Solving Linear Recurrences with Loop Raking*

Guy E. Blelloch
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213-3891

Siddhartha Chatterjee
RIACS
Mail Stop T045-1
NASA Ames Research Center
Moffett Field, CA 94035-1000

Marco Zagha
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213-3891

*This research was sponsored by the Avionics Laboratory, Wright Research and Development Center, Aeronautical Systems Division (AFSC), U.S. Air Force, Wright-Patterson AFB, Ohio 45433-6543 under Contract F33615-90-C-1465, ARPA Order No. 7597. CRAY Y-MP C90 time was provided by the Pittsburgh Supercomputing Center (Grant ASC890018P). This work was performed while the second author was a student at Carnegie Mellon University.

Proposed running head: Loop Raking

Direct all correspondence to:

Siddhartha Chatterjee
RIACS
Mail Stop T045-1
NASA Ames Research Center
Moffett Field, CA 94035-1000
Phone: (415) 604-4316
Email: sc@riacs.edu

Abstract

We present a variation of the partition method for solving linear recurrences that is well-suited to vector multiprocessors. The algorithm fully utilizes both vector and multiprocessor capabilities, and reduces the number of memory accesses and temporary memory requirements as compared to the more commonly used version of the partition method. Our variation uses a general loop restructuring technique called loop raking.

We describe an implementation of this technique on the CRAY Y-MP C90, and present performance results for first- and second-order linear recurrences. On a single processor of the C90 our implementations are up to 7.3 times faster than the corresponding optimized library routines in SCILIB, an optimized mathematical library supplied by Cray Research. On 4 processors, we gain an additional speedup of at least 3.7.

List of symbols

∞	infinity
$\oplus$	plus within circle
$\otimes$	times within circle
$\lfloor , \rfloor$	floor
$\approx$	approximately equal to

Loop	n_f	Lib r_∞	Lib $n_{1/2}$	Partition r_∞	Partition $n_{1/2}$	F77 r_∞	F77 $n_{1/2}$	Raked 1 Proc r_∞	Raked 1 Proc $n_{1/2}$	Raked 4 Procs r_∞	Raked 4 Procs $n_{1/2}$
R1 last	1	440	1500	230	2000	400	1500	450	4600	1790	61000
R1 all	1	200	3100	120	1800	180	1700	210	2500	830	41000
R1-C last	2	290	1900	210	2600	340	2000	430	3500	1720	37000
R1-C all	2	37	37	160	3100	200	1200	270	2000	1010	56000
R2 last	3	52	32	270	5200	N/A	N/A	320	1800	1260	52000
R2 all	3	41	28	190	4000	N/A	N/A	210	1300	840	62000

R1	`X(I) = X(I-1) + A(I)`
R1-C	`X(I) = X(I-1) * A(I) + B(I)`
R2	`X(I) = X(I-1) * A1(I) + X(I-2) * A2(I)`
last	Only `X(N)` is computed.
all	`X(1)` through `X(N)` are computed.

Table 1 Performance (in effective MFLOPS) on the CRAY Y-MP C90 for several implementations of first- and second-order recurrence solvers. The *Lib* rates are for the Cray SCILIB recurrence routines, which are only available for a single processor. The *Partition* rates are for a Fortran 77 implementation of the partition method on a single processor. The *F77* rates are for a Fortran 77 implementation of loop raking on a single processor. The *Raked* rates are for the CAL implementation of loop raking, on 1 and 4 processors. The number of floating point operations per iteration *of sequential code* is given by n_f. This number is used in calculating the asymptotic effective megaflops rate r_∞ of the various codes. The vector half-performance lengths of the codes are indicated as $n_{1/2}$.

1 Introduction

An m^{th}-order linear recurrence is a set of equations of the form:

$$x_i = \begin{cases} c_i, & 0 \leq i < m \\ (x_{i-1} \otimes a_{i,1}) \oplus \cdots \oplus (x_{i-m} \otimes a_{i,m}) \oplus b_i, & m \leq i < n \end{cases}$$

where $\oplus$ and $\otimes$ are binary associative operators, and $\otimes$ distributes over $\oplus$. Such linear recurrences frequently appear in scientific applications [19], are very useful in the design of parallel algorithms [18, 1], and can be used to solve a much broader class of recurrences [17, 10].

Researchers have been studying parallel and vector algorithms to solve linear recurrences since the 1960's [16, 3, 17, 29, 5, 15, 27, 21, 10, 33, 11, 22], and considerable effort has gone into producing fast implementations of these algorithms on parallel and vector machines [23, 20, 31, 30, 12, 24, 28, 26]. Some supercomputer manufacturers have considered the solution of linear recurrences important enough to warrant the addition of special hardware to execute recurrences efficiently [32]. Three major classes of algorithms have emerged out of the research on algorithms to solve linear recurrences: recursive doubling [17, 29], cyclic reduction [13, 3, 14], and the partition method [6, 10, 33]. These methods take advantage of the associativity of $\oplus$ and $\otimes$. Chen and Kuck [5] showed how to solve an m^{th}-order linear recurrence in $O(\lg n \lg m)$ time. This algorithm is impractical, however, since it requires nm^2 processors. It has been shown that both with a variation of cyclic reduction [11] and with the partition method [6], the total number of operations can be reduced to $nm(m+2)$ total calls to $\otimes$ and $nm(m+1)$ total calls to $\oplus$. When the number of processors p is significantly less than n, or the vector half-performance length $n_{1/2}$ on a vector computer is significantly less than n, the partition method has been shown generally to perform better in practice [12], and has been shown to be numerically stable for a large class of problems [31].

This paper discusses a variation of the partition method based on a loop restructuring technique we call *loop raking*. Although our variation requires the same number of arithmetic operations as the standard implementation of the partition method, it reduces the memory traffic (giving better running times), reduces the temporary memory requirements, and is better suited for a multiprocessor implementation because of its shorter vector half-performance length.

We have implemented our method on multiple processors of the CRAY Y-MP C90, and our performance results are summarized in Table 1 (effective MFLOPS on varying lengths are given in Figure 1). On one processor of the CRAY Y-MP C90 our recurrences are up to 7 times faster than the times of recurrences supplied as part of SCILIB [7] (an optimized scientific library supplied by Cray

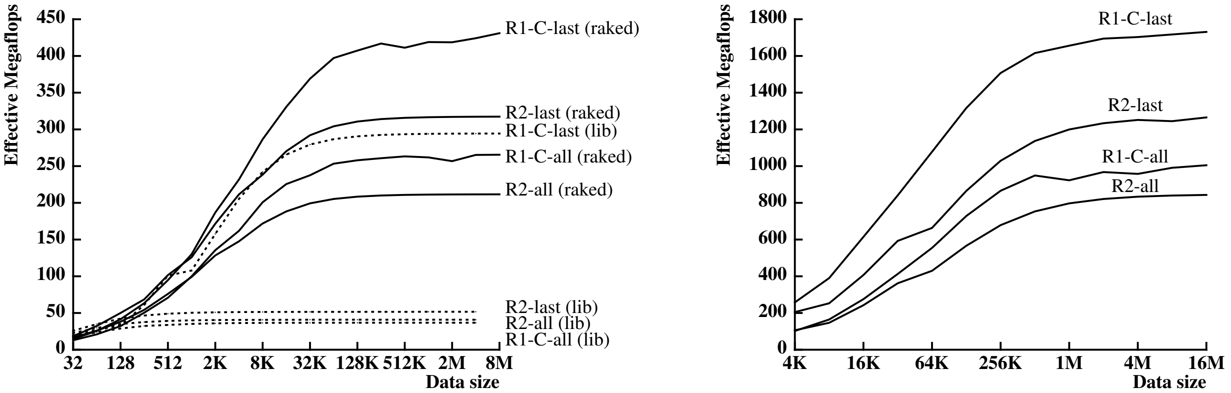

Figure 1: Performance results for first- and second-order linear recurrences on the CRAY Y-MP C90 (in effective MFLOPS). The graph on the left shows the performance of the SCILIB routines (FOLRNP, FOLR2P, SOLRN, SOLR) with dotted curves and single-processor loop-raked implementations with solid curves. The graph on the right shows the performance results for implementations of first- and second-order linear recurrences on four processors using loop raking.

Research). On multiple processors we achieve an additional almost linear speedup (the SCILIB recurrence routines do not use multiple processors). In our implementation we concentrated on the asymptotic performance rather than the half-performance length. We expect that the performance on short vectors could be significantly improved with an optimized implementation, especially for the multiprocessor implementation.

The paper is organized as follows. Section 2 explains the notion of loop raking. Section 3 reviews the partition method for solving linear recurrences and introduces our variation based on loop raking. Section 4 describes our implementation of linear recurrence solvers on the CRAY Y-MP C90. Conclusions and future directions are presented in Section 5.

2 Loop Raking

We assume a vector multiprocessor in which each processor has a set of vector registers each of length *L*. Loop raking uses a constant stride $s = \lfloor n/l \rfloor$ (where $l \leq L$) to access a set of elements that are evenly distributed across the input vector—as if a rake was placed over the vector and

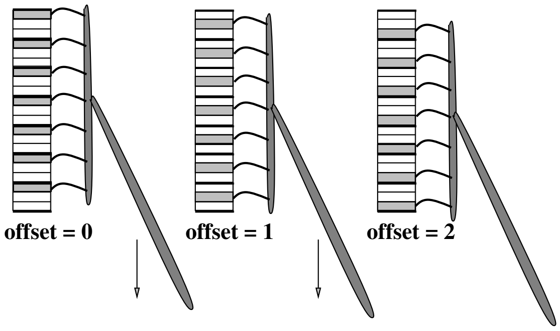

Figure 2: Loop raking. Vectors are accessed with a stride of $s = \lfloor n/l \rfloor$. To process an entire array of length N, we move the rake down one step on each of s iterations.

shifted on each vector load. On each load a total of *l* elements are loaded into a vector register. See Figure 2. Loop raking can be viewed as the complement of strip mining [25]. Both are loop transformations that define execution orders for a loop, *l* elements at a time. Strip mining can be viewed as mapping the loop iterations to an $l \times s$ matrix and operating on each column with a single vector operation; loop raking, on the other hand, can be viewed as organizing the iterations as an $s \times l$ matrix and operating on each row with a vector operation.

For loops without loop-carried data dependences [25], the results of strip mining and loop raking are identical. However, the methods are quite different in the presence of data dependences. In this paper we demonstrate how loop raking can be used to vectorize linear recurrences, which cannot be vectorized with strip mining. Elsewhere it has been shown how loop raking can be used to ensure stability in a step of a radix sort [34].

Choosing parameters It is important to choose the parameters of a raked loop so as to avoid memory conflicts. For the purposes of loop raking, we consider a vector of *n* elements as being organized in a rectangular fashion, characterized by three parameters *l*, *s*, and *f*, as shown in Figure 3. This characterization is solely for purposes of understanding; the vector is still stored in

$$\begin{array}{c|c|c|c|c|c|c}
\multicolumn{7}{c}{\longleftarrow \qquad\qquad l \qquad\qquad \longrightarrow} \\
\cline{2-6}
\uparrow & x_0 & x_s & \cdots & x_{(l-2)s} & x_{(l-1)s} & \uparrow \\
\cline{2-6}
 & \vdots & \vdots & \vdots & \vdots & \vdots & f \\
\cline{2-6}
s & \vdots & \vdots & \vdots & \vdots & x_{n-1} & \downarrow \\
\cline{2-6}
 & \vdots & \vdots & \vdots & \vdots & & \\
\cline{2-6}
\downarrow & x_{s-1} & x_{2s-1} & \cdots & x_{(l-1)s-1} & & \\
\cline{2-6}
\end{array}$$

$$n = (l-1)s + f$$

Figure 3: Rectangular organization of a vector of length *n* for loop raking. The vector is fully characterized by specifying the shape factors *l*, *s*, and *f*. Each cell represents one element of the vector. The vector is stored in standard order, with element i being stored at location i from the start of the vector. On the CRAY Y-MP C90, l $(0 < l \leq 128)$ is the length of the vector registers used in the computation. *s* is the stride used to load the vector register from memory, and also the number of iterations of each phase of the algorithm. *f* is the number of iterations that are executed with a vector register length of *l*. The remaining $(s - f)$ iterations are executed with the length set to $(l - 1)$.

standard order, with the i^{th} element residing at location i from the start of the vector. These three parameters characterize the vector completely and are called its *shape factors* [4]. From Figure 3, $n = (l-1)s + f$. This equation is underspecified for the purposes of determining the shape factors given *n*. We use the following additional constraints for choosing the parameters:

- $0 < f \leq s$.
- *s* is chosen to minimize bank conflicts; it is not allowed to be a multiple of `BANKS`, a constant that is determined by the number of memory banks and the bank cycle time (4 in the case of the CRAY Y-MP C90).
- *l* is kept as large as possible, and is bounded above by the length of the vector registers (128 in the case of the CRAY Y-MP C90).

Such an organization of the data makes all memory accesses constant-stride and avoids bank conflicts.

Extensions to vector multiprocessors To extend loop raking to a vector multiprocessor with P processors, a loop can be partitioned into $l \times P$ parts. For simplicity, assume that n is a multiple of $l \times P$. Each processor $P_i (0 \leq i < P)$ therefore loads with a constant stride of $n/(l \times P)$ starting at position $\frac{n}{P}i$. We can view this as increasing the rake size from l to $l \times P$. The code for each processor is identical—only the offset and possibly the f of the shape factor in the last processor are different. Since each region is independent, it is not necessary that the processors run synchronously. Synchronization is only required at the end of the raked loop.

Raking data instead of loops For some machines (such as the IBM 3090/VF), strided memory accesses are much slower than unit-stride accesses. In such a case, it may be more effective to store the data in raked order and use unit-stride memory operations. By this, we mean that the i^{th} element of the vector is stored at location $s \cdot (i \bmod s) + (i/s)$ from the start of the vector, where s is determined as above [31]. This imposes overhead in storage (since the shape factors must be kept as part of the storage descriptor for the vector), as well as in changing data representations. The same idea can be used for machines with caches. Since cache lines usually store multiple words, they can be used as an analog of vector registers. Raking data is clearly the correct approach in this case.

Raking the data only makes sense in a context where the data can be kept in raked order. Since standard elementwise vectorization can always run in raked order, it is reasonable to consider always keeping certain data in raked order.

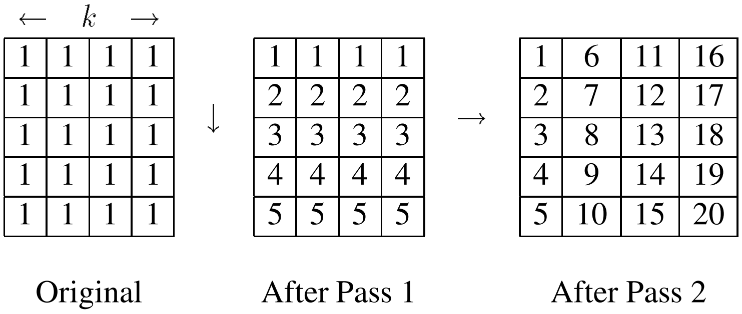

Figure 4: Standard Partition Method. $k \approx \sqrt{n}$.

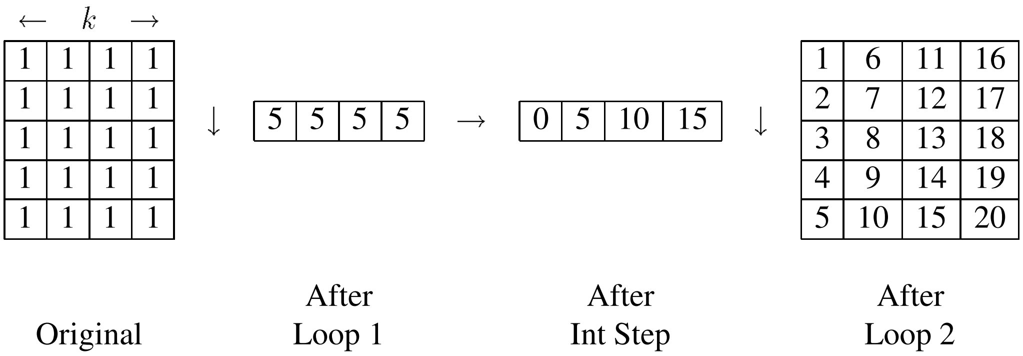

Figure 5: Loop-raked version of the Partition Method. $k \leq L$.

3 Overview of the Recurrence Algorithm

The partition method for solving linear recurrences is based on partitioning the input vectors into k contiguous blocks. The standard implementation views the n iterations as an $n/k \times k$ array [14]. The algorithm consists of two passes, the first across the columns and the second down the rows. This is shown in Figure 4. Our variation of the partition method is based on two raked loops with an intermediate step. The first loop is similar to the first pass of the standard technique since each addition is operating on elements separated by a distance n/k. In loop raking, however, we always select k based on machine parameters. In particular, we pick k so that the elements fit into a vector register. An example of the loop raking method is shown in Figure 5. It works as follows:

- **First raked loop:** Accumulate partial sums, one per processor. This only requires loading and accumulating. Only a single partial sum per vector element (partition) needs to be kept.
- **Intermediate step:** Sweep across the partial sums to generate starting offset of each partition. This is the only communication that is required between the partitions, and can be achieved in various different ways.
- **Second raked loop:** Generate final results from the offsets. This requires loading the original values, and writing the final results.

This variation has several potential advantages over the standard implementation.

Memory traffic and use The raked-loop version reduces memory traffic by keeping the partial sums in registers. It should be clear that for the case of a first-order recurrence with no constant (a scan), only one vector load is needed per iteration of the first loop. The second loop requires one vector load and one vector store per iteration. In the standard implementation of the partition method, if $n > L^2$ then either the number of columns or the number of rows has to be greater than *L*. This means that when passing over the dimension that is greater than *L*, the partial sums have to be written out of the registers on each iteration and then read back on the next iteration. For the more general case of an m^{th}-order recurrence, Table 2 gives the total number of loads and stores required by each version along with number of arithmetic operations, which is the same for both versions.

In order to keep the values in registers, at least m^2 vector registers are required for a recurrence with no constant, and $m(m+1)$ vector registers for a recurrence with a constant. This means that only first- and second-order recurrences can be implemented on the CRAY Y-MP C90 without

	Raked		Standard			
	Load	Store	Load	Store	$\otimes$	$\oplus$
No Const.						
Loop 1	m	0	$m(m+1)$	m	m^2	$m(m-1)$
Loop 2	m	1	m	1	m	$m-1$
With Const.						
Loop 1	$m+1$	0	$(m+1)^2$	$m+1$	$m(m+1)$	m^2
Loop 2	$m+1$	1	$m+1$	1	m	m

Table 2 Number of operations required by each iteration of an m^{th}-order linear recurrence. The complexity of the intermediate step is omitted as it is a lower-order term independent of the problem size.

running out of vector registers.

Another advantage of our method is that it requires only a constant amount of additional memory. This makes it more suitable for implementation as a library routine. Other implementations of the partition method require nm additional memory, assuming that we are not allowed to overwrite the coefficient arrays.

Vector half-performance length The standard model for the performance of a vector operation on a vector of length n is that the operation will take time:

$$T(n) = t_e(n + n_{1/2})$$

where t_e is the per-element incremental cost, and $n_{1/2}$ is the vector half-performance length (the number of elements at which half the peak performance $1/t_e$ is reached).

Based on this model, the time taken by the standard partition method is:

$$T(n) = (t_{e,1}(k + n_{1/2}))n/k + (t_{e,2}(n/k + n_{1/2}))k$$

where $t_{e,1}$ is the per-element cost of the first pass and $t_{e,2}$ is the per-element cost of the second pass.

This can be minimized to pick the best k, which gives approximately $k = (nt_{e,1}/t_{e,2})^{1/2}$. We can then generate an effective half-performance length for the standard method, which gives

$r_{1/2} \approx n_{1/2}^2$. We therefore see that the effective half-performance length of the standard partition method goes up as the square of the vector half-performance length. When going to multiprocessors $n_{1/2}$ goes up at least linearly with the number of processors P, and therefore $r_{1/2}$ increases quadratically with P.

On the CRAY Y-MP C90, for example, $n_{1/2} \approx 25$ on one processor, giving $r_{1/2} \approx 600$. However, if we go to 8 processors, $n_{1/2} \approx 500$, giving $r_{1/2} \approx 250,000$. This shows that on 8 processors of the C90, the partition method will not achieve half of its peak performance even for large n.

Using our variation, the intermediate step can be executed hierarchically on multiple processors giving an effective half performance length that only grows linearly with the number of processors.

Comparison with other work Our recurrence algorithm is similar to Version C of the algorithm proposed by Van der Vorst and Dekker [31], but differs in several respects that dramatically affect performance. First, we store the data in standard form; Van der Vorst and Dekker store the data in transposed (raked) form, but assume that transposition is free. Second, we restrict the number of partitions to the length of the vector registers, making optimal reuse of data. Van der Vorst and Dekker note the possibility of doing this in CAL. Third, we avoid writes altogether in the first loop, thus reducing the number of memory operations compared to their algorithm. Finally, our ability to choose shape factors circumvents the problem of performance degradation due to memory bank conflicts that they report.

4 Implementation on the CRAY Y-MP C90

We have implemented first- and second-order recurrences on multiple processors of the CRAY Y-MP C90. Table 1 summarizes their performance and compares them with several other implementations. The performance of the recurrence routines over a range of sizes is given in Figure 1.

In this section, we present pseudocode for our implementations of first- and second-order linear recurrences that should be applicable to any register-based vector machine. To implement the recurrences we used our own macro assembler, written in Lisp, to generate Cray Assembly Language (CAL) [9]. A direct Fortran implementation of loop raking resulted in the compiler storing and reloading the accumulation vectors in each iteration, which limited the performance of R1 and R1-C to about half of that achieved by our macro assembler. This difference in performances will increase for higher-order recurrences. (By unrolling the loops k-ways so that we pay this overhead once every k iterations, we have produced Fortran single-processor implementations of R1 and R1-C that perform within 25% of the CAL versions.)

In the pseudocode for the algorithms, we use the convention that names of vector registers begin with `V`, and names of scalar registers with `S`. The notation $X[\mathit{first} : \mathit{last} : \mathit{stride}]$ identifies elements from first to last, incrementing by stride. In the code we will assume $f = s$ for purposes of clarity. If $f < s$, the algorithm needs to be modified to decrement the contents of the Vector Length register by 1 for the last $(s - f)$ iterations. In our implementation on the CRAY Y-MP C90, we unroll the loops and rename registers to avoid locking vector registers. We only show the code for computing all the elements; the first two phases of the corresponding algorithm can be used if only the final value is required.

4.1 Algorithm for R1-C

In the first phase of the first-order recurrence R1-C (`x[i] = a[i]*x[i-1] + b[i]`), we must compute both a sum term and product term for each column. This is done with the following raked loop.

```
V_sum = 0.0;
V_prod = 1.0;
for (i = 0; i < s; i++) {
  V_a = a[i:(l-1)*s+i:s];
  V_b = b[i:(l-1)*s+i:s];
  V_temp = V_sum * V_a;
  V_sum  = V_temp + V_b;
  V_prod = V_prod * V_a;
}
```

The second phase fixes up the starting values for the final phase. This is done with a scalar loop that computes the appropriate combination of the sum and product terms resulting from the first phase with the initial value.

The final phase evaluates the appropriate terms and writes them out to memory. The following raked loop accomplishes this.

```
for (i = 0; i < s; i++) {
  V_a = a[i:(l-1)*s+i:s];
  V_b = b[i:(l-1)*s+i:s];
  V_prod = V_sum * V_a;
  V_sum = V_prod + V_b;
  x[i:(l-1)*s+i:s] = V_sum;
}
```

4.2 Algorithm for R2

We now consider the second-order linear recurrence `x[i] = a[i]*x[i-1] + b[i] * x[i-2]`. The computations in this recurrence involve multiplying 2×2 matrices [2]. The registers `V_11`, `V_12`, `V_21`, and `V_22` hold the elements from the corresponding positions in the matrices. The first phase of this algorithm is as follows.

```
V_11 = 1.0;
V_12 = 0.0;
V_21 = 0.0;
V_22 = 1.0;
for (i = 0; i < s; i++) {
  V_b = b[i:(l-1)*s+i:s];
  V_a = a[i:(l-1)*s+i:s];
  V_temp_2 = V_12 * V_b;
  V_temp_1 = V_11 * V_a;
  V_11 = V_12;
  V_12 = V_temp_1 + V_temp_2;
  V_temp_2 = V_22 * V_b;
  V_temp_1 = V_21 * V_a;
  V_21 = V_22;
  V_22 = V_temp_1 + V_temp_2;
}
```

The second phase fixes up the starting values for the final phase. This is done with a scalar loop that computes the successive products of the 2×2 matrices resulting from the first phase with the 2-vector of initial values.

The final phase evaluates the appropriate terms and writes them out to memory, using the following raked loop.

```
for (i = 0; i < s; i++) {
  V_b = b[i:(l-1)*s+i:s];
```

```
    V_bx2 = V_21 * V_b;
    V_a = a[i:(l-1)*s+i:s];
    V_ax1 = V_11 * V_a;
    V_21 = V_ax1 + V_bx2;
    x[i:(l-1)*s+i:s] = V_21;
    V_temp = V_11;
    V_11 = V_21;
    V_21 = V_11;
  }
```

4.3 Using multiple processors

The above algorithms require changes only to the scalar code (intermediate step) in order to use multiple processors of the CRAY Y-MP C90. To use P processors, we first partition the vectors into P pieces and assign each piece to one processor. Each processor executes the first phase of the single-processor recurrence algorithm. At the end of the first phase, each processor saves the contents of the vectors it has accumulated by writing them out to memory, and also writes out the reduction of all the elements in its partition. (The processor reduction is a pair of values for R1-C and four values for R2.) Then one of the processors computes all the terms of a recurrence on these P values. Each processor then restores state by reading in the vectors that it wrote out at the end of the first phase. It also reads in the updated processor reduction value and uses it as the initial value for performing a serial recurrence on the accumulation vectors. This gives the correct starting offsets for the final phase, which can then proceed as in the single processor algorithm. Note that the algorithm for the scalar phase is hierarchical. This reduces the serial overhead, since we perform the serial scan on a vector of length L, not of length L$\times P$.

Our multiprocessor implementation uses the $MDO Cray Assembly Language macro for generating microtasked loops [8]. In our prototype implementation, there is an overhead of approximately 2000 cycles for starting each raked loop. We expect that this overhead could be significantly reduced with further tuning and optimization.

5 Conclusions

We have described a variation of the partition method for solving m^{th}-order linear recurrences that is well-suited to vector multiprocessors such as the CRAY Y-MP C90. The algorithm fully utilizes the vector and multiprocessor capabilities of the machine, and reduces the number of memory accesses and the temporary memory requirements as compared to the more commonly used version of the partition method. This method uses a general loop restructuring technique called *loop raking*. On a single processor of the C90 our implementations are up to 7.3 times faster than the corresponding library routines in SCILIB. Performance increases almost linearly on multiple processors, resulting in a speedup of 3.7 on 4 processors.

Loop raking is a general loop restructuring technique that can be used to vectorize many problems where more standard execution orders such as strip mining fail. Further work is needed before it can become a standard compiler optimization. The treatment of raked loops must be expanded to handle nested loops. Also, work is needed to characterize loops that can profit from loop raking. We hope that the benefits of loop raking will stimulate further study of this compiler optimization.

References

[1] G. E. Blelloch. Scans as primitive parallel operations. *IEEE Transactions on Computers*, C-38(11):1526–1538, Nov. 1989.

[2] G. E. Blelloch. Prefix sums and their applications. Technical Report CMU-CS-90-190, School of Computer Science, Carnegie Mellon University, Nov. 1990.

[3] B. L. Buzbee, G. H. Golub, and C. W. Nielson. On direct methods for solving Poisson's equations. *SIAM Journal on Numerical Analysis*, 7(4):627–656, Dec. 1970.

[4] S. Chatterjee, G. E. Blelloch, and M. Zagha. Scan primitives for vector computers. In *Proceedings Supercomputing '90*, pages 666–675, Nov. 1990.

[5] S.-C. Chen and D. J. Kuck. Time and parallel processor bounds for linear recurrence systems. *IEEE Transactions on Computers*, C-24(7), July 1975.

[6] S. C. Chen, D. J. Kuck, and A. H. Sameh. Practical parallel band triangular system solvers. *ACM Transactions on Mathematical Software*, 4(3):270–277, Sept. 1978.

[7] Cray Research Inc., Mendota Heights, Minnesota. *UNICOS Math and Scientific Library Reference Manual SR-2081 Volume 3 (6.0)*.

[8] Cray Research Inc., Mendota Heights, Minnesota. *Macros and Opdefs Reference Manual, SR-0012D*, June 1988.

[9] Cray Research Inc., Mendota Heights, Minnesota. *Symbolic Machine Instructions Reference Manual, SR-0085B*, Mar. 1988.

[10] D. D. Gajski. An algorithm for solving linear recurrence systems on parallel and pipeline machines. *IEEE Transactions on Computers*, C-3(3):190–206, Mar. 1981.

[11] A. C. Greenberg, R. E. Ladner, M. S. Paterson, and Z. Galil. Efficient parallel algorithms for linear recurrence computation. *Information Processing Letters*, 15(1):31–35, Aug. 1982.

[12] H. Häfner and W. Schönauer. Investigation of different algorithms for the first order recurrence. *Supercomputer*, 7(6):34–41, Nov. 1990.

[13] R. W. Hockney. A fast direct solution of Poisson's equation using Fourier analysis. *Journal of the Association for Computing Machinery*, 12(1):95–113, Jan. 1965.

[14] R. W. Hockney and C. R. Jesshope. *Parallel Computers: Architecture, Programming, and Algorithms*. A. Hilger, Philadelphia, PA, second edition, 1988.

[15] L. Hyafil and H. T. Kung. The complexity of parallel evaluation of linear recurrences. *Journal of the Association for Computing Machinery*, 24(3):513–521, July 1977.

[16] R. H. Karp, R. E. Miller, and S. Winograd. The organization of computations for uniform recurrence equations. *Journal of the Association for Computing Machinery*, 14:563–590, 1967.

[17] P. M. Kogge and H. S. Stone. A parallel algorithm for the efficient solution of a general class of recurrence equations. *IEEE Transactions on Computers*, C-22(8):786–793, Aug. 1973.

[18] C. P. Kruskal, L. Rudolph, and M. Snir. The power of parallel prefix. In *Proceedings International Conference on Parallel Processing*, pages 180–185, Aug. 1985.

[19] D. J. Kuck, P. Budnik, S. C. Chen, E. Davis, J. Han, P. Kraska, D. Lawrie, Y. Muraoka, R. Strebendt, and R. Towle. Measurements of parallelism in ordinary FORTRAN programs. *IEEE Computer*, 7(1):37–46, Jan. 1974.

[20] S. R. Kunkel and J. E. Smith. Solving linear recurrences on pipelined computers. In *Proceedings of the Second International Conference on Supercomputing (ICS 87)*, pages 384–391, San Francisco, California, May 1987.

[21] R. E. Ladner and M. J. Fischer. Parallel prefix computation. *Journal of the Association for Computing Machinery*, 27(4):831–838, Oct. 1980.

[22] S. Lakshmivarahan and S. K. Dhall. New class of parallel algorithms for solving first-order and certain classes of second-order linear recurrences. In *Proceedings International Conference on Parallel Processing*, pages 843–845, Aug. 1985.

[23] J. J. Lambiotte and R. G. Voigt. The solution of tridiagonal linear sytems on the CDC STAR-100 computer. *ACM Transactions on Mathematical Software*, 1(4):308–329, Dec. 1975.

[24] R. Overill. On the efficient vectorization of the general first-order linear recurrence relation. *Supercomputer*, 8(2):31–36, Mar. 1991.

[25] D. A. Padua and M. Wolfe. Advanced compiler optimizations for supercomputers. *Communications of the ACM*, 29(12):1184–1201, Dec. 1986.

[26] L. J. Podrazik and H. E. Conn. Parallel recurrence solvers for vector and SIMD supercomputers. In *Proceedings International Conference on Parallel Processing*, pages III.88–95, Aug. 1992.

[27] A. H. Sameh and R. P. Brent. Solving triangular systems on a parallel computer. *SIAM Journal on Numerical Analysis*, 14(6):1101–1113, Dec. 1977.

[28] K. Shimizu and Y. Kanada. Solving linear recurrence problems on supercomputers. *Supercomputer*, 8(1):30–47, Jan. 1991.

[29] H. S. Stone. Parallel tridiagonal equation solvers. *ACM Transactions on Mathematical Software*, 1(4):289–307, Dec. 1975.

[30] Y. Tanaka, K. Iwasawa, S. Gotoo, and Y. Umetani. Compiling techniques for first-order linear recurrences on a vector computer. *Journal of Supercomputing*, 4(1):63–82, 1990.

[31] H. A. Van Der Vorst and K. Dekker. Vectorization of linear recurrence relations. *SIAM Journal on Scientific and Statistical Computing*, 10(1):27–35, Jan. 1989.

[32] H. Wada, K. Ishiii, M. Fukagawa, H. Murayama, and S. Kawabe. High-speed processing schemes for summation type and iteration type vector instructions on HITACHI supercomputer S-820 system. In *Proceedings 1988 ACM Conference on Supercomputing*, pages 197–206, July 1988.

[33] H. H. Wang. A parallel method for tridiagonal equations. *ACM Transactions on Mathematical Software*, 7(2):170–183, June 1981.

[34] M. Zagha and G. E. Blelloch. Radix sort for vector multiprocessors. In *Proceedings Supercomputing '91*, pages 712–721, Nov. 1991.

Biographies

Guy E. Blelloch has been an assistant professor in the School of Computer Science at Carnegie Mellon University since 1988. He received his B.A. (1983) in Physics from Swarthmore College, and his M.S. (1986) and Ph.D. (1988) in Computer Science from the Massachusetts Institute of Technology. While working on his Ph.D. he also worked for Thinking Machines Corporation where he was the designer and principal implementor for the original sorting, scanning, and hypercube communication instructions for the Connection Machine CM-2. He is interested in practical issues in the design and implementation of parallel algorithms for parallel computers.

Siddhartha Chatterjee received his B.Tech. (1985) in electronics and electrical communications engineering from the Indian Institute of Technology, Kharagpur, and his M.S. (1988) and Ph.D. (1991) in computer science from Carnegie Mellon University. He is currently a Postdoctoral Scientist at the Research Institute for Advanced Computer Science at NASA Ames Research Center in Moffett Field, California. His research interests include the design and implementation of data-parallel languages for irregular problems, and compiler techniques for automatic data placement on distributed-memory parallel machines.

Marco Zagha received his B.S. (1988) in Computer Science and Engineering from the University of California at Los Angeles and his M.S. (1991) in Computer Science from Carnegie Mellon University. He is currently a doctoral candidate in the School of Computer Science at Carnegie Mellon University. His research interests include parallel algorithms for unstructured and combinatorial problems (such as sparse matrix computation and sorting) and parallel language implementation.