Chapter 5
PARALLELIZATION OF PIAP

5.1 Introduction

Parallel processing has been a subject of interest for the computer scientists and has always fascinated them. In the context of today's technology, which has reached a limit where it seems impossible to gain higher performance from sequential machines, parallel processing is being accepted as an alternative architectural approach to overcome this technology barrier.

The problem of computing prime implicants for a prepositional formula in CNF is computationally intractable [Provan 90]. Except for certain specific cases, the algorithms to solve the problem are known to be exponential in nature. Hence, for reasonably big problem size, prime implicants computation may not be practical with the existing computational methods. Therefore, it is necessary to resort to high performance computing.

This chapter explores the possibility of developing a parallel algorithm for the present problem. As discussed in earlier chapters, the importance of prime implicants in RMS have been established by [Reiter 87, Inoue 90, Provan 90], and hence most of the attempts have been focused on developing an efficient and fast algorithm to compute prime implicants. However, there has not been any attempt so far to utilize parallel computing technique for this purpose. The new algorithm, PIAP proposed in this dissertation has the additional advantage of being parallelizable. The aim of the present study is to investigate different aspects of parallelization of PIAP.
The coarse-grain parallelism can be seen trivially due to the tree structure proposed in Chapter 4. The prime paths for different nodes at the same level of the tree can be computed simultaneously and independently. The medium-grain parallelism can be achieved when concatenation and subsumption operations are performed between sets of paths with respect to two sibling nodes of the same parent node. Finally, the fine-grain parallelism can be achieved when the subsumption between the sets of paths is carried out as vector computation. Thus, three different levels of parallelism are explored here and the suitable architecture for each of these is also investigated.

The principles of parallel computing, the parallel architectures, and earlier works which deal with propositional formula using parallel computers are briefly reviewed. In later sections, each of the levels of parallelization are dealt with separately. The hybrid architecture to solve the problem of computation of prime implicants is also proposed.

5.2 Review of Architecture

This section is concerned with a brief review of the parallel architecture and parallel algorithms. There are many ways of classifying machine architecture. The first classification of parallel computers is given by Flynn's [Flynn 66] taxonomy in 1966. In this classification, multiprocessors are distinguished according to multiplicity of instructions and data streams. Though this classification is not comprehensive for parallel processors today, it spans the complete spectrum of the processor organization. According to this classification, there are four categories of parallel processors.

SISD- Class of computations with Single Instruction stream and Single Data stream.

SIMD- Class of computations with Single Instruction stream and Multiple Data stream.

MISD- Class of computations with Multiple Instruction stream and Single Data stream.
MIMD- Class of computations with Multiple Instruction stream and Multiple Data stream.

Flynn's classification does not cover all aspects of interconnection in the parallel processor. However, it is general enough for the design of many parallel algorithms. Generally, SIMD and MIMD classes of parallel computers are relevant in the present context. SIMD computation which exhibits data parallelism may be employed when the data is regular and calculation tends to be uniform. MIMD machine further falls into two categories; shared memory and distributed memory machines, though many architectures exhibit both parallelism. Finally, there are systolic arrays, pipe-line computation, mesh-connected arrays, hypercube network, shuffle exchange method, and partitionable and hierarchical multiprocessor architecture [Hwang 89] that have been proposed and studied for varieties of applications.

Pipeline Computation

Pipelining is one of the most primitive forms of synchronous computation. It consists of processors connected in a certain fashion by which on each machine cycle each processor receives data from its input ports, performs the required computations and passes the result and data through its output ports. Once the pipeline is filled, all the processing elements operate in parallel and one output is produced per cycle. Therefore, pipeline parallelism is effective for handling batches of data when operations on them can be broken down into distinct suboperations.

Shared Memory

Shared memory systems share a common block of the memory, and the processor cannot be used as a stand alone computer since the memory is shared. High interaction between various processing units and fast data communication is feasible in shared memory.
multiprocessors. In a shared memory SIMD computer, data communication takes place through shared memory location. In this, information flowing from a processing unit to a memory module contains data, address and other memory controls. In a local memory computer, information exchange between two processing units is through discrete messages which flow on the network wires in either serial or parallel order. The processing units in a MIMD computer interact with each other to solve a problem collectively and the computer organization is such that the interaction between various processing units is high and the data communication is fast. These are highly coupled multiprocessors and an interaction of this type is only possible with multiprocessors using the concept of shared memory. An important issue in shared-memory systems is data synchronization. That is, processors must synchronize with each other to avoid write conflicts, and to enforce the data precedence relationship inherent in the algorithm. The synchronization between processors in a shared memory multiprocessor is difficult and the basic mechanism of synchronization in such type of multiprocessors is by setting a lock, either in hardware or in software, before changing a critical variable shared by two or more processors.

Vector Processor

Vector processors are useful when an identical function is repeated many times for different data values, namely the elements of vectors. The efficiency of vector processing is primarily determined by the way the vector or matrix is handled. The vector processing can be achieved through a pipeline where a set of data is computed one after another in a pipeline mode, or through array processing where all data items can be computed in parallel.

A vector operand contains an ordered set of $n$ elements, where $n$ is the length of the vector. Four basic types [Ghosh 95] of primitive vector processing instructions are:
1. Those that operate on one vector and produce another vector. The input to these instructions is, therefore, only one vector.

2. Those that operate on one vector to produce a scalar. The input to these instructions is only one vector and the output is a scalar.

3. Those that operate on one vector and one scalar to produce one vector output.

4. Those that operate on two vectors to produce one vector.

The vector processing required for the computation of prime paths is of the fourth primitive type, where two vectors are given as input to produce one vector output.

A generic vector processor requires two input vector streams and provides one output vector stream. For the vector processor to work, two inputs must be simultaneously present on the input ports. Vector processor performs the designated operation on the vector elements and sends the result through the output port.

5.3 Design of Parallel Algorithms

The designing of efficient parallel algorithms must be guided by the architectural organizations that may support at a particular time. However, for a generalized approach, the extent of theoretical studies on parallelism exhibited by a problem should be far more developed to absorb any new architectural innovations that augment the technology of parallel computation. The emphasis of theoretical studies lies in extracting inherent parallelism of a problem. This study results in designing efficient parallel algorithms taking into consideration one of the following.

1. Mapping an existing sequential algorithm to a suitable parallel processor.

2. Designing afresh a new algorithm exploiting fully the problem's inherent characteristics.
It is observed in the approach of mapping a sequential algorithm to a suitable parallel machine that the best sequential algorithm need not be the best parallel algorithm. In the present context, the new algorithm, PIAP, is superior to other known algorithms in sequential computing, and is also suitable for parallelization. In this context, it is worthwhile to study the different parameters of the nature of parallelism.

Nature of Parallelism: Nature of parallelism has a number of attributes that depend on what kind of parallelism is used and the way in which the algorithms and/or data can be decomposed [Jamieson 88]. The following are some important parallelisms.

1. Data parallelism and function parallelism.

2. Data granularity.

3. Module granularity.

4. Degree of parallelism.

5. Uniformity of operations.


7. Static and Dynamic characters of the algorithm.

8. Data dependencies.

The most widely used theoretical model for parallel algorithms is parallel random access machine (PRAM). PRAM skirts the communication overheads entirely by enforcing communication through shared memory wherein the access is allowed to be made in a transparent manner simultaneously by all processors as long as they avoid clash for a specific location of the memory. Even the clashes or memory conflicts are allowed under some predetermined protocol which the designer of the algorithm may have to tackle.
5.4 Motivation and Earlier Work

The problem of computing prime implicants of a prepositional formula is NP-hard. So, in any real life application, when the number of prime implicants is very large, it would be difficult to compute the prime implicants in acceptable time. Therefore, it may be necessary to resort to multiprocessor computing to solve this problem. Since PIAP has an inherent tree structure, mapping the algorithm to a parallel computer becomes easy. Since the set of clauses representing the formula is successively divided into subsets and the paths of the subsets are concatenated to get the paths of the original formula, it is natural to process the subsets simultaneously. Thus different nodes of the tree can be processed in a parallel computer. On the other hand, when the paths of two nodes are combined to get the paths of the common parent node, the concatenation process for different paths can also be done in parallel. Every path consists of a set of literals and the subsumption operation between two paths is essentially comparing the clause set of the respective paths. This comparison can also be accomplished for each pair of clauses in parallel. Thus, PIAP facilitates multiple levels of parallelism,

- when the problem is subdivided into subproblems
- at the intermediate level (at each intermediate node of the tree)
- at the primitive data, namely, literals.

One can term all these as parallelization of different granularities-coarse-grain, medium-grain and fine-grain. In the following sections, suitable architectures for each of these levels of parallelism are explored and a suitable hybrid architecture for PIAP is proposed to solve the prime implicants problem.

There have been some attempts earlier to handle theorem proving, logic programming and ATMS [Rothberg 89] by parallel computers. But so far, no attempt has been made
to device parallel algorithm to compute prime implicants. The present study is the first of its kind and can also be extended for parallel RMSs. The earlier work relevant to the present study is briefly reviewed here.

Earlier work

Any theorem proving problem involves the combinatorial exploration of a solution space. One of the theorem proving procedures that appears to be particularly efficient in many cases is the Davis-Putnam procedure (DPP) [Davis 60]. Based on the DPP, Chen and Liu [Chen 87] proposed a parallel approach to decompose a formula. At each iteration, a variable is chosen arbitrarily and the formula is split into two subformulae with respect to the variable. Based on this approach, a divide-and-conquer strategy together with a pipeline discipline has been proposed for theorem proving in prepositional logic.

In order to fully minimize the vectorization techniques, Fang and Chen [Fang 92] generalize the rules by considering more than one variable at a time. Then, for efficient vectorization, a vectorized representation of a clause is given. Finally, vectorization techniques are utilized to deal with the generalized rules so that they can be implemented in terms of vector instructions. It is shown that the approach is effective in a sense that most operations involved in the algorithm are simple operations like AND, OR, and MERGE bit vector instructions, which are most efficient on vector computers.

5.5 Different Levels of Parallelism in PIAP

The major goal in characterizing the algorithm is to identify and exploit its inherent parallelism (i.e., potential for concurrency). The levels of resolution at which we can attempt to find this parallelism are coarse-grain, medium-grain and fine-grain. The levels of granularity in PIAP is the subject of discussion in this section.
Chapter 5,  \textit{PARALLELIZATION OF PIAP}

5.5.1 Coarse-grain parallelism in PIAP

The architecture at this level of parallelism is a general purpose, shared memory parallel processing system. In such a shared memory system, the binary tree discussed in Chapter 4 can be made available to all processors. The tree is partitioned in such a way that each processor does approximately the same amount of work, in order to balance the work load. This task can be accomplished by a judicious choice of the literal $r$ at every step of tree partitioning. However, a perfect balance of work load may not be possible in practice. In the coarse-grain-level of parallelization, PIAP works as follows:

During each iteration $k$, all prime paths of nodes at level $k$ are computed. The $k$ ranges from 'depth' (depth of the tree) to 0. Computation of prime paths of each node at level $k$ of the tree is independent, and is computed by each processor, assuming that there are as many number of processors as there are nodes at level $k$. In such cases, once the computation of one node is over, the processor corresponding to that node is idle. If there are only $m$ processors available and the maximum number of nodes at a level is $n$, different cases arise; (1). $m \geq n$ and (2). $m < n$.

Case 1. In this case, $n$ nodes at the level $k$ is given to $n$ processors and the remaining $m-n$ processors are idle. Since the number of processors exceeds the number of nodes at the level, each of the node is given to different processors.

Case 2. In this case, since the maximum number of nodes at the particular level exceeds the number of processors, we find the level which has number of nodes less than or equal to $m$. At this level, the nodes are given to $m$ processors, and the computation of paths upto this level is performed sequentially.

The entire tree is available to all the processors through the shared-memory. Each of the processors performs its computations and the result is stored in the common memory.
The processors which need the computed data collect the data from the common memory. The $m$ processors are assigned to it by a wrap-mapping scheme so that each one has approximately the same amount of work. After the computation of prime paths at each level is complete, the processors must synchronize with each other before they can proceed to the next level. Since $m$ is never very large, and since partitioning and synchronization may be expensive, it is proposed to switch to a one processor sequential phase where there are fewer than $m$ nodes.

The parallel algorithm PARPIAP to compute the prime paths for Case 1 ($m \geq n$) can formally be stated as given below. The function $\text{nodepath}(r, L_i(r), R_i(r))$ in the algorithm PARPIAP computes the prime paths of the node $N_i$, having label $r$ and $L_i(r)$ and $R_i(r)$ as the left and right child, respectively.

**Algorithm PARPIAP**

(Number of processors is assumed to be $m$)

begin { initialization }

For $d = \text{depth}$ to 0 do in parallel

for each node at level $d$ do in parallel

assign the node $N_i$ to processor $P_i$

for each processor $P_i$ do in parallel

collect the label $r$, prime paths of $L_i(r)$ and $R_i(r)$

$\text{nodepath}(r, L_i(r), R_i(r))$;

write paths of node $N_i$, to the common memory

end

For case 2 ($m < n$), $d$ ranges from $k$ to 0 instead of $\text{depth}$ to 0 where $k$ is the level of the tree where the number of nodes is less than or equal to $m$. 

5.5.2 Medium-grain parallelism in PIAP

At the medium-grain parallelism, each of the processors $P_i$ handles a set of paths. The function of these processors is to collect the label $r$ of the node and the paths generated by both the child nodes, check for subsumption among the set of paths and concatenate the set of paths with the literal $r$ and with another set of paths. Since the paths are independent, their concatenation with a literal, subsumption check between pair of paths, and the concatenation of paths can be carried out simultaneously by these processors. Hence, each of these processors is visualized as a set of processors $(V_1, V_2, ..., V_{m_1})$ so as to handle the paths simultaneously. Such a processor is depicted in Figure 5.1.

Concatenation of literal $r$ to a path $p$

The Step 5 of PIAP is to concatenate a literal $r$ to the prime paths in $P[S — S_r, T U \{r\}]$. Let $p_1, p_2, ..., p_{m_1}$ be the prime paths in $P[S — S_r, T U \{r\}]$. As mentioned above, since these prime paths are independent, the concatenation of these paths with the literal $r$ can be performed independently and simultaneously.

It is assumed that there is a host which distribute to each of these processors $V_i$ a path $p$, in $P[S — S_r, T U \{r\}]$. It is also assumed that the number of processors is large enough to handle all the paths simultaneously with one path residing at one processor. However, necessary modification can be done trivially when the number of processors is less than the number of paths. For concatenation of literal $r$ with the paths $P_i$ ($i = 1, 2, ..., m_1$), $r$ is given to all the processors at a time and then each processor after checking fundamentality returns $p_i \cup \{r\}$ to the host. This is pictorially depicted in Figure 5.1. If each processor takes unit time to perform this, then $m_1$ concatenations are performed in unit time since there are $m_1$ processor. If the number of vector processors is less than the number of paths in $P[S — S_r, T U \{r\}]$, then one set of $m_1$ paths arc
Chapter 5. PARALLELIZATION OF PIAP

passed onto the processors first, and then another set of $m_1$ paths and so on till the concatenation with all paths are over. If there are $n$ prime paths in $P[S - S_r, T U \{r\}]$ and $m_1$ processors, then the concatenation of $r$ with these paths can be performed in $\lceil n/m_1 \rceil$ units of time.

Figure 5.1: Concatenation of $r$ with $p_i$, $i = 1, 2, ..., m_1$
using $m_i$ vector processors.

Concatenation, $p_i \cup q_j$ of two paths $p_i \in P[S - S_r, T U \{r\}]$ and $q_j \in P[S_r, T U \{r\}]$ can also be performed in the same manner. Let $q_1, q_2, ..., q_{m_2}$ be the prime paths in $P[S_r, T U \{r\}]$. The host distributes to each of the processors a path $p_i$. The $q_i$'s are send to the processors through the input port in a pipeline fashion by the host.

The paths $q_j$'s enter with a time lag of one cycle between consecutive paths. A cycle is equal to the time taken by a processor to concatenate two paths $p_i$ and $q_j$ and to output the resulting path $p_i \cup q_j$ at its output port to the host. This is illustrated in Figure 5.2.

The problem of subsumption is essentially to check whether $p_i \subseteq q_j$ or $q_j \subseteq p_i$. Subsumption is handled in a the same manner as concatenation. Instead of returning $p_i \cup q_j$ at every instance of time, every processor returns 1, if $p_i$ subsumes $q_j$, —1, if $p_i$ is subsumed by $q_j$, and 0, otherwise. Once when one cycle of computation is over, the host checks the results obtained and acts accordingly. If any output (say, $i^{th}$) is 1, then the $i^{th}$ path $p_i$ subsumes the $q_j$ and hence $q_j$ is deleted from the set of paths by making all the components of $q_j$ a positive value bigger than 2 (say, 3). On the other hand, if any
output (say, $j^{th}$) is $-1$, then the $j^{th}$ path $P_j$ is subsumed by some path $q_i$ and hence $P_j$ is deleted from the set of paths by making all components of $P_j$ a positive value bigger than 2 (say, 3). A value bigger than 2 is given so as to make sure that this path does not play any role in deciding the subsumption of other paths.

Figure 5.2: Concatenation of $p_i \forall j = 1, 2, \ldots, m_1$ for $j = 1, 2, \ldots, m_2$ using $m_1$ vector processors.

5.5.3 Fine-grain parallelism in PIAP

At the finest level of granularity, the paths are visualized as set of literals and parallel implementations of the concatenation and subsumption operation are performed. In order to accomplish these, let us assume that some ordering of literals exists. Without loss of generality we also assume that all the paths follow this ordering of literals. The paths are represented in the form, of a binary vector with 1 if the literal is present and 0, otherwise. The number of processing elements (PEs) in these vector computers is assumed to be equal to the number of literals. Concatenation and subsumption are handled as follows:

Concatenation in parallel

Let $p_i = \{a_{i1}, a_{i2}, \ldots, a_{ik}\}$ and $q_j = \{b_{j1}, b_{j2}, \ldots, b_{jk}\}$. In order to perform concatenation of $p_i$ and $q_j$, each PE computes the maximum of $a_{ik}$ and $b_{jk}$ for all $k$. The result is sent to
the host through the output port. The host on receiving this, checks for fundamentally and sends it to the common memory if it is fundamental, and discards it, otherwise.

Subsumption in parallel

In order to accomplish subsumption, each processor computes $a_{ik} - b_{jk}$ for all $k$. Since $p_i$ and $q_j$ are binary vectors, $a_{ik} - b_{jk}$ has the values 0, +1, or -1 for all $k$. The host of these processors checks $a_{ik} - b_{jk}$ for all $k$. If $a_{ik} - b_{jk}$ is either 0 or -1 for all $k$, then $p_i$ subsumes $q_j$, and if $a_{ik} - b_{jk}$ is either 0 or 1 for all $k$, then $q_j$ subsumes $p_i$, and if $a_{ik} - b_{jk}$ is 0 for all $k$, then both $p_i$ and $q_j$ are the same and either of them is considered to subsume the other. If $a_{ik} - b_{jk}$ has the value 0, +1, and -1 for some $k$, then $p_i$ and $q_j$ are entirely two different clauses which do not subsume each other at all. The host on receiving the values checks values it received. If some $q_j$ subsumes $p_i$, then $p_i$ is deleted from the set of prime paths $P[S - S_r,T(r)]$ and the elements in the next path are distributed to the PEs. If $P_i$ subsumes $q_j$, then the $q_j$ is given a flag and is not considered further. Based on these discussions, the working of the parallel algorithm and implementation are explained in the following section.

![Diagram](https://via.placeholder.com/150)

**Figure 5.3:** Computing $a_{ik} - b_{jk}$, for $i = 1,2,...,m_1$ and $j = 1,2,...,m_2$ for subsumptions using $m_1$ vector processors.
5.6 Parallel Algorithm and Implementation

The three levels of parallelism in PIAP are discussed in Section 5.5. As mentioned earlier, the entire binary tree is available to all processors $Pi$ through shared memory. At the $d^{th}$ level of the tree, let us assume that there are $i$ nodes and correspondingly, there are $i$ processors. The $i^{th}$ processor, $Pi$ collects the data relevant to the $i^{th}$ node in the $d^{th}$ level, i.e., $Pi$ collects the label $r_i$, left child prime paths $P(L_i(r_i))$, and right child prime paths $P(R_i(r_i))$ of the $i^{th}$ node in the $d^{th}$ level. It is assumed that there are as many number of vector processors as there are paths in $P(L_i(r_i))$. The $j^{th}$ path $p_j$ of $P(L_i(r_i))$ is allocated to the $j^{th}$ vector processor $V_j$. As already mentioned, it is assumed that the length of the vector representing any path is equal to the number of literals in $\mathcal{E}$ (the set of literals). In $V_j$, the $k^{th}$ component in $p_j$ is allocated to the $k^{th}$ PE of $V_j$. The entire architecture is pictorially depicted in Figure 5.4.

The parallel algorithm using this architecture for the example with tree structure given in Figure 4.5 is explained here. At level 2 of the tree there are four nodes, of which the first node is NULL. The remaining three nodes (nodes 5, 6 and 7) are assigned to three different processors $P_1, P_2$ and $P_3$, respectively. $P_1$ collects the label $c$ and the prime paths of its left child and right child. Similarly, processors $P_2$ and $P_3$ also collects the data required for nodes 6 and 7, respectively. Left child of node 5 is NULL and hence $P_1$ writes the prime paths for the node 5 as $\{c\}$ (the label), and $\{a\}, \{f, g\}$ (the right child prime paths of node 5 into the shared memory). The processor $P_2$ has two paths $d = (0001000000000)$ and $f = (0000000000100)$ in $P(L_2(g))$ and paths $d = (0000010000000)$ and $f = (0000000000010)$ in $P(R_2(g))$. The two paths in $P(L_2(g))$ are distributed to two processors; $b$ to $V_1$ and $f$ to $V_2$. The path $d$ in $P(R_2(g))$ is sent to $V_1$. 


and $V_2$ simultaneously. Thus, $V_1$ has the vectors $(0001000000000)$ and $(0000001000000)$.  

![The hybrid architecture for PARPIAP](image)

Figure 5.4: The hybrid architecture for PARPIAP

In order to perform the subsumption operation on these data, the components of these vectors are given to each of the PEs of $V_1$. The first PE $PE_1$ has $a_{i1} = 0$ and $b_{i1} = 0$; $PE_2$ has 0 and 0, and so on. Each of the processors compute $a_k - b_k$ and
Chapter 5. **PARALLELIZATION OF PIAP**

$V_1$ obtains the vector (000100-100000). This vector has 0, 1 and -1, and hence neither of them subsumes nor is subsumed. Similarly the paths in $V_2$ also neither subsumes nor is subsumed by any path. Hence, the host of the processor $P_2$ sent $g$ to $V_1$ and $V_2$, simultaneously so as to perform the concatenation operation. In order to perform the concatenation, $V_1$ has vectors (0001000000000000000) and (0000000000000000001). The concatenated path obtained ($\{b, g\} = (0001000000000001))$ is sent to the common memory. Similarly $\{f, g\}$ is sent to the common memory by $V_2$.

In order to perform the concatenation operation of the form plsl $q$ where $p \in P(L_2(\bar{g}))$ and $q \in P(R_2(\bar{g}))$, $\{d\}$ and $\{f\}$ are sent to both $V_1$ and $V_2$ one after the other and the result obtained is stored in the local memory of $P_2$. Subsumption operation is performed on the paths ($\{b, d\}$, $\{d, f\}$ and $\{b, f\}$) thus obtained. For this, the first path (say, $\{b, d\}$) is passed to all the vector processors in $P_2$. The other paths are sent to these processors one after the other for Subsumption check. The results sent by these vector processors are stored in the local memory of the processor. The unsubsumed paths are sent to the common memory. The processor $P_3$ also works in a similar fashion for the node 7. Once all the three processors have written the results in the common memory, the algorithm proceeds to the next lower level.

5.7 Conclusion

An attempt is made to explore the inherent parallelism in PIAP, and a parallel algorithm PARPIAP, the parallel version of PIAP, to compute the prime implicants of a formula is designed. The different levels of granularities—coarse-grain, medium-grain and fine-grain are explained. Coarse-grain parallelism is trivial due to the divide-and-conquer paradigm. The prime paths of two subsets of clauses are computed at this level of granularity. The prime paths thus obtained are concatenated followed by subsumption...
check. The concatenation of paths as well as the concatenation of paths with a literal with respect to which the set of clauses is partitioned are performed simultaneously. The subsamption check is also performed simultaneously. These computations exhibit medium-grain parallelism. Finally, fine-grain parallelism is achieved when each of the path is considered a vector, and the components of the path obtained as a result of concatenation are computed simultaneously using vector processors.

The hybrid architecture suitable for the parallel computation of prime paths is proposed. The proposed algorithm will help in designing a parallel knowledge compilation technique which in turn can be an efficient tool for RMS.

This algorithm has the synchronization cost of one synchronization per level of the tree. Judicious choice of the literal \( r \) would help in obtaining a balanced tree and hence proper load balancing can take place. However, even for a balanced tree, the number of prime paths at each node need not be the same and hence, proper load balancing is difficult to achieve. Moreover, if the number of vector processors in the processors is fixed, there may not be that many prime paths in the left child of the node. Hence there may be idle vector processors.