in this general case. It is shown that the optimal computational complexity is of the order of $\log^2(n)$.

Some possible applications of parallel algorithms include high throughput dedicated processors. In chapter 5, parallel processors dedicated to polynomial evaluation are developed. A parallel bit-sequential processor pipelined at the bit level is shown and its performance is analysed.

The results developed in this thesis have significant applicability. Nevertheless, because of the problem dependency of partitioning, only few types of problems are described. However, it should be stressed that the *methodology* used is quite general and is applicable to *many different classes of problems*. In addition, it is possible to accommodate operations requiring different execution times without any change in the model.

This work assumes that algorithms are targeted towards computers having as many processors as required. As an extension of this work, it would be useful to develop a model to relate the performance of a parallel algorithm to the number of available processors as well as to the partitioning scheme.

REFERENCES

1. A. Aho and A. Ullman, *The Design and Analysis of Computer Algorithms*. Addison-Wesley, 1975.
2. D. Knuth, *The Art of Computer Programming*. Volume 1: Fundamental Algorithms. Addison-Wesley, 1968.
3. S. Sahni and E. Horowitz, *Fundamentals of Computer Algorithms*. Computer Science Press, 1978.
4. D. Heller, *A survey of parallel algorithms in numerical linear algebra*. Siam Review, Vol. 20 (1978), pp. 740-777.
5. A. Borodin, *Computational Complexity of Algebraic and Numeric Problems*. American Elsevier, 1975.
6. J. Ja'Ja', *Time-space trade-offs for some algebraic problems*. J. ACM, Vol. 30 (1983), pp. 657-667.
7. M. Marden, *The Geometry of the Zeros of a Polynomial in a Complex Variable*. American Mathematical Society, 1949.
8. K. Hwang and F. Briggs, *Computer Architecture and Parallel Processing*. Wiley, 1985.
9. W. Dorn, *Generalizations of Horner's rule for polynomial evaluation*. IBM J. Res. Dev., Vol. 6 (1962), pp. 239-245.
10. Y. Muraoka, *Parallelism exposure and exploitation in programs*. Report No. 424, Department of Computer Science, University of Illinois, Urbana-Champaign, IL, 1971.
11. D. Knuth, *The Art of Computer Programming*. Volume 2: Seminumerical Algorithms. Addison-Wesley, 1968.
12. K. Maruyama, *Parallel methods and bounds of evaluating polynomials*. Report No. 427, Department of Computer Science, University of Illinois, Urbana-Champaign, IL, 1971.
13. I. Munro and M. Paterson, *Optimal algorithms for parallel polynomial evaluation*. J. Comp. System Sci., Vol. 7 (1973), pp. 189-198.
14. D. Heller, *On the efficient computation of recurrence relations*. ICASE, Hampton, VA; Department of Computer Science, Carnegie-Mellon University, Pittsburgh, PA, 1974.

15. K. Hwang and Y. Cheng, *VLSI Computing Structures for Solving Large Scale Linear Systems of Equations*. Internal Conf. Par. Proc. 1980, pp. 217-227.

VITA

Ghassan Bakdash was born to Mr. and Mrs. Bakdash in August 26, 1961, in Beirut, Lebanon. After graduating from the Omar High School in 1979, he enrolled in the American University of Beirut, and obtained a B.S. in Electrical Engineering in 1983. During his study at the American University of Beirut, he received the Douk memorial scholarship in view of his academic achievements. He joined the Computer Science and Electrical Engineering Department at Lehigh University in 1984.

After his graduation in January, 1987 with a Master of Science degree, he would be joining AT&T Bell Laboratories in Naperville, Illinois, to work on enhancing Bell Laboratories' Electronic Switching System (5ESS).

His interests include computer architecture, parallel processing and system software. He enjoys traveling, hiking, history and poetry.