The Strict Time Lower Bound and Optimal Schedules for Parallel Prefix with Resource Constraints*

Haigeng Wang Alexandru Nicolau
Department of Information and Computer Science
Kai-Yeung S. Siu
Department of Electrical and Computer Engineering
University of California, Irvine
Irvine, CA 92717-3425

Technical Report 92-114, December 1992

The Strict Time Lower Bound and Optimal Schedules for Parallel Prefix with Resource Constraints *

Haigeng Wang Alexandru Nicolau
Department of Information and Computer Science
Kai-Yeung S. Siu
Department of Electrical and Computer Engineering
University of California, Irvine
Irvine, CA 92717-3425

Technical Report 92-114, December 1992

Abstract

Parallel prefix is a fundamental common operation at the core of many important applications, e.g., the Grand Challenge problems, circuit design, digital signal processing, graph optimizations, and computational geometry. Given $x_1, \ldots, x_N$, parallel prefix computes $x_1 \circ x_2 \circ \ldots \circ x_k$, for $1 \leq k \leq N$, with associative operation $\circ$. For prefix of N elements on p processors in $N > p(p+1)/2$, we derive *Harmonic Schedules* and show that the Harmonic Schedules achieve the *strict optimal time* (steps), $\lceil 2(N-1)/(p+1) \rceil$. We also derived *Pipelined Schedules*, optimal schedules with $\lceil 2(N-1)/(p+1) \rceil + \lceil (p-1)/2 \rceil - 1$ time, which take a constant overhead of $\lceil (p-1)/2 \rceil$ time steps more than the strict optimal time but have the smallest loop body. Both the Harmonic Schedules and the Pipelined Schedules are simple, concise, with nice patterns of computation organizations, and easy to program. For prefix of N elements on p processors in $N \leq p(p+1)/2$, we use an algorithm to construct schedules. For $N = \sum_{i=0}^{k} \binom{m}{i}$ and $p = \sum_{i=1}^{k} \binom{m-1}{i-1}$, which are a class of values of N and p in $N \leq p(p+1)/2$, we derive strict time-optimal schedules and show that they compute a prefix of size N on p processors in m steps. For other values of N and p in $N \leq p(p+1)/2$, we give strong empirical evidence that the same algorithm can also create the strict time-optimal schedules. The parallel schedules operate on the concurrent-read-exclusive-write(CREW) parallel random access machine(PRAM) model.

Index Terms— Parallel prefix computation, resource-constrained parallel algorithms, strict time-optimal schedules, loop parallelization, loop-carried dependences, associative operations, tree-height reduction, Pascal Triangle, combinatorial optimization.

1 Introduction

Given $x_1, \ldots, x_N$, parallel prefix computes $x_1 \circ x_2 \circ \ldots \circ x_k$, for $1 \leq k \leq N$, with associative operation $\circ$. Prefix sums(or the first-order linear recurrence with coefficients 1) is a major special case of prefix computation that can be stated as follows ($x[0] = 0$),

*This work is supported in part by NSF grant CCR8704367 and ONR grant N0001486K0215.

$do\ \ i = 1, N$

$\quad a[i] = x[i-1] + a[i].$

We refer to our problem interchangeably as the recurrence, prefix sums or prefix computation, since the results on prefix sums can be readily applied to prefix computation with other associative operations.

Prefix computation is a fundamental common operation at the core computations of many important applications, e.g., the Grand Challenge problems, circuit design, digital signal processing, graph optimizations, and computational geometry[1, 8, 13]. One of its most important applications is loop parallelization. Traditional automatic loop parallelization techniques[5] respect loop-carried dependences, thus unable to generate scalable parallel code from loops containing loop-carried dependences. To understand how to parallelize loops with loop-carried dependence beyond these techniques, it is essential to understand the simplest case of loops with loop-carried dependence, namely prefix sums. The optimal schedules and the technique used to derive them in this paper can be applied—with some extensions— to parallelizing many sequential algorithms containing loop carried dependences.

Since in practical applications the number of resources (i.e., functional units, processors) is fixed and *independent of* the problem size, it is critical to devise a scheme which performs the computation in optimal time with a given set of *resource constraints*. Since we are interested in extending the technique to handle more general forms of loops with loop-carried true dependence, we are concerned with other properties of these optimal schedules, such as the clarity, simplicity of implementation, and extendibility.

We assume the parallel random access machine(PRAM) model[13]. A PRAM consists of p autonomous processors, executing synchronously, all having access to a globally shared memory. Our PRAM operate in concurrent-read-exclusive-write(CREW) mode: multiple processors may read simultaneously from a memory location (i.e., broadcasting data) but exclusive access is required for write to the same memory location.

We now define the terms frequently used in this paper. A *schedule* A is used to perform a computation[1]. We denote the time to run schedule A on our machine as $T_p(A)$, or T_p and T when unambiguous, where p refers to the number of processors in the machine. We refer to T_p and T interchangeably as time, time steps or steps. The time to compute A sequentially is denoted $T_1(A)$. The *speedup* of a machine with p processors over a uniprocessor, for schedule A, is denoted $S_p(A) = T_1(A)/T_p(A)$, or simply $S_p = T_1/T_p$ when unambiguous. The *efficiency* of this computation is $E_p = S_p/p$, which can be interpreted as actual speedup divided by the maximum possible speedup using p processors. Let O_p be the number of operations executed in some computation using p processors. We define *operation redundancy* to be $R_p = O_p/O_1 (\geq 1)$, where $O_1 = T_1$. Finally, we define *utilization* as $U_p = O_p/pT_p \leq 1$, where pT_p is the maximum number of operations that p processors can perform in T_p steps. The *final values* of prefix sums are the values computed by the definition of prefix sums. The *final operations* of prefix sums are operations that assign the final values. The *intermediate operations* are operations that compute auxiliary values in an effort to speed up the final value computation. The *intermediate values* refer to the auxiliary values computed by intermediate operations.

In this paper, we establish the connections between the organization of the prefix computation and the Pascal Triangle, and thus ground the optimal schedules for the prefix computation on combinatorial optimization. Based on this finding, the minimum time T required to compute a prefix problem of size N on $p > 1$ processors can be determined from the following system of inequalities.

$$\sum_{i=1}^{k-1} \binom{T}{i} < N - 1 \leq \sum_{i=1}^{k} \binom{T}{i}, \tag{1}$$

$$0 \leq pT - \left[\sum_{i=1}^{k-1} \binom{T}{i} i + kl \right] < p,$$

[1] We distinguish between our *algorithm* and *schedule* —our algorithm is used to produce a schedule(for the given prefix computation) which, when run on the PRAM, will execute the actual prefix computation.

where $l = N - 1 - \sum_{i=1}^{k-1} \binom{T}{i}$, $1 \leq k \leq T$, $1 \leq T < N$. We derive a class of optimal schedules(we define schedule at the end of this section), called *Harmonic Schedules*, for prefix problem of size $N \geq p(p+1)/2+1$ on $p \geq 2$ processors. The Harmonic Schedules achieve an execution time of $\lceil 2(N-1)/(p+1) \rceil$ for $N \geq p(p+1)/2$ on p processors, which we show to be the strict time lower bound. We also present a pipelined form of optimal schedules with $\lceil 2(N-1)/(p+1) \rceil + \lceil (p-1)/2 \rceil - 1$ time, which takes a constant overhead of $\lceil (p-1)/2 \rceil$ time more than the optimal schedules. The pipelined schedules are simpler, more concise and more program-space efficient than the Harmonic Schedules. We establish our algorithm for finding schedules for prefix computations of size N on p processors for $N \leq p(p+1)/2$. We conjecture that our schedules for $N \leq p(p+1)/2$ also achieve the strict optimal time and we show strong empirical evidence for our conjecture.

To facilitate understanding, our paper is organized in the order in which the findings were made. Section 2 reviews related work. Section 3 presents the Harmonic Schedule. Section 4 shows the optimality of the Harmonic Schedule and its properties. Section 5 gives the Pipelined Schedule, a simpler, more concise and more program-space efficient schedule than the Harmonic Schedule. Section 6 finds the connections between the organization of the prefix computation and the Pascal Triangle, and formulates the problem of finding optimal schedules for prefix computations using a system of inequalities. Section 7 establishes an algorithm for finding schedules for prefix computations of $N \leq p(p+1)/2$, which can also be used to derive Harmonic Schedules. Section 8 proves the existence of the schedules constructed using our algorithm in Section 7 and provides evidence for our conjecture that these schedules for prefix computations in $N \leq p(p+1)/2$ are strict time-optimal. Section 9 summarizes our results.

2 Related Work

The results in parallel prefix computation have been in two categories: with no resource constraints, where usually the number of processor p is a function of the problem size N, and with resource constraints where p is independent of N.

Special forms of the prefix circuits had been previously known to Ofman as early as 1963, whose carry circuit[17] is a form of the carry circuits later discussed by Ladner and Fischer [12]. Muraoka[15] showed that this simplest linear recurrence can be computed in $\log N$ with $N/2$ processors, and used the name *tree-height reduction* for this technique. Kogge and Stone's *recursive doubling*[11] for first-order linear recurrences is essentially equivalent to tree-height reduction when applied to prefix problem . Chen and Kuck[10] showed that linear recurrences can be computed in $1 + 2\log N$ with $p \leq N/2$. Ladner and Fischer[12] found a class of circuits for a prefix of size $N = 2^k$ for $k > 0$ assuming enough processors are available.

With unlimited resources, Fich[9] gave upper bounds and lower bounds on the circuit size(i.e., the number of gates or operations)for prefix circuits of input N with depth $\log N + d$, where $d \leq \lceil \log N \rceil$ is an extra depth, for most cases of unbounded and bounded fan-out. Fich constructed the circuits using a more complex recursive procedure than Ladner and Fischer's. With unbounded fan-out, the lower bounds for prefix circuits of input N were $(2 + 2^{1-d}/3)2^k - O(k^2)$ for $1 \leq d \leq m-1$, and $10/32^k - O(k^3)$ for $k \geq 0$. Upper bounds with fan-out bound $f \geq 2$ were also obtained. Finally, lower bounds with fan-out two were $k2^{k-1} + 2^{k-1} - O(k^2)$ and $(m-2)2^{k-d-1} + 2^{k-1} - O(k(k+d)2^{-d})$ for $k \geq 0$ and $d \geq 1$. Bilgory and Gajski[6] gave a different algorithm that generates suffix(the term they used) solutions with minimum cost for a given length n, depth d, initial value availability e, and fanout f. The *cost* is defined as the minimum number of operation nodes along a pair of corresponding input and output. A lower bound on the cost is therefore the maximum number of nodes on a path from an input to its corresponding output. They chose cost as the measurement instead of circuit size because it better fitted with their consideration for silicon layout.

Lastly, Cole and Vishkin[7] gave an algorithm that solves the prefix sums problem in $O(\log N/\log\log N)$ time using $N\log\log N/\log N$ processors, provided that each $a[i]$ is represented by $O(\log N)$ bits, which is different from the problem we address.

With resource constraints, a well-known algorithm that can also be found in [4], computes the prefix problem in $2N/p+\log p$ time. Given p processors, the algorithm divides the problem into p partitions, and "conquers" the local computation in each partitions with one processor, and "combines" the results. This algorithm takes $2N/(p(p+1))+\log p$ more steps than the optimal time.

Snir [18] gave an algorithm for dynamically constructing parallel prefix circuits with a fixed number of processors for CREW machines such that the depth of the resulting circuit for p processors is $2N/(p+1)+O(1)$ in $N \geq p(p+1)/2$, which is very close to strict time-optimal in depth. For a given problem size N, Snir's algorithm works to find the right partition of the problem in order to guarantee the depth of the resulting schedule, and must recompute the partition each time a new N is given. This is an overhead associated with problem size N at either compile time or run time. If N is known only at run time, then the overhead incurred by finding the right partition will have an impact on performance, which is another overhead at run time.

The Harmonic Schedule published in [16] achieved the strict time lower bound for $N \leq p(p+1)+1$, thus concluding the search for strict time-optimal schedules for prefix computations of size N on p processors for $N \leq p(p+1)+1$ under CREW PRAM model. Besides achieving the strict optimal time, the Harmonic Schedules have the nice property of regularity in the organization of computation which makes it easy to program. The problem of finding strict time-optimal schedules in $N \leq p(p+1)/2$ has remained open since the Harmonic Schedules.

This paper presents a new formulation of finding strict time-optimal schedules in terms of combinatorial optimization that unifies in a singl framework the optimal schedules for prefix problems for $N >$ and $N \leq p(p+1)/2$. Section 3, 4 and 5 of this paper are the significantly improved rewrite of [16]. In particular, the formalization and the proofs in Section 4 substantially clarify the initial results in [16]. Section 6, 7 and 8 of this paper will present our new unified framework and results for the remaining open problem.

3 The Harmonic Schedule

The idea of the Harmonic Schedule is to do all operations on multiple processors in a way that minimizes intermediate operations and achieves full utilization of all the processors. We know that the number of final operations in schedule H is $N-1$, which equals the number of operations in the sequential schedule and cannot be reduced— by the definition of required outputs. Thus the only way to speed up the computation is to use multiple processors to compute intermediate values ahead of the final value computation, and then to obtain multiple final values in a single parallel step by adding the last available final value and each of the multiple intermediate values in parallel. Thus we trade multiple processors and intermediate operations for speedup, or parallelism. The speed of a schedule is therefore the average number of final values produced in a step. In order to compute as many final values as possible, given a fixed number of processors, a schedule should do as few intermediate operations as possible. However, one cannot reduce the number of intermediate operations arbitrarily, say, to zero, because the schedule would degenerate to the sequential schedule and all but one processor would stay unused. So the fastest parallel schedules for a fixed number of processors would use the fewest possible intermediate operations to achieve full utilization of all processors. We shall prove in the next section that the Harmonic schedule satisfies this.

Prefix sums of size N can be computed by Harmonic Schedule, denoted H, described below. Let $p \geq 2$, $\delta = \sum_1^p$, and $\eta = \sum_1^{p-1}$. Let us assume *for the moment* that the array size $N = M\delta + 1$ for $M \geq 1$. The Harmonic Schedule consists of a loop body of p steps(see below). Each step consists of p operations that are

executed by the p processors in parallel. In the schedule, we use an auxiliary array t to store the "look ahead" intermediate values or "prefix values". Operations that assign results to array a or t are final operations and intermediate operations respectively.

The Harmonic Schedule

for $i = 1, \ldots, M$ *do each step in parallel with* p *processors*

step 1

$a[i\delta + 1] = a[i\delta] + a[i\delta + 1]$,

$t[i\eta + j] = a[i\delta + j(j+1)/2 + 1] + a[i\delta + j(j+1)/2 + 2]$,

$1 \leq j < p$.

step $k = 2, \ldots, p$

$a[i\delta + k(k-1)/2 + 1] =$

$= a[i\delta + k(k-1)/2] + a[i\delta + k(k-1)/2 + 1]$,

$a[i\delta + k(k-1)/2 + 1 + l] =$

$= a[i\delta + k(k-1)/2] + t[i\eta + k - 1 + (2p - 2 - l)(l-1)/2]$,

$1 \leq l < k$.

$t[i\eta + (2p-k)(k-1)/2 + j] =$

$= t[i\eta + (2p - k + 1)(k-2)/2 + j + 1] +$

$+ a[i\delta + (j+k)(j+k-1)/2 + k + 1]$,

$1 \leq j \leq (p-k)$.

end for

□

We can easily show that schedule H computes the given recurrence correctly. We can also easily verify that in schedule H, no memory location is written into simultaneously.

The generic H schedule above can be instantiated during compilation in negligible time(when the number of processors is known). Note that there is no run-time scheduling overhead found in [18], since the full instantiation of the schedule depends on p and does not depend on N. The array subscript calculation in an instantiated H schedule becomes very simple and can be reduced to addition using strength reduction[2]. Furthermore, the size of array t can be reduced to η since the contents of t produced in an iteration are only used in that same iteration.

Example We instantiate a Harmonic Schedule for $p = 7$ processors. We have $\delta = \sum_1^7 = 28$ and $\eta = \sum_1^6 = 21$. Figure 1 shows an iteration of the schedule. Figure 2 gives the fully instantiated schedule. The nodes on the top fringe in Figure 1 represent the final values of the prefix computed by the final operations. The inner nodes represent the intermediate values computed by the intermediate operations and stored in array t. The bottom ends of the vertical lines represent the given elements of the prefix sums. The inner nodes of height l in Figure 1 are computed by operations in the l-th step of the instantiated schedule in Figure 2. Note that all operations in each step execute in parallel. As expected, this schedule is optimal for $p = 7$ regardless of the value of N.

The first step of the iteration has one final operation and $p - 1$ intermediate operations, the second step has two final operations and $p - 2$ intermediate operations, and the k-th step has k final operations and $p - k$ intermediate operations. Each step does one more final operation and one fewer intermediate operation than the immediate preceding step. One of the nice properties of the H schedules is that the number of operations in an iteration is the sum of final operations and intermediate operations, $\delta + \eta = \sum_1^p + \sum_1^{p-1} = p^2$. We can see that all p processors are in full utilization in all p steps in the loop body and that final and intermediate computations "meet" at the end of each iteration. That is, at the end of each iteration all intermediate values are used in computing final values, hence the name "Harmonic Schedule". □

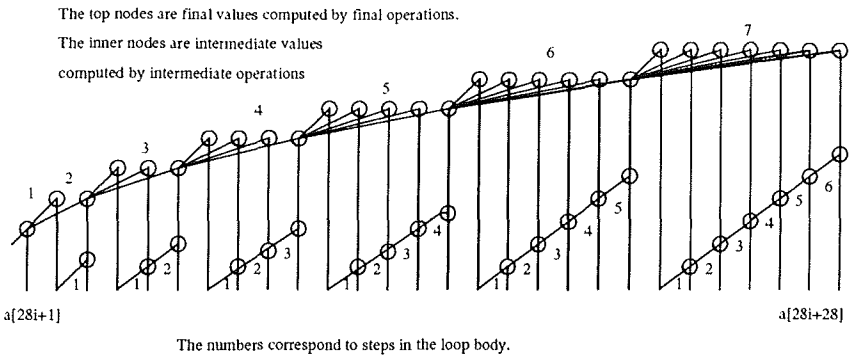


Figure 1: An Iteration of Schedule H for $p = 7$.

Theorem 3.1 For prefix sums of size $N = m\delta + 1, m \geq 1$ and $p \geq 2$ processors, the Harmonic Schedule computes the given prefix sums. It completes $N - 1 = m\delta$ final values in $mp = 2(N-1)/(p+1)$ steps(yielding a speedup $S_p(H) = (p+1)/2$) with a processor utilization $U_p(H) = 1$.

Proof: The correctness and utilization of schedule H are clear. We consider the time. Schedule H computes $m\delta$ final values in m iterations by the theorem's condition and the fact that δ final values are computed in each iteration. So,

$$\begin{aligned} T_p(H) &= (\#\ /\ \text{iterations})(\#\ \text{steps}\ /\ \text{iteration}) \\ &= \frac{(N-1)}{\delta}\frac{(\delta+\eta)}{p} \\ &= \frac{N-1}{\frac{p+1}{2}}. \end{aligned}$$

Therefore, $S_p(H) = (N-1)/T_p(H) = (p+1)/2$. □

So far, we have shown that schedule H computes $m\delta$ final values in $T = mp$ steps, i.e., some multiple of p steps. Next, we show that the performance stated in Theorem 3.1 is also true for any $N > p(p+1)/2$.

Theorem 3.2 (Extended Harmonic Schedule) Schedule H can be extended to compute the prefix sums of size $N \geq p(p+1)/2 (= \delta = \sum_1^p)$ in time $\lceil 2(N-1)/(p+1) \rceil$. Thus, the speedup is $(p+1)/2$.

Proof: The previous theorem proved the above statement for $N = m\sum_1^p + 1$, for $m \geq 1$. Suffice to show that the above statement is true for $N > \sum_1^p$, and $N \neq m\sum_1^p$ for $m \geq 1$. The proof proceeds in two cases.

Case 1: $p \geq 2$ is odd. Let $N = m\sum_1^p + 1 + q$, for $m \geq 1$ and $0 < q < \sum_1^p$. The last q elements are called the trailing elements. The final operations that compute the q trailing elements are called the trailing final operations, and the intermediate operations which values are used by the trailing final operations are called the trailing intermediate operations. Without loss of generality, let us look at the example for $p = 7$ in Figure 3.

In the $(mp+1)$-th step, we want to compute $(p+1)/2$ final operations and $(p-1)/2$ intermediate operations. How do we do it, given that an intermediate tree of height 3 takes exactly 3 steps to compute? The solution is to amortize the computation, i.e., to delay some of the final operations in steps $((m-1)p+1)$ through mp, and to use these freed processor time slots to compute the intermediate trees incurred by those q steps, and then to compute those delayed operations with the idle processor slots in steps $mp+1$ through $(m+1)p-1$. For example in Figure 3, the intermediate tree used in step 8 is computed in step 5, 6 and 7, which causes three

for $i = 0$ to $M - 1$ do

each step in parallel with p processors

1. $a[28i+1] = a[28i] + a[28i+1]$, $t[28i+1] = a[28i+2] + a[28i+3]$, $t[28i+2] = a[28i+4] + a[28i+5]$,
   $t[21i+3] = a[28i+7] + a[28i+8]$, $t[21i+4] = a[28i+11] + a[28i+12]$, $t[21i+5] = a[28i+16] + a[28i+17]$,
   $t[21i+6] = a[28i+22] + a[28i+23]$;
2. $a[28i+2] = a[28i+1] + a[28i+2]$, $a[28i+3] = a[28i+1] + t[21i+1]$, $t[21i+7] = t[21i+2] + a[28i+6]$,
   $t[21i+8] = t[21i+3] + a[28i+9]$, $t[21i+9] = t[21i+4] + a[28i+13]$, $t[21i+10] = t[21i+5] + a[28i+18]$,
   $t[21i+11] = t[21i+6] + a[28i+24]$;
3. $a[28i+4] = a[28i+3] + a[28i+4]$, $a[28i+5] = t[21i+2] + a[28i+3]$, $a[28i+6] = t[21i+7] + a[28i+3]$,
   $t[21i+12] = t[21i+8] + a[28i+10]$, $t[21i+13] = t[21i+9] + a[28i+14]$, $t[21i+14] = t[21i+10] + a[28i+19]$,
   $t[21i+15] = t[21i+11] + a[28i+25]$;
4. $a[28i+7] = a[28i+6] + a[28i+7]$, $a[28i+8] = t[21i+3] + a[28i+6]$, $a[28i+9] = t[21i+8] + a[28i+6]$,
   $a[28i+10] = t[21i+12] + a[28i+6]$, $t[21i+16] = t[21i+13] + a[28i+15]$, $t[21i+17] = t[21i+14] + a[28i+20]$,
   $t[21i+18] = t[21i+15] + a[28i+26]$;
5. $a[28i+11] = a[28i+10] + a[28i+11]$, $a[28i+12] = t[21i+4] + a[28i+10]$, $a[28i+13] = t[21i+9] + a[28i+10]$,
   $a[28i+14] = t[21i+13] + a[28i+10]$, $a[28i+15] = t[21i+16] + a[28i+10]$, $t[21i+19] = t[21i+17] + a[28i+21]$,
   $t[21i+20] = t[21i+18] + a[28i+27]$;
6. $a[28i+16] = a[28i+15] + a[28i+16]$, $a[28i+17] = t[21i+5] + a[28i+15]$, $a[28i+18] = t[21i+10] + a[28i+15]$,
   $a[28i+19] = t[21i+14] + a[28i+15]$, $a[28i+20] = t[21i+17] + a[28i+15]$, $a[28i+21] = t[21i+19] + a[28i+15]$,
   $t[21i+21] = t[21i+20] + a[28i+28]$;
7. $a[28i+22] = a[28i+21] + a[28i+22]$, $a[28i+23] = t[21i+6] + a[28i+21]$, $a[28i+24] = t[21i+11] + a[28i+21]$,
   $a[28i+25] = t[21i+15] + a[28i+21]$, $a[28i+26] = t[21i+18] + a[28i+21]$, $a[28i+27] = t[21i+20] + a[28i+21]$,
   $a[28i+28] = t[21i+21] + a[28i+21]$;

Figure 2: An instantiated schedule H for $p = 7$ processors.

final operations, one in each of step 5 to 7 to be delayed. Then, three idle slots in step 8 are used to compute the three delayed operations in step 5 through 7.

The question is whether there is a feasible amortizing scheme between the delayed operations and the idle processor time slots, i.e., whether there always are enough delayable operations to match the tailing intermediate operations in this fashion. Note that there are $\eta = \sum_1^{p-1}$ final operations in each iteration that can be delayed because no other operations depend on them. Hence there are enough delayable operations to match the trailing intermediate operations. In similar fashion, we can always find a feasible amortizing scheme for steps $mp + 1$ through $(m + 1)p - 1$. The above arguments are true for all odd $p \geq 2$. Therefore, the extended schedule H computes prefix sums of size $N \geq p(p + 1)/2$ in time $\lceil 2(N - 1)/(p + 1) \rceil$ on odd $p \geq 2$ processors.

Case 2: $p \geq 2$ is even. Let N be the same as for case 1. The amortizing scheme is shown in Figure 4. Note that for even p the heights of the trailing local trees are in turn $\lfloor (p+1)/2 \rfloor$ and $\lceil (p+1)/2 \rceil$ subject to the number of trailing elements, while for odd p the height of local trees is always $(p + 1)/2$ subject to the number of trailing elements. The rest of the proof is similar to case 1. Hence, the extended schedule H computes prefix sums of size $N > p(p + 1)/2$ in time $\lceil 2(N - 1)/(p + 1) \rceil$ on even $p \geq 2$ processors.

It can be seen from the proof above that the ratio of the number of final operations O_f to the number of intermediate operations O_i in schedule H is $O_f/O_i = (p + 1)/(p - 1)$. This property will be used later in deriving simpler, more concise and more program-space efficient schedules than H. □

By theorem 3.2, we can construct the extended part of schedule H as follows. We unwind the last iteration of the unextended schedule H, and use our amortizing scheme between the delayable operations in the last iteration and the idle processor slots in those q steps. Thus, we obtain a generic extended schedule H with $p - 1$ epilogues that handles all length of trailing steps for $1 \leq q < p$. Note that although q depends on N, the generic extended schedule H can be precomputed and installed as a library routine. When q is known, the

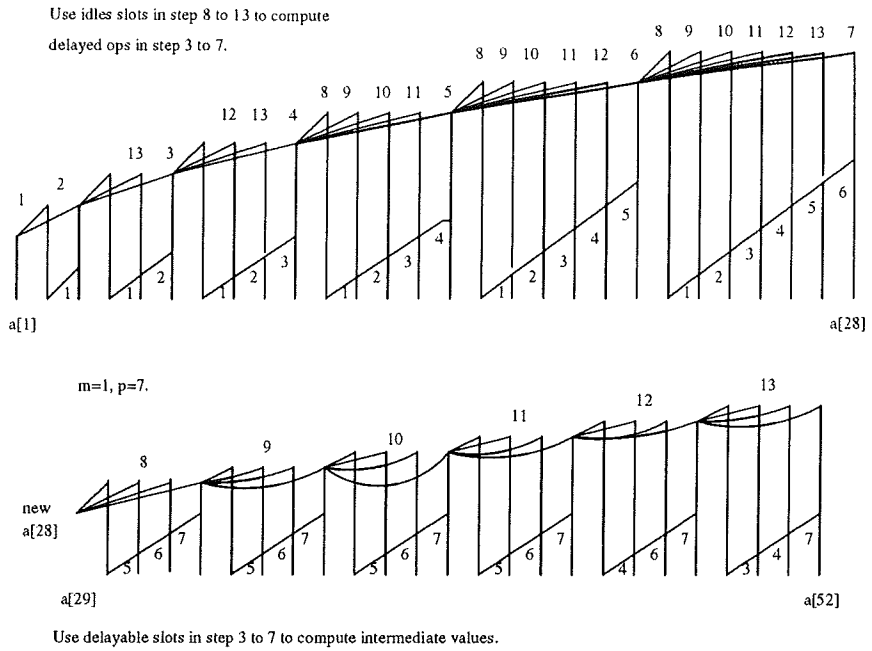


Figure 3: The amortizing scheme for extended schedule H with odd p.

corresponding epilogue will be chosen to run.

4 Optimality of the Harmonic Schedule

In this section we show that schedule H is optimal given a fixed number of processors $p \geq 2$ and prefix sums of size $N > p(p+1)/2$, meaning that schedule H gives the maximum possible speedup $(p+1)/2$ with p processors and uses the minimum possible number of processors for that speedup. Then we reveal an interesting property of the parallel prefix computation.

The proof proceeds as follows. Lemma 4.1 presents a lower bound on time $T_p(A)$ for parallel schedule A with p processors. Lemma 4.2 and 4.4 combined show that for all schedules A with $O_i(A) \leq O_i(H)$, $T_p(A) \geq T(H)$ for a fixed number of p processors. Theorem 4.1 proves the optimality of schedule H.

Consider the elements of the given prefix problem that are stored in array a. We call an element of array a a *redundantly scheduled element* if it is used in a redundant operation, otherwise a non-redundantly scheduled element. A set of redundant operations is said to form a *redundant computation cluster*(*redundant cluster* for short) iff the dependences of these operations form a connected graph after removing the final operations in the dependence graph that use the results produced by these redundant operations. A redundant cluster provides redundant lookahead values that can be used by the final operations in one parallel step in a schedule, as shown in Figure 5. Note that the dependence graph of a redundant cluster in schedule A is not restricted to a tree, i.e., it can be an arbitrary dependence graph.

The non-leaf nodes of a redundant cluster are those redundant values produced by the redundant operations in the cluster. The leaves of a redundant cluster are those array elements used by the redundant operations in the redundant cluster. A redundant cluster covers those redundantly scheduled elements as its leaves.

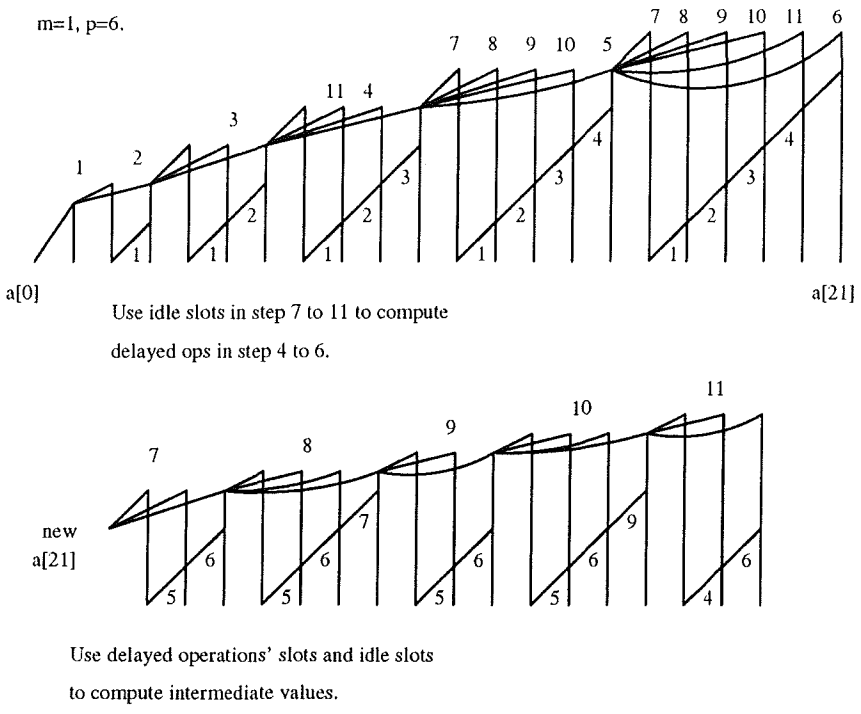


Figure 4: The amortizing scheme for extended schedule H with even p.

Lemma 4.1 $lns(A) \geq$ (number of locally unscheduled elements) + (number of local trees).

Proof: In a single step, only one locally unscheduled element alone or the intermediate values of one local tree alone can be used to compute final values. □

Lemma 4.2 For prefix sums of size $N = T(p+1)/2$ and $T \geq p$, if a schedule A has no fewer than T local trees, then A needs at least $T+1$ steps to complete the computation.

Proof: Schedule H computes prefix sums of size N in time T by Theorem 3.2. Let $p_1 = (p+1)/2$. The number of locally scheduled elements of schedule $A \leq 2T + T(p_1 - 1) - T = Tp_1$. So

$lns(A) \geq$ number of locally unscheduled elements+
number of local trees =
$=$ ((array size $-$ 1)$-$
number of locally scheduled elements) $+ T \geq$
$\geq (Tp_1 - Tp_1) + T = T,$

showing that a schedule with T or more local trees completes Tp_1 final operations in at least T steps. $lns(A) > T$ holds because all elements except $a[0]$ are covered by local trees and the first local tree has height of at least one. □

Lemma 4.3 In a parallel schedule A for prefix sums of size N, the number of redundantly scheduled elements is smaller than or equal to the sum of number of redundant operations and number of redundant clusters, i.e.,

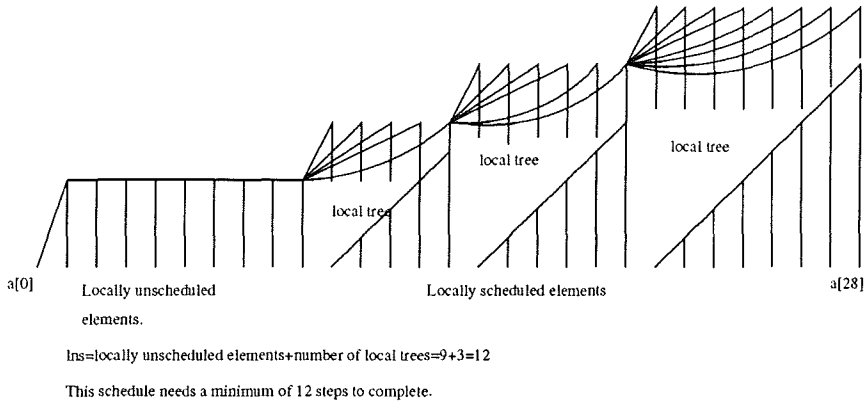


Figure 5: Local trees and least number of steps needed to compute recurrence.

(number of redundantly scheduled elements in A)$\leq$
leq $O_i(A)$+number of redundant clusters.

Proof: A redundant cluster C with $O_i(C)$ redundant operations that uses h elements of array a has at least $h-1$ redundant operations, i.e., $O_i(C) \geq h-1$, or equivalently $h =$ number of redundantly scheduled elements in $A \leq O_i(C)+1$. The truth of this statement can be shown by induction on the number of elements used by a redundant cluster.

Let c be the number of redundant clusters in schedule A. Because the redundant clusters in A are disconnected(i.e., there is no dependence arc between any pair of redundant clusters),

(number of redundantly scheduled elements in A)=
$= \sum_j^c$(number of redundantly scheduled elements
in redundant cluster C_j in A)$\leq$
$\leq \sum_j^c(O_i(C_j)+1) =$
$= O_i(A) + c =$
$=$ $O_i(A)$+number of redundant clusters in A.

□

Lemma 4.4 For prefix sums of size $N = T(p+1)/2$ and $T \geq p$ and any schedule A with $O_i(A) \leq O_i(H)$, then $lns(A) \geq T$(i.e., schedule A uses more time steps than schedule H to complete the computation).

Proof: When schedule A has T or more local trees, $lns(A) > T$ by Lemma 4.2. We need only to show that the lemma holds when A has less than T local trees.

By Lemma 4.3, for a schedule A with fewer than T local trees,

(the number of locally unscheduled elements in A)=
= (array size)-1-
(number of locally scheduled elements in A)=
$\geq Tp_1-(O_i(A)$+(number of local trees in A)).

Then by Lemma 4.2,

$$\begin{aligned} lns(A) \geq & \text{ (number of local trees in } A) + \text{ (number of} \\ & \text{locally unscheduled elements in } A) \geq \\ \geq & \text{ (number of local trees in } A) + \\ & [Tp_1\text{-}(O_i(A)+(\text{number of local trees in } A))]= \\ = & Tp_1 - O_i(A). \end{aligned}$$

Since $O_i(H) = T(p_1 - 1)$ by Theorem 3.2,

$$\begin{aligned} Tp_1 - O_i(H) &= Tp_1 - T(p_1 - 1) = \\ &= 1/2[T(p+1) - T(p-1)] = \\ &= T. \end{aligned}$$

From the two statements above and given that $O_i(A) \leq O_i(H)$, we have

$$\begin{aligned} lns(A) &\geq Tp_1 - O_i(A) \geq \\ &\geq Tp_1 - O_i(H) = T, \end{aligned}$$

for $T \geq p \geq 2$. □

Theorem 4.1 (Optimality of the Harmonic Schedule) Schedule H gives the maximum possible speedup $(p+1)/2$ with p processors and it uses the minimum possible number of processors for that speedup. Equivalently, given $p \geq 2$ processors, schedule H computes the maximum possible number, $T(p+1)/2$, of final values of the recurrence in $T \geq p$ steps.

Proof: We know that in any schedule computing the given recurrence of size N, the number of final operations remains a constant $N-1$. In order to make a parallel schedule faster than schedule H, we can only change the intermediate operations. There are only two ways of changing the intermediate operations: rearrange with equal or more intermediate operations than H, or rearrange with fewer intermediate operations than H. It suffices to show that (1) any schedule with more intermediate operations than H cannot compute more final results than H in time $T \geq p$, and (2) any schedule with no more intermediate operations than H cannot compute more final results than H in time $T \geq p$.

Statement (1) is true because schedule H has maximum possible processor utilization in $T \geq p$ steps. Any schedule with more intermediate operations than H in T steps can only produce fewer final results than H regardless of how the intermediate operations are arranged, assuming it fully utilizes p processors.

Now we turn to statement (2). If a schedule A outperforms H, then there exists $T_0 \geq p$ such that A computes more final results than H after T_0. Because H is already fully utilizing all p processors at any time $T \geq p$ including T_0, and because the number of final operations remains the same for any schedule, to outperform H, A must reduce its intermediate operations and use the saved processor time slots to do more final operations than H. That is, A must compute more results than H at the end of T_0 using fewer intermediate operations than H. We may ask what if we reduce the number of intermediate operations in the T steps and use the saved processor time slots to reschedule with more time steps $T' > T$. The answer is that we still have to come back to prove that any schedule with more than, equal to, or fewer operations than H at step T' cannot compute more final values than H. Therefore, to show H's optimality, it suffices to show that at any $T \geq p$, any schedule A with $O_i(A) \leq O_i(H)$ computes no more final results than H. This has been precisely proven by Lemma 4.4. □

Corollary 4.1 Schedule H computes the prefix sums of size $N \geq p(p+1)/2$ in time $\lceil 2(N-1)/(p+1) \rceil$, i.e., schedule H achieves the strict lower bound on time.

	T																							
N	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25
p 1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26
2		4	5	7	8	10	11	13	14	16	17	19	20	22	21	23	24	26	27	29	30	32	33	35	36
3			7	9	11	13	15	17	19	21	23	25	27	29	31	33	35	37	39	41	43	45	47	49	51
4				11	13	16	18	21	23	26	28	31	33	36	38	41	43	46	48	51	53	56	58	61	63
5					16	19	22	25	28	31	34	37	40	43	46	49	52	55	58	61	64	67	70	73	76
6						22	25	29	32	36	39	43	46	50	53	57	60	64	67	71	74	78	81	85	88
7							29	33	37	41	45	49	53	57	61	65	69	73	77	81	85	89	93	97	101
8								37	41	46	50	55	59	64	68	73	77	82	86	91	95	100	104	109	113

T is time steps, p number of processors, N array size.

Figure 6: The maximum size N of prefix sums that can be done in time T using p processors.

Proof: Theorem 4.1 proves this statement for $(N-1)$ a multiple of $(p+1)/2$. For $(N-1)$ not a multiple of $(p+1)/2$, we use the amortizing scheme in Theorem 3.2 to adjust the schedule. □

Corollary 4.2 (Ratio) For any optimal schedule for prefix sums, the ratio of the final to the intermediate operations is $(p+1)/(p-1)$ for $N \geq p(p+1)/2$, or $T \geq p$ and $p \geq 2$.

Proof: It follows from Theorem 3.2 and Theorem 4.1. □

Corollary 4.2 is a guide to deriving simpler, more concise and more program-space efficient optimal schedules. Such schedules will be presented in the next section.

To illustrate, in Figure 6, we show the maximum size of prefix sums N that can be done in time $T = 1, \ldots, 25$ using p ranging from 1 to 8 processors. Each number on the diagonal is equal to $1+\sum_1^p$. Starting from $p = 2$, each number on the diagonal is p larger than the preceding number on the diagonal. This implies the optimality—given one more step, p more final values can be computed. In each row for odd p starting from $T = p$, N increases by $(p+1)/2$ as T increases by one. In each row for even p starting from $T = p$, as T increases by one, N increases in turn by $\lfloor (p+1)/2 \rfloor$ and $\lceil (p+1)/2 \rceil$, so N increases on average by $(p+1)/2$.

5 The Piplelined Schedule

Schedule H specifies precisely the computation with maximum speedup for prefix sums of size $N \geq p(p+1)/2$. Note that it has p steps in the loop body for p processors. Although it does not make schedule H impractical, we wondered whether simpler schedules than H exist. By Corollary 4.2, the ratio of final to intermediate operations of an optimal schedule is $(p+1)/(p-1)$ for $N \geq p(p+1)/2$. Schedule H achieves this ratio in every iteration of p steps. If a schedule can achieve this ratio in fewer steps than p, then it has a smaller loop body than H and is still optimal. Applying the idea of Software Pipelining[3], we can indeed derive a simpler, more concise and more program-space efficient schedule than H called Pipelined Schedule or schedule P. Schedule P has in its loop body one statement with $(p+1)/2$ final and $(p-1)/2$ intermediate operations and yields the same speedup as schedule H except for some startup and wind-down overhead. Strictly speaking, it is possible to directly achieve the ratio $(p+1)/(p-1)$ in a single step for odd p, but not for even p, where "amortizing" is needed to achieve the same ratio. Next, we give the schedule P for an odd number of processors p in detail, and show how the schedule P for an even number of processors p can be derived.

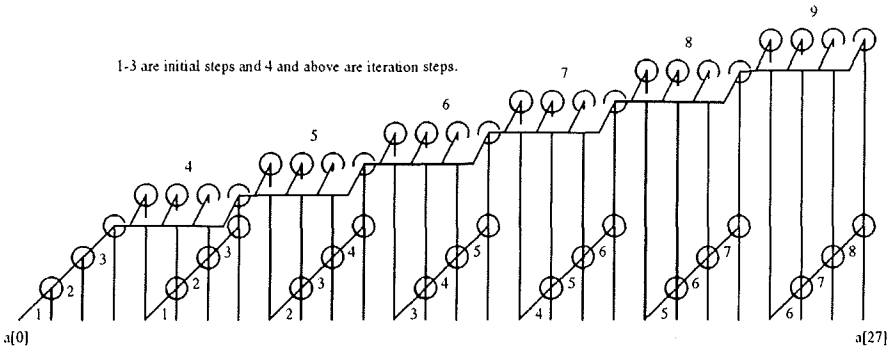


Figure 7: The Pipelined Schedule for $p = 7$.

Prefix sums of size N can be computed by the Pipelined Schedule with an odd number of processors p. The first $(p_1 - 1)$ where $p_1 = (p + 1)/2$ steps of the schedule set up for the actual iterations. The iteration step computes both final values(using preceding final values, and previously computed intermediate values), and intermediate values for future iterations. The array subscript calculation in an instantiated schedule P becomes very simple and can be reduced to addition using strength reduction[2]. Note that all operations in the loop body are in a single step and thus execute in parallel.

The Pipelined Schedule(P):

Step $j = 1, \ldots, p_1 - 1$.
$a[j] = a[j-1] + a[j]$,
$t[1 + (j-1)(p_1 - 2) + l] =$
$= t[1 + (j-2)(p_1 - 2) + l] + a[p_1 + j + (p_1 - 1)l]$,
$l = 0, \ldots, j - 2$,
$t[1 + (j-1)(p_1 - 2) + j - 1] = a[p_1 j] + a[p_1 j + 1]$.
for $i = 1, m = \lceil (N - p_1)/p_1 \rceil$
do each step in parallel on p processors
$a[p_1 i] = a[p_1 i - 1] + a[p_1 i]$,
$a[p_1 i + k] = a[p_1 i - 1] + t[(i-1)(p_1 - 1) + 1 + (k-1)(p_1 - 2)]$,
$k = 1, \ldots, p_1 - 1$,
$t[(p_1 - 1)i + m + (p_1 - 2)^2] =$
$a[p_1(i+m) + p_1 - m] +$
$t[(i-1)(p_1 - 1) + m + 1 + (p_1 - 2)^2]$,
$m = 1, \ldots, p_1 - 2$,
$t[(p_1 - 1)i + p_1 - 1 + (p_1 - 2)^2] =$
$a[p_1(i + p_1 - 1)] + a[p_1(i + p_1 - 1) + 1]$.
end for. □

In each iteration, p_1 final values of the array are computed by the first p_1 expressions, and $(p_1 - 1)$ intermediate values are computed by the last $(p_1 - 1)$ expressions. In comparison with schedule H, schedule P does not have to choose the proper epilogue at either compile or run time. Also, schedule P may be extended to handle a loop with multiple statements more easily than schedule H.

Example Let us instantiate a schedule P with $p = 7$. The schedule is as follows and is illustrated in Figure 7. Note that the array subscript calculation can obviously be reduced to addition using strength reduction. Each operation in a step is done in parallel on one of the $p = 7$ processors.

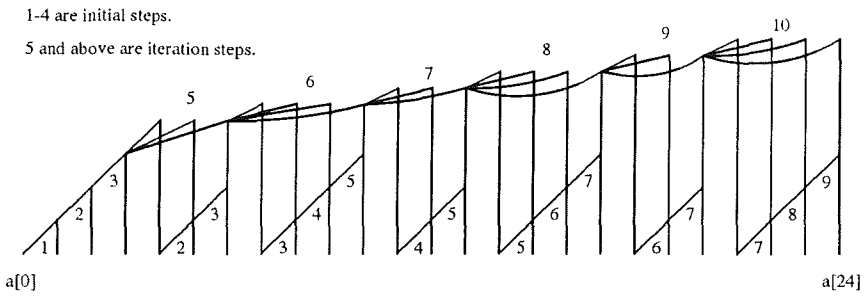


Figure 8: The Pipelined Schedule for $p = 6$.

step 1 $a[1] = a[0] + a[1], t[1] = a[4] + a[5];$
step 2 $a[2] = a[1] + a[2], t[3] = t[1] + a[6], t[4] = a[8] + a[9];$
step 3 $a[3] = a[2] + a[3], t[5] = t[3] + a[7], t[6] = t[4] + a[10],$
$t[7] = a[12] + a[13];$
for $i = 1, m = \lceil (N-4)/4 \rceil$ *do each step in parallel on p processors*
$a[4i] = a[4i-1] + a[4i], \quad t[3i+5] = a[4i+7] + t[3i+3],$
$a[4i+1] = a[4i-1] + t[3i-2], \quad t[3i+6] = a[4i+10] + t[3i+4],$
$a[4i+2] = a[4i-1] + t[3i], \quad t[3i+7] = a[4i+12)] + a[4i+13],$
$a[4i+3] = a[4i-1] + t[3i+2].$
end for □

Figure 8 illustrates the schedule derived for $p = 6$. In general, the schedule obtained for for an even number of processors p is an amortized schedule; for odd iterations it computes $\lfloor (p+1)/2 \rfloor$ final and $\lceil (p-1)/2 \rceil$ intermediate operations, while for even iterations it computes $\lceil (p+1)/2 \rceil$ final and $\lfloor (p-1)/2 \rfloor$ intermediate operations.

Theorem 5.1 Prefix sums of size N can be computed by schedule P on $p \geq 2$ processors in time $\lceil 2N/(p+1) \rceil + \lceil (p-1)/2 \rceil - 1$ for $N \geq \lceil p_1 \rceil$. Schedule P has speedup $S_p = (p+1)/2$ and utilization $U_p = 1$, as N goes to infinity.

Proof: The proof is relatively simple and can be found in [16]. □

6 A Formulation of Strict Time-Optimal Schedules

We have derived and proved strict time-optimal schedules for prefix sums of size $N > p(p+1)/2$. In this section, we present a new formulation of finding strict time-optimal schedules in terms of combinatorial optimization that unifies in a singl framework the optimal schedules for prefix problems for $N >$ and $N \leq p(p+1)/2$. We show some connections between the organization of prefix computation and the Pascal Triangle, and thus formulate the problem of finding strict time-optimal schedules as solving a system of inequalities, base on which an algorithm for constructing optimal schedules can be derived. We first give an intuition of our solution. We then characterize different costs of the final results in a computation tree[2]. Thirdly, we find the minimum cost

[2]The dependence graph of a prefix computation in $N \leq p(p+1)/2$ may actually not be a tree, but we use the term tree for it to avoid confusion.

for producing N final results on p processors. Next, we give the algorithm that builds the optimal computation tree based on the cost information for prefix problem of size N on p processors. In section 8, we shall present our algorithm in an algebraic formulation. This will facilitate our presentation of the proof of optimality of our schedule.

Since $T_p(H) = p$ is the strict time lower bound for $N = p(p+1)/2 + 1$, for any schedule A, $T_p(A) \leq p$ for $N \leq p(p+1)/2$. Given $N \leq p(p+1)/2$ and $p > 1$, finding optimal T_p is equivalent to finding maximum size N of prefix sums that can be computed in $T < p$ steps. We shall find our optimal schedules via this formulation for $N \leq p(p+1)/2$, because it gives us more insight into the problem.

By Lemma 4.2, an optimal schedule in T steps can have no more than $(T-1)$ redundant trees. Thus with $p \geq T$ processors, an optimal schedule must compute the maximum number of final values with no more than $(T-1)$ redundant trees. The only way this can be accomplished is to make each redundant tree produce, within the maximum possible height of the redundant tree, as many redundant results as possible that will be used to compute the final results. Equivalently, we want to make each redundant tree cover as many elements of array a as possible within the maximum possible height. The maximum possible height of the kth redundant tree is k for $1 \leq k < T$, because, with the kth redundant tree having a height greater than k, the final computation at step $(k+1)$ and afterwards will be delayed. Neither, should the heights of the redundant trees be lowered, because it would otherwise make the redundant trees cover fewer original elements, i.e., produce fewer final results than they could otherwise. For a computation tree of height k, the maximum number of original elements that can be covered by it is 2^k. With enough processors, all redundant trees in a schedule will grow to full redundant trees. The schedule is then saturated, i.e., no more final results can be computed with more processors in T steps.

The basic idead is try to achieve the dual goals: maximum utilization of processors and minimum number of redundant operations. However, for $N \leq p(p+1)/2$, the Harmonic Schedules in Section 3 is no longer applicable, intuitively because there are not enough prefix elements to make a single period(the minimum number of prefix elements that make a single period is $p(p+1)/2$(the period lenght), plus a starting element). Hence, the difficulty lies in determining, given that no regular pattern of computation exists for $N \leq p(p+1)/2$, how many redundant operations should be used and how the redundant trees are organized so that the dual goals can be accomplished.

Definition 6.1 A node N_1 in a binary dependence graph is said to *right depend* on node N_2 iff N_1 is the sink and N_2 is the right source for N_1. A node N_1 is said to right depend on node N_{k+1} *with a distance of* k iff there exist distinct nodes $N_2, N_3, \ldots, N_k$ such that N_i right depends on N_{i+1} for $i = 1, \ldots, k$. The concept of *left dependence* and *left dependence with a distance of* k are defined similarly.

Notation 6.1 **[Cost Vector]** Let $o_0, o_1, \ldots, o_m$ be the top fringe operations of a computation tree of height k, $k < m < 2^k$. We represent the sequence of right-dependence distances from o_j to $a[j](0 \leq j \leq m)$using a *cost vector* $C_k = (d_0, d_1, \ldots, d_m)$ such that the right-dependence distance from operation o_j to $a[j]$ is d_j. The value d_j in a cost vector is equal to the number of operations that the jth partial sum of this tree right depends on. Thus the sum of d_j's, $0 \leq j \leq m$, is equal to the number of operations in this tree.

As shown in Figure 9, the cost vectors for the trees(from left to right) is $(0,1,1,1,1,1,1,1)$ and $(0,1,1,2,1,2,2,3)$ respectively, and the total numbers of operations for the trees are equal to $(0+1+1+1+1+1+1+1) = 7$ and $(0+1+1+2+1+2+2+3) = 12$ respectively. We shall see soon that a computation tree can be constructed once a cost vector is determined. Theorem 6.1 and its corollary characterize different costs of final values in a full computation tree and show relevant properties, which will help us minimize the total cost when problem size and number of processors are given.

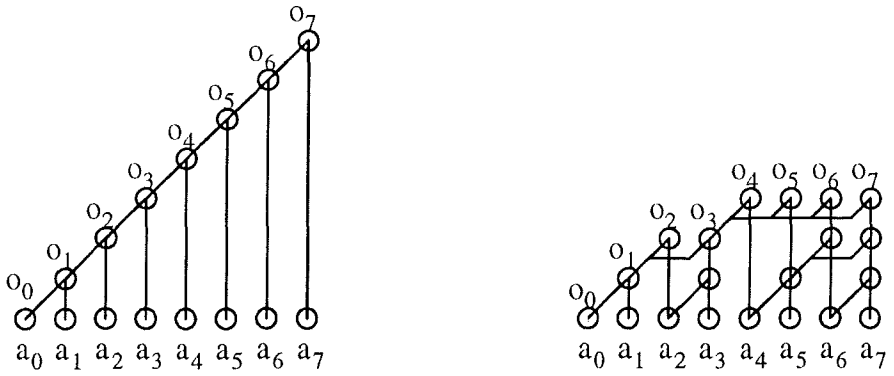


Figure 9: The right-dependences of a computation tree.

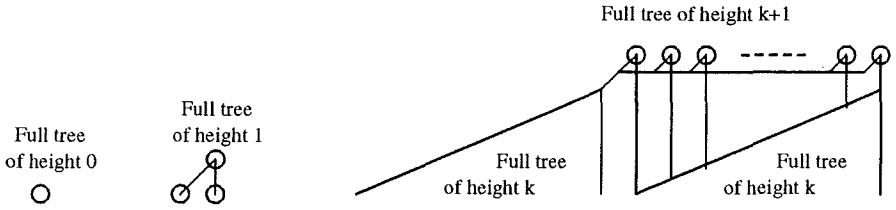


Figure 10: Number of nodes on the top fringe of the tree of height k with a right-dependence distance of j, $0 \leq j \leq k$.

Theorem 6.1 In a full tree of height k, the number of operation nodes on the top fringe that have a right-dependence distance of j, $0 \leq j \leq k$, is $g_{kj} = \binom{k}{j}$.

Proof: We prove this theorem by induction on the height k of a full tree.

Base. For tree height $k = 0$, there is only one quantity of right-dependence distance, 0. Thus $g_{00} = \binom{0}{0} = 1$. To facilitate understanding, we show one more case as our base of induction. For tree height $k = 1$, there are two quantities, 0 and 1, of right-dependence distance. As illustrated in Figure 10, the left leaf of the tree of height 1 is also the node on the top finge having a right-dependence distance of 0, and the root is the node on the top finge having a right-dependence distance of 1. Thus we have $g_{10} = \binom{1}{0} = 1$ and $g_{11} = \binom{1}{1} = 1$.

Induction assumption. We assume the theorem holds for $k \geq 1$, i.e., $g_{kj} = \binom{k}{j}$.

We now show that the theorem holds for $k + 1$. The intuition is shown in Figure 10: the number of nodes having a right-dependence distance of j on the top fringe of a full tree of height $k + 1$ is the sum of the number of top-fringe operation nodes of the first subtree of height k having right-dependence distance of j and the number of top-fringe operation nodes of the second subtree of height k having right-dependence of distance of $j - 1$, since there is a new layer of operation nodes on top of the second subtree of height k. By the preceding

statement and the induction assumption, we have

$$g_{k+1,j} = \binom{k}{j} + \binom{k}{j-1} = \binom{k+1}{j}.$$

The equation surrounding the last equal sign is a well-known combinatorial identity[14]. □

Corollary 6.1 1. The total number of operation nodes on the top fringe of a full tree of height k, $k \geq 0$, is

$$G_{k*} = \sum_{j=0}^{k} \binom{k}{j} = 2^k.$$

2. Given n full trees having height 0 through $n-1$, the sum of the numbers of operation nodes on the top fringes that have a right-dependence distance of j, $0 \leq j \leq n-1$, is

$$G_{*j} = \sum_{k=j}^{n-1} \binom{k}{j} = \binom{n}{j+1}.$$

3. The total number of operation nodes on the top fringes of n full trees of height from 0 through $n-1$ is equal to $2^n - 1$.

Proof: Claim 1 is true because the total number of operations on the top fringe of a full tree of height k is equal to the sum of the numbers of operations on that top fringe having a right-dependence distance of j, i.e., the sum of g_{kj}'s, $0 \leq j \leq k$. By Theorem 6.1, $g_{kj} = \binom{k}{j}$. Note that the truth of claim 1 can also be found by observing the number of array elements covered by a full tree.

Proof of claim 2.

$$\begin{aligned} G_{*j} &= \sum_{k=j}^{n-1} g_{kj} = \quad \text{(by requirement of the claim)} \\ &= \sum_{k=j}^{n-1} \binom{k}{j} = \quad \text{(by Theorem 6.1)} \\ &= \binom{n}{j+1} \quad \text{(by a combinatorial identity[14]).} \end{aligned}$$

The last claim follows from Claim 1 by summing up G_{k*},

$$\sum_{k=0}^{n-1} G_{k*} = \sum_{k=0}^{n-1} 2^k = 2^n - 1.$$

Equivalently, it also follows from Claim 2 by summing up G_{*j},

$$\sum_{j=0}^{n-1} G_{*j} = \sum_{j=0}^{n-1} \binom{n}{j+1} = 2^n - 1.$$

□

Figure 11 that shows the correspondence between Theorem 6.1 and its corollary and the Pascal triangle.

Corollary 6.2 [Cost vector of a full tree] Let $\vec{1}_i$ be an i-dimensional vector all elements of which are equal to 1 and $\dim(\vec{c})$ be the dimension of vector $\vec{c}$. The cost vector $\vec{c}_k$ of a full tree of height k can be determined recursively as follows.

$$\begin{aligned} \vec{c}_0 &= (0), \\ \vec{c}_k &= (\vec{c}_{k-1}, \vec{c}_{k-1} + \vec{1}_{\dim(\vec{c}_{k-1})}), \quad k \geq 1. \end{aligned} \tag{2}$$

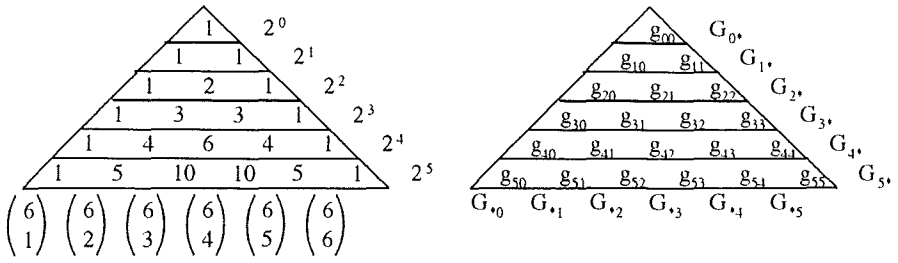


Figure 11: The correspondence between Theorem 6.1 with Corollary 6.1 and the Pascal triangle.

Proof: The proof is straightforward. □

In Figure 9, the cost vector of the full tree of height 3 is determined by this sequence of recursion: $\vec{c}_0 = (0)$, $\vec{c}_1 = (\vec{c}_0, \vec{c}_0 + \vec{1}_1) = (0,1)$, $\vec{c}_2 = (\vec{c}_1, \vec{c}_1 + \vec{1}_2) = (0,1,1,2)$, and $\vec{c}_3 = (\vec{c}_2, \vec{c}_2 + \vec{1}_4) = (0,1,1,2,1,2,2,3)$.

In what follows, we try to find the minimum cost for the computation tree for prefix of size N on p processors. Theorem 6.2 gives the system of inequalities from which the minimum number of time steps required to compute prefix problems using p processors can be determined.

Theorem 6.2 A lower bound T on time required to compute a prefix problem of size $N \leq p(p+1)/2+1$ on p processors can be solved from the following system of inequalities.

$$\sum_{i=1}^{k-1} \binom{T}{i} < N-1 \leq \sum_{i=1}^{k} \binom{T}{i}, \tag{3}$$

$$0 \leq pT - \left[\sum_{i=1}^{k-1} \binom{T}{i} i + kl\right] < p,$$

where $l = N-1-\sum_{i=1}^{k-1}\binom{T}{i}$, $1 \leq k \leq T$, $1 \leq T < N$. l is equal to the number of the top-fringe operations(i.e., number of final results) that have a right-dependence distance of k. k is the maximum right-dependence distance used by the schedule.

Proof: By Theorem 6.1, $\binom{T}{i}$ is the number of operations (i.e., number of final results) on the top fringe of the tree having a right-dependence distance of i(which can be seen as a cost of i) in a full tree of height T. $\sum_{i=1}^{k}\binom{T}{i}$, $1 \leq k \leq T$, is the number of final operations (results) having right-dependence distance no more than k in a full tree of height T. Solving the first inequality for T and k means to find a tree height T such that its final operation nodes of right-dependence distance equal to k cover the $N-1$ elements of the given prefix while its final operation nodes of right-dependence distance equal to $k-1$ are not enough to cover the $N-1$ elements. These final results that have minimum sum of right-dependence distances (i.e., minimum cost) would use minimum number of operations To compute $N-1$ final results in T steps. Since the number of processors p(i.e., the resource constraint) is not involved in the first inequality, there may be multiple pairs of solutions of T and k, i.e., each pair of T and k gives the minimum number of operations to be used to compute the $N-1$ prefix *with respect to* a particular number of processors. With the number of processors p used, the unique pair of T and k can be determined for computing $N-1$ results of the given prefix with minimum number of operations. The second inequality is used to choose T and k with respect to p.

In the second inequality, $\left[\sum_{i=1}^{k-1}\binom{T}{i}i + kl\right]$ equals the number of operations required by T redundant trees having height 0 through $T-1$ that computes $N-1$ final results. $0 \leq pT - \left[\sum_{i=1}^{k-1}\binom{T}{i}i + kl\right]$ means that there

are sufficiently many processor slots in T steps on p processors to compute those operations required by the $(T-1)$ redundant trees that covers N prefix elements. $pT - \left[\sum_{i=1}^{k-1} \binom{T}{i} i + kl\right] < p$ means that T is no more than the minimum number of steps that $N-1$ final results of the prefix problem can be produced on p processors. Thus the second inequality states that T is the lower bound on time that gives sufficiently many processor slots to compute the minimum number of operations required to produce $(N-1)$ prefix results. □

To show the T given by Theorem 6.2 is the strict optimal time for prefix computations of size N on p processors in $N \leq p(p+1)/2+1$, it suffices to show that, with the known conditions and the solutions for inequalities (3), a valid schedule can be constructed. We shall give an algorithm for constructing such schedules in next section.

7 Algorithm for Constructing Computation Trees

Based on Theorem 6.2, we construct our schedule for computing a prefix of N elements on p processors in $N \leq p(p+1)/2+1$ using Algorithm 7.1 given below.

Algorithm 7.1 Input: a prefix computation of N elements and number of processors p.
Output: a computation tree for making a parallel schedule that computes the prefix computation.

---

construct_schedule_for_ppc(N, p)
{
1. *find_min_time_min_p*(N, p);
2. *find_min_cost_vector*(T, k, l);
3. *construct_tree*($\vec{c}_T$, M_{TN});
}

□

We describe the three procedures of Algorithm 7.1 in the following and complete this section with examples. Once we have a computation tree, the parallel schedule for the given prefix can be constructed by allocating the operations in the tree to the p processors. Because the processor allocation is a difficult problem, We shall devote Section 8 to discussing it.

Procedure 7.1 Input: A prefix of size N and number of processors p.
Output: Procedure "find_min_time_min_p(N, p)" solves the inequalities in Theorem 6.2 for the minimum time T, the minimum number of processors p_0 required by T, and the maximum right-dependence distance k of our resulting schedule, and the number of final results l having right-dependence distance k. If the input $p \geq p_0$, the procedure completes and the resulting schedule will be constructed using p_0 processors, since the remaining $p - p_0$ processors would be of no use in achieving the minimum time. If the input $p < p_0$, Procedure "find_min_time_for_p(N, p)" continues to find the minimum time for N and p.

This procedure can be further sped up by starting with a better T and k than one for the outer loops in both procedures. We omit this detail to focus on the main ideas.

---

find_min_time(N, p)
{
find_min_time_min_p(N, p);

```
  if (p < p0) find_min_time_for_p(N, p);
}

find_min_time_min_p(N, p)
  {
   1.for (T = 1; T < N; T++)
   2.    for (k = 1; k ≤ T; k++)
            {
   3.        l = N − 1 − Σ_{i=1}^{k−1} C(T, i);
   4.        if (Σ_{i=1}^{k−1} C(T, i) < (N − 1) ≤ Σ_{i=1}^{k} C(T, i))
   5.          {find m such that 2^m ≤ N < 2^{m+1};
   6.           for (p0 = 2^{m−1}; p ≤ 2^m; p++)
   7.               if (0 ≤ pT − [Σ_{i=1}^{k−1} C(T, i) i + kl] < p)
   8.                  if (p ≥ p0) return (T, k, l, p = p0);
                       else return ;
            }
         }
  }

find_min_time_for_p(N, p)
  {
   1. for (T = 1; T < N; T++)
   2.    for (k = 1; k ≤ T; k++)
            {
   3.        if (Σ_{i=1}^{k−1} C(T, i) < N − 1 ≤ Σ_{i=1}^{k} C(T, i))
               and
               (0 ≤ pT − [Σ_{i=1}^{k−1} C(T, i) i + kl] < p)
               return (T, k, l);
            }
  }
```

□

Procedure 7.2 [Cutting Scheme] This procedure takes as input the solutions T, k and l from Procedure 7.1, and generates as output a cost vector, from which a computation tree for evaluation of the given prefix problem will be constructed. The dimension of the resulting vector is equal to the size N of the given prefix problem. The cost vector of a full tree is derived using the procedure given in Corollary 6.2. We shall show in Section 8 that valid schedules are constructed using the cost vector produced by this procedure.

```
construct_cost_vector(T, k, l)
  {
    1. derive cost vector c_T for full tree
       of height 0 through T − 1;
    2. remove from c_T elements greater than k;
    3. remove from c_T (C(T, k) − l) elements (i.e., keep l of them) equal to k as follows
```

("group" means a group of consecutive elements equal to k in $\vec{c}_T$):

3.1 point to the right-most group;

3.2 **while** $(\binom{T}{k} - l > 0)$

{

remove the last element in the group pointed to;

shift the pointer to the group on the left

(point to the right-most group after the left-most group

being operated on, i.e., the pointer moves in a clockwise

circular fashion);

};

4. **return** (resulting cost vector $\vec{c}_T$);

}

□

Note that there are other schemes for choosing from $\vec{c}_i$, $0 \leq i < T$, those l elements equal to k. We chose to use a simplest scheme as stated in step 3 to facilitate easy understanding of the main idea.

Procedure 7.3 This procedure establishes the dependence arcs between the nodes of the computation tree, which is placed in M_{TN} to be described next, resulting in a computation tree as the output that evaluates the given prefix problem of size N on p processors.

We store the operations of a tree in a $T \times N$ sparse matrix M_{TN}, called *construction matrix* of the tree, organized using its cost vector $\vec{c}_T = (d_0, d_1, \ldots, d_N)$ in the following way. The elements of the given prefix problem are stored in the 0th row of M_{TN}. d_j operations that the jth partial sum of the tree right depends on are stored in the jth column, with the operation with right-dependence distance i placed in the ith row, $1 \leq i \leq d_j$. Four fields are associated with an operation $o[i,j]$ in M_{TN}, *left_source*, *right_source*, *right_dep_distance*, and *number_of_depended*. $o[i,j]$.left_source and $o[i,j]$.right_source point to the left and right source nodes respectively. $o[i,j]$.right_dep_distance stores the right-dependence distance of node $o[i,j]$. $o[i,j]$.number_of_depended is the number of nodes that node $o[i,j]$ depends on, including $o[i,j]$ self.

---

construct_tree($\vec{c}_T$, M_{TN})

{

for $(j = 1; j <= N; j++)$

for $(i = 1; i <= d_j; i++)$

{

1. $o[i,j]$.right_source= $o[i-1,j]$;
2. $o[i,j]$.right_dep_distance = 1+
   ($o[i,j]$.right_source).right_dep_distance;
3. column_of_left_source=$j-$
   (1+($o[i,j]$.right_source).number_of_depended);
4. row_of_left_source =
   $d_{\text{column\_of\_left\_source}} - (d_j - i)$;
5. $o[i,j]$.left_source =
   o[row_of_left_source, column_of_left_source];
6. $o[i,j]$.number_depended= 1+

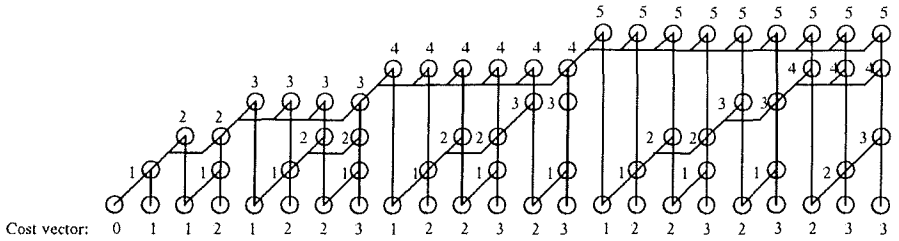


Figure 12: An optimal schedule for a prefix of input size 23 on $p_0 = 10$ processors determined by our procedure.

```
            (o[i, j].left_source).number_of_depended+
            +(o[i, j].right_source).number_of_depended;
        }
    }
```

□

Example 7.1 Find the minimum time and an optimal schedule for a prefix of size $N = 23$ on $p = 17$ processors.

Procedure "find_min_time_min_p" of Procedure 7.1 determined that $N - 1 = 22$ results can be computed on $p_0 = 10 < p = 17$ processors in $T = 5$ steps (i.e., in a computation tree of height $T = 5$) with $l = 7$ results having right-dependence distance $k = 3$.

Procedure 7.2 generated a cost vector $\vec{c}_5 = (0; 1; 1, 2; 1, 2, 2, 3; 1, 2, 2, 3, 2, 3; 1, 2, 2, 3, 2, 3, 2, 3, 3)$. With the cost vector, Procedure 7.3 constructed the schedule as shown in Figure 12. The processor allocation to the tree was done manually for this example and we shall give a processor allocation scheme in Section 8. By Theorem 6.2 and the fact that a valid schedule is constructed using the cost vector, the schedule in Figure 12 achieves the strict optimal time for prefix of size 23 on 10 processors.

□

Example 7.2 Find the minimum time and an optimal schedule for a prefix of size $N = 16$ on $p = 8$ processors. This example shows that our procedure produces a full tree-height-reduction schedule for N a power of 2 with enough processors.

Procedure "find_min_time_min_p" of Procedure 7.1 determined that $N - 1 = 15$ results can be computed on $p = 8$ processors in $T = 4$ steps (i.e., in a computation tree of height $T = 4$) with $l = 1$ result having right-dependence distance $k = 4$. We have obtained a full tree schedule of height $k = 4$!

Procedure 7.2 generated a cost vector $\vec{c}_5 = (0; 1; 1, 2; 1, 2, 2, 3; 1, 2, 2, 3, 2, 3, 3, 4)$, and Procedure 7.3 constructed the schedule as shown in Figure 13. The processor allocation to the tree was done manually for this example and we shall give a processor allocation scheme in Section 8. It is well known that it is the strict time-optimal schedule for prefix of $2^4 = 16$ on $2^3 = 8$ processors.

□

Example 7.3 Find the minimum time and an optimal schedule for a prefix of size $N = 29$ on $p = 7$ processors. This example illustrates that our procedure produces precisely a Harmonic Schedule when $N = p(p+1)/2 + 1$.

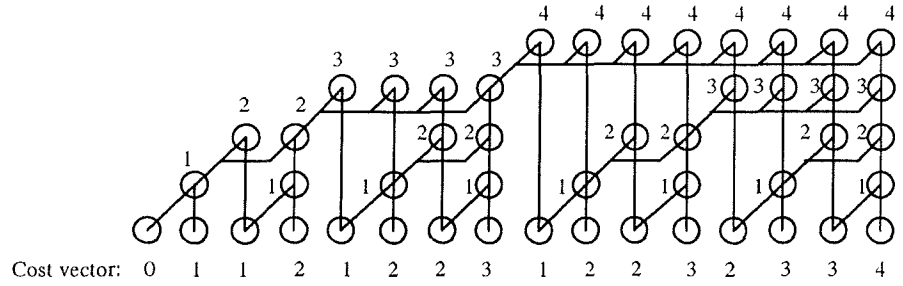


Figure 13: Our procedure generates a full tree-height reduction schedule for $N = 16$ and $p = 8$.

Procedure "find_min_time_min_p" of Procedure 7.1 decided that the minimum time required to compute $N - 1 = 28$ prefix sums is 5 steps and the minimum number of processors to accomplish that is $p_0 = 15 > p$. Procedure "find_min_time_for_p" of Procedure 7.1 continued and found the minimum time required to compute $N - 1 = 28$ prefix sums with $p = 7$ processors is 7 steps. That is, $N - 1 = 28$ results can be computed on $p = 7$ processors in $T = 7$ steps (i.e., in a computation tree of height $T = 7$) with $l = \binom{7}{2} = 21$ results having right-dependence distance $k = 2$.

Procedure 7.2 generated a cost vector $\vec{c}_5 = (0; 1; 1, 2; 1, 2, 2; 1, 2, 2, 2; 1, 2, 2, 2, 2; 1, 2, 2, 2, 2, 2; 1, 2, 2, 2, 2, 2, 2)$. Procedure 7.3 created precisely the same computation tree of the Harmonic Schedule as the one shown in Figure 1, which has been shown to achieve the strict optimal time in Section 4. □

8 Existence of Schedules in $N \leq p(p+1)/2$

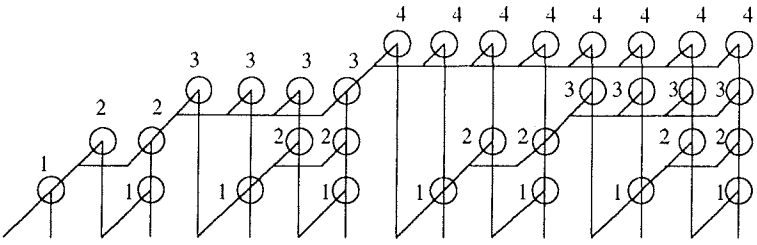
Algorithm 7.1 constructs a computation tree for computing the prefix of N elements on p processors. Now, whether the computation tree would make a valid schedule depends solely on the existence of a legal processor allocation onto the computation tree for arbitrary N and p in $N \leq p(p+1)/2$. It turns out that it is extremely difficult to prove this existence, since one would need to characterize certain properties of all the computation trees for N and p in $N \leq p(p+1)/2$. Fortunately, we were able to characterize a class of these computation trees and to show the existence of legal processor allocations for them, i.e., the existence of a class of strict time-optimal schedules(because by Theorem 6.2, the time implied by the computation trees is a lower bound).

We start with algebraic characterization for the schedule of a full tree and then proceed to find the characterization for the class of schedules. Given a full tree of 2^m elements, we label the columns as $k = 0, 1, ..., 2^m - 1$. Let b_k be the binary representation of k(recall that a column in a computation tree consists of a final operation and all other operations that the final operation right depends on). We have the following observations that hold for each column k.

Claim 8.1 *

1. The number of 1's in b_k is the number of operation nodes in column k. Thus to each operation node, we can associate the corresponding 1-bit. The kth element in the cost vector of a full tree is the number of 1's in b_k.

2. The position of the 1-bit (from the least significant bit) in b_k equals the left-most dependence distance (*lmdd*) of the corresponding node.



Cost vector: 0 1 1 2 1 2 2 3 1 2 2 3 2 3 3 4

0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1
0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
a0	a1	a2	a3	a4	a5	a6	a7	a8	a9	a10	a11	a12	a13	a14	a15

b0 b1 b2 b3 b4 b5 b6 b7 b8 b9 b10 b11 b12 b13 b14 b15

Order: 0 4 3 8 2 9 6 12 1 10 7 13 5 14 11 15

Figure 14: The binary representation for a full tree for prefix computation of 16 elements and the order relation between columns.

3. The operation nodes corresponding to the right-most consecutive blocks of 1's in b_k are nodes on the critical path of the schedule.

4. If a node n_j does not correspond to one of the right-most consecutive block of 1's in b_k and n_j corresponds to the j^{th} 1 relative to the left-most 1 in b_k, then the latest time n_j can be scheduled is $m - j + 1$. (This fact can be proved by induction.)

For example in Figure 14, in the column $b_{11} = (1011)$, the left-most node has $lmdd = 4$, and the right-most two 1's corresponds to two nodes on the critical path of the schedule.

Our algebraic formulation of the algorithm gives a concise description of the relations among the columns, using the binary representation. To see this, define an order relation $>$ between any two columns k_1 and k_2.

Definition 8.1 Given columns k_1 and k_2, we say k_1 is "larger" than k_2, or $k_1 > k_2$ if either of the following conditions holds:

1. b_{k_1} has more 1 bits than b_{k_2}, or
2. b_{k_1} and b_{k_2} have the same number of 1 bits and
   (a) the position of the first (from the least significant) 1-bit in b_{k_1} is earlier than that in b_{k_2}, or
   (b) the positions of the first 1-bit in b_{k_1} and b_{k_2} are the same, and $\tilde{b}_{k_1} > \tilde{b}_{k_2}$(binary arithmetic comparison) where $b_{k_1} = \tilde{b}_{k_1}10...0$ and $b_{k_2} = \tilde{b}_{k_2}10...0$ □

We show the order relation of columns in the full tree of 16 elements in Figure 14, the column with larger number means "larger" in the order relation. For example, $b_{11} > b_9$ since b_{11} has more number of 1's than b_9.

On the other hand, $b_{11} > b_{13}$ because of condition 2(a) above. It can also be seen that the relation defined above is transitive.

The order relation above between the columns enables us to show that legal processor allocations exist for a class of problem size N and number of processors in $N \leq p(p+1)/2+1$. It can also be used to describe Algorithm 7.1 equivalently well to using the cost vector. Suppose we want to determine a parallel schedule for computing the prefixes of $N < 2^m$ elements. Starting with the full tree schedule of 2^m elements, we first delete the "largest" $2^m - N$ columns with respect to above order relation.

By Theorem 6.1, there are $\binom{m}{i}$ columns each with i operation nodes for $i \leq m$. We now show that, for $N = \sum_{i=0}^{k} \binom{m}{i}$ and $p = \sum_{i=1}^{k} \binom{m-1}{i-1}$, legal processor allocations exist, i.e., the strict time-optimal schedules for the N and p above are found. The following lemma says that the number of processor slots (i.e. the number of operation nodes in the computation tree) is always a multiple of m when $N \sum_{i=0}^{k} \binom{m}{i}$ for some k.

Lemma 8.1 If $S = \sum_{i=0}^{k} i\binom{m}{i}$, then $S = mp$ for some integer p.

Proof: Observe that $i\binom{m}{i} = m\binom{m-1}{i-1}$. Thus $S = m\sum_{i=1}^{k} \binom{m-1}{i-1} = mp$ where $p = \sum_{i=1}^{k} \binom{m-1}{i-1}$ is an integer. □

Theorem 8.1 If $N = \sum_{i=0}^{k} \binom{m}{i}$, then the prefixes of N elements can be computed in m steps with p processors, where $p = \sum_{i=1}^{k} \binom{m-1}{i-1}$. That is, for this class of N and p, m is the strict optimal time (steps) for computing prefix computation of N elements on p processors.

Proof: Consider the set of all columns with exactly i 1's. The number of 1's in each row of this set is equal to $\binom{m-1}{i-1}$. If we consider the set of all columns with $\leq k$ 1's, then the number of 1's in each row of this set (which corresponds to the number of operation nodes at each time step) is equal to $\sum_{i=1}^{k} \binom{m-1}{i-1}$. Now the total number of processor slots required to compute the $N = \sum_{i=0}^{k} \binom{m}{i}$ prefixes is $\sum_{i=0}^{k} i\binom{m}{i} = m\sum_{i=1}^{k} \binom{m-1}{i-1} = mp$ by Lemma 8.1. Thus the prefixes of N elements can be computed in m steps with p processors, where $p = \sum_{i=1}^{k} \binom{m-1}{i-1}$.

Since these computation trees for N and p characterized in this theorem can also be constructed using Algorithm 7.1, by Theorem 6.2(on which Algorithm 7.1 is based), m is the strict optimal time for prefix of size N on p processors. □

As Theorem 8.1 indicates, the computation trees for the class of N and p in $N \leq p(p+1)/2+1$ have the nice structure that there are exactly p operation nodes at each time step, and thus p is the minimum number of processors required for to carry out the computation in m steps. For general values of N and p in $N \leq p(p+1)/2+1$, the computation trees seem much less apparent and at this moment, we do not have as a clean characterization for them as for the class of N and p in Theorem 8.1, thus we are unable to show the existence of legal processor allocations for these computation trees, i.e., we can only claim the strict time-optimal schedules for the values of N and p characterized in Theorem 8.1 but not for other values of N and p in $N \leq p(p+1)/2+1$.

9 Conclusion

We have founded the problem of finding strict time-optimal schedules for parallel prefix computation on an optimization(Theorem 6.2 formulated out of the correspondence of combinatorial properties of cost vectors to the Pascal Triangle (Theorem 6.1 and its Corallaries). On this foundation, we established Algorithm 7.1 for constructing parallel schedules. We have divided the parallel schedules for prefix computation of size N in two areas according to number of processors p: schedules in $N > p(p+1)/2$ and in $N \leq p(p+1)/2+1$.

For prefix of N elements on p processors in $N > p(p+1)/2$, we derived *Harmonic Schedules* and showed that the Harmonic Schedules achieve the strict optimal time (steps), $\lceil 2(N-1)/(p+1) \rceil$. We also derived *Pipelined*

Schedules, optimal schedules with $\lceil 2(N-1)/(p+1) \rceil + \lceil (p-1)/2 \rceil - 1$ time, which takes a constant overhead of $\lceil (p-1)/2 \rceil$ time steps more than the strict optimal time. Both the Harmonic Schedules and the Pipelined Schedules are expressed in program templates that are parameterized for the number of processors p, i.e., they can be generated in negligible time at compile time when p is known. Both the Harmonic Schedules and the Pipelined Schedules exhibit nice patterns of computation structure which make it easy to parallel program them. A main advantage of the Pipelined Schedules over the Harmonic Schedules is that the former has the smallest possible loop body, meaning smaller program space than the latter.

For prefix of N elements on p processors in $N \leq p(p+1)/2$, we have used Algorithm 7.1 to construct schedules, because for N and p in this range, the problem size is not large enough to accommodate any repeating pattern as in $N > p(p+1)/2$. For $N = \sum_{i=0}^{k} \binom{m}{i}$ and $p = \sum_{i=1}^{k} \binom{m-1}{i-1}$, We have derived schedules and shown in Theorem 8.1 that they complete the computation in m steps, the strict optimal time. For other values of N and p in $N \leq p(p+1)/2$, we have shown strong empirical evidence that the strict time-optimal schedules can be generated by Algorithm 7.1 but we are not yet able to provide a proof due to the difficulties in showing the existence of legal processor allocations for computation trees.

Except for this open end, we have concluded the search for strict time lower bound and schedules to achieve the bound for parallel prefix computation with resource constraints under CREW PRAM model.

References

[1] R. K. Agarwal, "Computational Fluid Dynamics on Parallel Processors", McDonnell Douglas Research Laboratories, A Tutorial at 1992 the 6th ACM SigArch International Conference on Supercomputing, Washington, D. C., July, 1992.

[2] A. Aho, R. Sethi, J. Ullman, "Compilers-Principles, Techniques, and Tools", Addison Wesley, 1986.

[3] A. Aiken, A. Nicolau, "Perfect Pipelining: A New Loop Parallelization Technique", Proceedings of ESOP, France, Springer-Verlag, 1988.

[4] G. Almasi and A. Gottlieb, Chapter 4, *Highly Parallel Computing*, Benjamin/Cummings, 1989.

[5] U. Banerjee, R. Eigenmann, A. Nicolau and D. Padua, "Automatic program parallelization", to appear in *Proceedings of IEEE*, Jan-March 1993.

[6] A. Bilgory, D. Gajski, "A heuristic for suffix solutions", IEEE Trans. Computers, Vol. c-35, No. 1, January 1986.

[7] R. Cole, U. Vishkin, "Faster Optimal Parallel Prefix Sums and List Ranking", Information and Control, 81, pp. 334-352, 1989.

[8] T. Corman, C. Leiserson, and R. Rivest, Chapter 30, *Introduction to algorithms*, The MIT Press and McGraw-Hill, 1990.

[9] F. E. Fich, "New bounds for parallel prefix circuits", Proc. of the 15th ACM STOC, pp 100-109, 1983.

[10] D. Kuck, "The Structure of Computers and Computations", Vol. 1, John Wiley & Sons, Inc., 1978.

[11] P. Kogge and H. Stone, "A Parallel Algorithm for the Efficient Solution of a General Class of Recurrence Equations", IEEE Transactions on Computer, Vol., C-22, No.8, August 1973.

[12] R. Ladner, M. Fischer, "Parallel Prefix Computation", JACM, Vol. 27, No.4, October 1980, pp. 831-838.

[13] F. T. Leighton, *Introduction to Parallel Algorithms and Architectures: Arrays-Trees-Hypercubes*, Morgan Kaufmann Publishers, Inc., San Mateo, California, 1992.

[14] C. L. Liu, *Introduction to combinatorial mathematics*, McGraw-Hill, 1968.

[15] Y. Muraoka, "Parallelism exposure and exploitation in programs", Ph. D thesis, Department of Computer Science, University of Illinois at Urbana-Champion, Report No. 424, February 1971.

[16] A. Nicolau, H. Wang, "Optimal Schedules for Parallel Prefix Computation with Bounded Resources", *SIGPLAN Notice and Proceedings of ACM 1991 Symposium on Principles and Practice of Parallel Programming*, Williamsburg, VA, April 21-24, 1991.

[17] Y. Ofman, "On the algorithmic complexity of discrete functions", *Cybernetics and control theory, Soviet Physics Doklady*, Vol. 7(7), pp. 589-591, January 1963.

[18] M. Snir, "Depth-size Trade-offs for Parallel Prefix Computation", Journal of Algorithms 7, 185-201, 1986.