Computing Programs Containing Band Linear Recurrences on Vector Supercomputers

ICS Technical Report 92-113

Haigeng Wang Alexandru Nicolau
Department of Information and Computer Science

Computing Programs Containing Band Linear Recurrences on Vector Supercomputers *

Haigeng Wang Alexandru Nicolau
Department of Information and Computer Science
University of California, Irvine
Irvine, CA 92717-3425

Abstract

Many large-scale scientific and engineering computations, e.g., some of the Grand Challenge problems [1], spend a major portion of execution time in their core loops computing band linear recurrences (BLR's). Conventional compiler parallelization techniques [4] cannot generate scalable parallel code for this type of computation because they respect loop-carried dependences (LCD's) in programs and there is a limited amount of parallelism in a BLR with respect to LCD's. For many applications, using library routines to replace the core BLR requires the separation of BLR from its dependent computation, which usually incurs significant overhead. In this paper, we present a new scalable algorithm, called the *Regular Schedule*, for parallel evaluation of BLR's. We describe our implementation of the Regular Schedule and discuss how to obtain maximum memory throughput in implementing the schedule on vector supercomputers. We also illustrate our approach, based on our Regular Schedule, to parallelizing programs containing BLR and other kinds of code. Significant improvements in CPU performance for a range of programs containing BLR implemented using the Regular Schedule in C over the same programs implemented using highly-optimized coded-in-assembly BLAS routines [11] are demonstrated on Convex C240. Our approach can be used both at the user level in parallel programming code containing BLR's, and in compiler parallelization of such programs combined with recurrence recognition techniques for vector supercomputers.

keywords: band linear recurrences (BLR's), parallel evaluation of BLR's with resource constraints, programs with BLR's, parallel programming, vector supercomputer.

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

1 Introduction

Systems of linear equations arise in many large-scale scientific and engineering computations such as some of the Grand Challenge problems [1, 16], e.g., computational fluid dynamics, weather forecasting, structural analysis, plasma physics, etc. Frequently, the discretization of partial differential equations in these computations result in band linear systems involving band linear recurrences (BLR's). These band linear systems consume the major portion of computing time in these computations [1]. Furthermore, a parallel program that preserves loop-carried dependences (LCD's) and contains BLR's is not *scalable*: the speedup obtained with such a program over its sequential counterpart is limited by a constant (usually very small) regardless of how many processors are used. As a result, not only solving these band linear systems is time-consuming but also a significant amount of resources, e.g., vector processors or processing elements (PE's) in massively parallel computers is wasted.

Compiler parallelization techniques that respect LCD's [4] cannot turn the programs containing BLR's into scalable parallel ones. To compute these programs on parallel computers, special parallel algorithms are devised for some of them, e.g., recursive-doubling [19] for 1st-order LR's, modified cyclic reduction for 1st and 2nd-order LR's, cyclic reduction [22] for diagonally-dominant tridiagonal linear solver, "burn-at-both-ends" [12] for symmetric positive definite tridiagonal linear solvers. The most frequently used programs for solving band linear systems are written into library routines using the different methods mentioned above, e.g., BLAS routines in Vector Library on Convex and Scientific Library on Cray. However, three problems arise with using these library routines. First, many programs mix band linear systems and other code in the same main loops. Transforming these loops to facilitate invoking library routines will make the loops perform significantly worse than parallel programming these loops directly. The main reason is that separation of band linear code from the other code in the loop severely degrades locality of reference. Another reason is that the library routines conventionally destroy the input arrays to store intermediate and final results, which incurs the overhead of keeping multiple copies of input arrays for possible later use. Secondly, many other kinds of frequently used band linear programs, both standard and hybrid with other codes, are not covered by the library routines, and the number of such programs increases as more large-scale computations are created. It would be inefficient, (probably infeasible), to create a library routine for each application. Thirdly, none of those parallel programming methods is optimal (in terms of execution time) with resource constraints, e.g., with a fixed number of processors independent of the problem size. That is, there may exist a faster and more general parallel-programming method than the problem-specific methods for the band linear systems.

Main Results: In this paper, we present a new scalable algorithm, called the *Regular Schedule*, for parallel evaluation of general BLR's (i.e., mth-order LR's for $m \geq 1$). The schedule's regular organization of the computation for BLR greatly enhances its scalability and makes it extremely well suited for vector supercomputers and massively parallel computers. We describe our implementation of the Regular Schedules and discuss how to obtain maximum memory throughput in implementing the schedule using analytical and experimental methods on vector supercomputers (our experiments were performed on Convex C240). We also illustrate our approach, based on our Regular Schedule, to parallelizing programs containing BLR and other kinds of code. Significant improvements in CPU performance for a range of programs containing BLR implemented using the Regular Schedule in C over the same programs implemented using highly-optimized coded-in-assembly BLAS routines [11] are demonstrated on Convex C240. Our approach can be used both by programmers in coding parallel programs containing BLR's and by compilers in automatic parallelization of such programs in conjunction with recurrence recognition techniques for vector supercomputers.

The rest of the paper is organized as follows. Section 2 briefly reviews related work. Section 3 presents the Regular Schedule. Section 4 describes the implementation of the Regular Schedule and discusses how to obtain the maximum memory throughput on a vector supercomputer. Section 5 describes our approach based on the Regular Schedule to parallelizing programs containing BLR's intermixed with additional code. Section 6 reports the timing results of several programs containing BLR implemented using the Regular Schedule with the same programs implemented using BLAS routines of VECLIB on Convex C240. Section 7 concludes this paper with some open problems and future work.

2 Related Work

Because of its important role in scientific computing, efficient evaluation of recurrence equations has been studied extensively for more than twenty years. The study of Uniform Recurrence Equations (UREs) was introduced by Karp, Miller and Winograd in [18] in 1967. Many algorithms in digital signal processing, numerical linear algebra, discrete methods for solving differential equations, and graph theory can be reformulated as UREs. Much research has been focused on identifying parallelism in UREs which led to efficient implementations in regular processor arrays [27, 28, 29]. For a comprehensive treatment of UREs and related work, see the dissertation of Roychowdhury [24]. All these works exploited parallelism with loop-carried dependences (LCD's) preseved.

Linear recurrences (LRs) form one of the most important subclass of UREs. The linearity in LRs allows a higher degree of parallelism to be extracted from the LRs beyond preseving LCDs, while the non-linearity in general UREs makes it much harder to exploit parallelism beyond LCDs. Kogge and Stone [19] developed the recursive doubling technique for computing the first order linear recurrence system and defined some properties that extend applicability of their technique to a broader class of problems. Their technique assumes an unlimited number of processors. Chen, Kuck and Sameh comprehensively studied parallel evaluation of LR's and BLR's and reported an algorithm[6] for computing mth-order BLR's with a fixed number of processors. Hyafil and Kung [17] established a time bound for parallel evaluation of the Horner expression, which is equivalent to evaluating the last equation in a first-order LR, i.e., evaluating x_N only without having to compute $x_1, ..., x_{N-1}$, Note that the Horner expression only requires the final value of the last equation, while the BLR evaluation computes the solutions of all equations in the recurrences. Gajski [15] improved the time bound of [6] for computing BLR with resource constraints. Recently, Wang and Nicolau [33] further improved on Gajski's results for general BLR's (i.e., mth-order LR's) with resource constraints and in particular found the strict time-optimal schedules

```
for  (i=1;i<=N;i++)
  x[i]=c[i]+a[i]*x[i-1];
```

(a)

```
for  (i=1;i<=N;i++) {
 for  (j=i-m; j<i; j++)
   x[i]=x[i]+a[i][j]*x[j];
 x[i]=x[i]+c[i];
}
```

(b)

Figure 1: (a) first-order linear recurrence code and (b) mth-order linear recurrence code.

for computing 1st and 2nd-order LR's.

Prefix sum is the simplest first-order linear recurrence, i.e., all coefficients in the recurrences equal 1 and no multiplication is needed. It was extensively studied in [21, 14, 30, 23]. The strict time optimal schedule for computing parallel prefix sum with a fixed number of processors (independent of problem size) on CREW PRAM model was first published in [23].

3 Regular Schedule—A Scalable Algorithm for BLR

Definition 3.1 An *mth-order linear recurrence system of N equations* $R\langle N, m\rangle$ computes

$$x_i = c_i + \sum_{j=i-m}^{i-1} a_{ij} x_j, \quad 0 < m < N, \ \ 1 \le i \le N.$$

$R\langle N, m\rangle$ is called *band linear recurrence* (BLR) if m is a fixed number independent of the problem size N.

To facilitate comparison with other algorithms, we assume that the first m values x_1 through x_m of $R\langle N, m\rangle$ are given, and we thus have $R\langle N, m\rangle$ as follows.

$$\begin{aligned} x_1 &= c_1 \\ x_2 &= c_2 \\ &\ldots \\ x_m &= c_m \\ x_k &= c_k + a_{k,k-1}x_{k-1} + a_{k,k-2}x_{k-2} + \ldots + a_{k,k-m}x_{k-m}, \ k > m. \end{aligned}$$

□

The sequential program for the first and higher order linear recurrence can be written as shown in Figure 1. Because of the loop-carried dependences, the two loops above cannot be parallelized into scalable parallel programs with LCD-preserving parallelization techniques.

Given a BLR as shown in Figure 1 the idea of a scalable algorithm is to formulate BLR in terms of matrix chain multiplication. The associativity of matrix chain multiplication enables us to treat the problem as parallel prefix [22, 23], i.e., we can obtain scalable parallel programs by computing *redundant* look-ahead values and producing multiple final results in parallel.

Definition 3.2 [Matrix chain multiplication] The vectors throughout this paper are $m+1$ dimensional. Let $\vec{x}_m = (x_m \; x_{m-1} \; \ldots \; x_1 \; 1)$, initialized to $(c_m \; c_{m-1} \; \ldots \; c_1 \; 1)$, be the vector of initial values. Let $\vec{a}_k = (a_{k,k-1} \; a_{k,k-2} \; \ldots \; a_{k,k-m} \; c_k)$ be a vector of coefficients, and $\vec{x}_k = (x_{k-1} \; x_{k-2} \; \ldots \; x_{k-m} \; 1)$ a vector of results, $k > m$. Let $\vec{e}_{(j)}$ be a vector with the jth element 1 and the rest 0's for $1 \leq j \leq m+1$. Let $A_k = (\vec{a}_k^T \; \vec{e}_{(1)}^T \; \vec{e}_{(2)}^T \; \cdots \; \vec{e}_{(m-1)}^T \; \vec{e}_{(m+1)}^T)$ be an $(m+1)$-by-$(m+1)$ matrix. The computation of $R\langle N, m\rangle$ can be expressed in matrix chain multiplication, for $m < k \leq N$,

$$\vec{x}_k = \vec{x}_m A_{m+1} \ldots A_k.$$

Final value x_k is the first element of the resulting vector $\vec{x}_k$. □

Note that in Definition 3.2,

$$\begin{aligned}
\vec{x}_{k-1} A_k &= (x_{k-1} \; x_{k-2} \; \ldots \; x_{k-m} \; 1)(\vec{a}_k^T \; \vec{e}_{(1)}^T \; \vec{e}_{(2)}^T \; \cdots \; \vec{e}_{(m-2)}^T \; \vec{e}_{(m)}^T) \\
&= (\vec{x}_{k-1}\vec{a}_k^T \; x_{k-1} \; x_{k-2} \; \ldots \; x_{k-m+1} \; 1) \\
&= (x_k \; x_{k-1} \; x_{k-2} \; \ldots \; x_{k-m+1} \; 1).
\end{aligned}$$

In the vector and matrix multiplication above, only the vector product $\vec{x}_{k-1}\vec{a}_k^T$ carries real computation, while the vector products of $\vec{x}_{k-1}\vec{e}_j^T$, $j = 1, 2, \ldots, m-2, m$, merely have its elements shifted one position right with second to last element x_{k-m} shifted out.

The matrix chain multiplication in Definition 3.2 allows us to treat the parallel computation of $R\langle N, m\rangle$ in similar fashion to parallel prefix computation [23]. Different forms of matrix chain multiplication for BLR were used in [6, 17, 15, 33]. However, our Regular Schedule is new and different in its regular and periodic organization of the computation that uses the matrix chain multiplication. Algorithm 3.1 describes the generic Regular Schedule in terms of matrix chain multiplication for parallel computing $R\langle N, m\rangle$. It is also called *Square Schedule* as each iteration of its outermost loop produces p^2 final results, where p is the number of processors. To facilitate presentation and analysis, Algorithm 3.1 can be understood (described) on a concurrent-read-exclusive-write (CREW) parallel machine model (PRAM). Issues of mapping the algorithm onto real vector supercomputers will be addressed in Section 4. A CREW PRAM machine is assumed to have p parallel processors interconnected with a global memory. Each operation is assumed to take a unit time. A memory location can be read simultaneously, i.e., broadcasting of data is allowed, but it can only be written by a single processor at a time.

Algorithm 3.1 **[Regular schedule]**

Input: Matrix chain multiplication of $R\langle N, m\rangle$, $\vec{x}_k = \vec{x}_m A_{m+1} \ldots A_k$, $m < k \leq N$, and $p > m$ parallel processors.

Output: final results x_1, x_2, ..., x_N.

```
regular_schedule_for_BLR (N, m, p)
1. for $i = m+2$ to $N$ step $p^2$ do
```

```
2.      for $j = i$ to $(i + p^2 - p)$ step $p$ do in parallel
3.          for $k = j$ to $(j + p - 2)$ do
4.              $A_k = A_k A_{k-1}$;
            end for
        end for
5.      for $j = i - 1$ to $(i - 1 + p^2 - p)$ step $p$ do
6.          for $k = 0$ to $(p - 1)$ do in parallel
7.              $\vec{x}_{j+k} = \vec{x}_{j-1} A_{j+k}$;
            end for
        end for
    end for
```

□

The outermost loop labeled "1" in Algorithm 3.1 partitions the problem $R\langle N, m\rangle$ into chunks of size p^2, called *period*, given p processors. Each iteration of the outermost loop labeled "1" computes p^2 final values of a period, called *current period*. In an iteration of the outermost loop labeled "1", two inner-loops labeled "2" and "5" in sequence partition a period further into p sections of size p each, a section called a *redundant tree*. The first inner-loop labeled "2" computes redundant values: its outer-loop computes in parallel the redundant values in p sections of the current period and its inner-loop labeled "3" computes sequentially the redundant values in each redundant tree. The second inner-loop labeled "5" of the outermost loop computes the final values of the current period. Its outer-loop travels sequentially computing the final values of each of the p sections in the period, and its inner-loop labeled "6" (a do-all type of loop) computes all final values in a section in parallel.

Example 3.1 Consider computing $R\langle N, 2\rangle$ on $p = 4$ processors for $N = jp^2 + 2$ and $j \geq 1$. The matrix chain multiplication for $R\langle N, 2\rangle$ can be written as follows, where $2 < k \leq N$,

$$[x_k \; x_{k-1} \; 1] = [x_2 \; x_1 \; 1] \begin{bmatrix} a_{31} & 1 & 0 \\ a_{32} & 0 & 0 \\ c_3 & 0 & 1 \end{bmatrix} \begin{bmatrix} a_{42} & 1 & 0 \\ a_{43} & 0 & 0 \\ c_4 & 0 & 1 \end{bmatrix} \begin{bmatrix} a_{53} & 1 & 0 \\ a_{54} & 0 & 0 \\ c_5 & 0 & 1 \end{bmatrix} \cdots \begin{bmatrix} a_{k,k-2} & 1 & 0 \\ a_{k,k-1} & 0 & 0 \\ c_k & 0 & 1 \end{bmatrix}. \qquad (1)$$

The computation tree for a period is shown in Figure 2. The outermost loop (labeled "1") in Algorithm 3.1 iterates on periods of size $p^2 = 4^2 = 16$ and an iteration produces all 16 final values (the X's in Figure 2) of a period in the following fashion. The first inner-loop (labeled "2") is a do-all loop; it sends all of its $p = 4$ iterations, each computing $p - 1$ redundant values of a redundant tree, to execute in parallel on $p = 4$ processors. For example, R_4, R_5, R_6 are the three matrices of redundant values on the first redundant tree in Figure 2. The innermost loop labeled "3" in the algorithm computes $p - 1 = 3$ redundant matrix multiplications sequentially, thus it takes the time for $(p - 1)$ redundant matrix multiplications to complete the redundant computation in a period since the $p = 4$ redundant trees in a period are computed in parallel.

The second inner-loop labeled "5" in Algorithm 3.1 computes all final values in a period using redundant values produced by the first inner-loop of the algorithm. Its outer-loop labeled "5" is a sequential loop, iterating on $p = 4$ redundant trees in a period, i.e., the second innermost loop labeled "6" produces the final results on

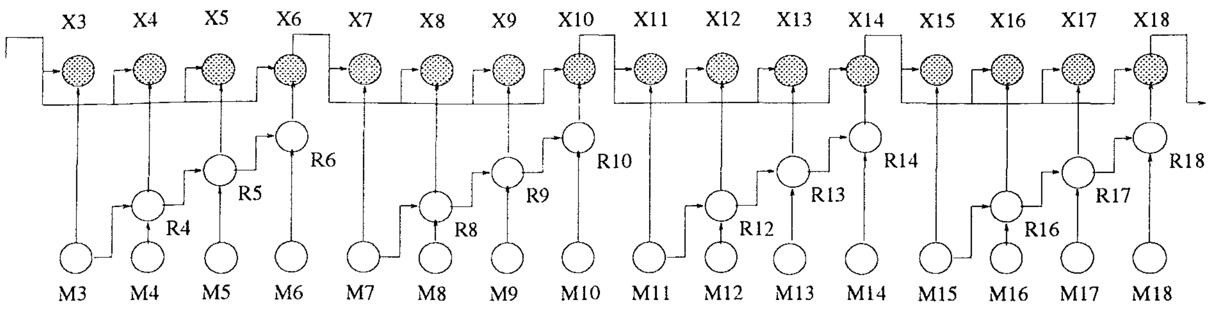

Figure 2: A period of the computation tree for 2nd-order LR on $p = 4$ processors.

top of a redundant tree in parallel by computing a vector-by-matrix multiplication. Since there are p redundant trees in a period, the time for computing all final values in a period equals $pT_{\mathrm{v-by-m}}$ (where $T_{\mathrm{v-by-m}}$ is the time (steps) for a vector-by-matrix multiplication). □

The following theorem gives the time bound of the regular schedule when the number of processors $p > m$. For $p \leq m$, a direct application of the column-sweep algorithm [20] will perform the computation in time close to optimal.

Theorem 3.1 Let $N = jp^2 + m$, where $j \geq 1$. The regular schedule (Algorithm 3.1) computes $R\langle N, m\rangle$ on p processors, where $p > m$ is a constant independent of N, in time

$$T_p = \frac{N}{p}((2m^2 + 3m) - \frac{m+1}{2p}(2m^2 + m)). \tag{2}$$

Proof: The execution time of the algorithm for $R\langle N, m\rangle$ is the execution time of a period of size p^2 multiplied by the number of periods in N. The computation of the results in a period consists of two phases. In the first phase, p processors compute in parallel operations specified by the p redundant trees and each processor sequentially multiplies $(p-1)$ coefficient matrices for $p \geq m+1$, which takes $m(2m+1)(p-(m+1)/2)$ (this can be shown by induction on m). In the second phase, p processors compute in parallel p final results on top of each of the p redundant trees, which takes $T_{\mathrm{v-by-m}} = 2m$ steps (this can also be shown by induction on m). Since there are p redundant trees in a period, the second phase takes $pT_{\mathrm{v-by-m}} = 2mp$ steps. Hence, for $p \geq m+1$,

$$\begin{aligned} T_p(\text{period of size}\, p^2) &= (\text{time for multiplying}\,(p-1)\,\text{coefficient matrices}) + pT_{\mathrm{v-by-m}} \\ &= m(2m+1)(p-(m+1)/2) + 2mp. \end{aligned}$$

Therefore,

$$\begin{aligned} T_p(\text{regular schedule for}\, R\langle N, m\rangle) &= (\text{number of periods})T_p(\text{period of size}\, p^2) \\ &= \frac{N}{p^2}(m(2m+1)(p-(m+1)/2) + 2mp) \\ &= \frac{N}{p}((2m^2+3m) - \frac{m+1}{2p}(2m^2+m)). \end{aligned}$$

□

By Theorem 3.1, Algorithm 3.1 is scalable in terms of number of processors p. The regular schedule can be derived using similar techniques for deriving the Harmonic Schedule presented in [33]. Gajski's algorithm [15] achieves the same execution time as the Regular Schedule, under the model of concurrent-read-and-exclusive-write (CREW) parallel random access machine (PRAM), but uses a different formulation and organizes the computation differently. When mapped onto real machines, the Regular Schedule will perform better since it has better utilization of locality of reference. The algorithm for first-order LR in [7] reduces to the regular schedule only for first-order LR, but the blocked first-order formulation for mth-order LR ($m \geq 2$) described in [7] differs from the regular schedule and has longer execution time. The Regular Schedule is not time-optimal under the CREW PRAM model—-both its execution time and program-space efficiency are inferior to many other schedules derived using Harmonic Schedule [33]. However, its regular organization of the computation makes it extremely well suited in practice for vector supercomputers and massively parallel computers. Moreover, the fact that the Regular Schedule organizes the computation in *periods* of the same regular structure makes it a good choice for real-time applications (e.g., digital signal processing) where algorithms in [19, 6, 17, 15] do not, as will be explained in Section 6.

4 Implementing the Regular Schedule on a Vector Computer

Although the regular schedule is scalable, the real machine constraints (e.g., different styles of parallelism exploitation, memory access latency, inter-processor communications and operating system of the machines) may greatly affect the performance. This section presents our implementation of the regular schedule on a vector supercomputer and discusses how to obtain maximum memory throughput using both analytical and experimental methods.

Our experiments were performed on Convex C240, installed on the campus of University of California at Irvine (UCI). Convex C240 is representative of a class of vector supercomputers such as Cray Y-MP, and Fujutsu VP's.

Before discussing programming the regular schedule, we give an overview of relevant portion of Convex C240 architecture [9]in Figure 3. Convex C240 is a shared memory MIMD vector parallel computer. It has 4 CPU's, each performing its own instruction stream, interconnected to a global main memory of upto 2 gigabytes (UCI installed 1 gigabytes) via the memory bus. Its memory has five ports allowing simultaneously accesses by all 4 CPU's plus an I/O subsystem. Each CPU consists of an address/scalar processor (ASP) and a vector processor (VP). A VP uses 8 vector registers and 3 pipelined functional units to operate on data, a load/store unit, an arithmetic/logic unit and multiply unit. No data cache is provided in a VP. Each functional unit has its own micro controller to control the sequence of actions required to perform the instructions dispatched to it, and thus the 3 functional units may execute different instructions concurrently. Vector *chaining* is provided: once the result of an instruction has been written into the vector register file, another functional unit may select that vector register to be used as an operand for the instruction it is performing. Each vector register has 128 vector

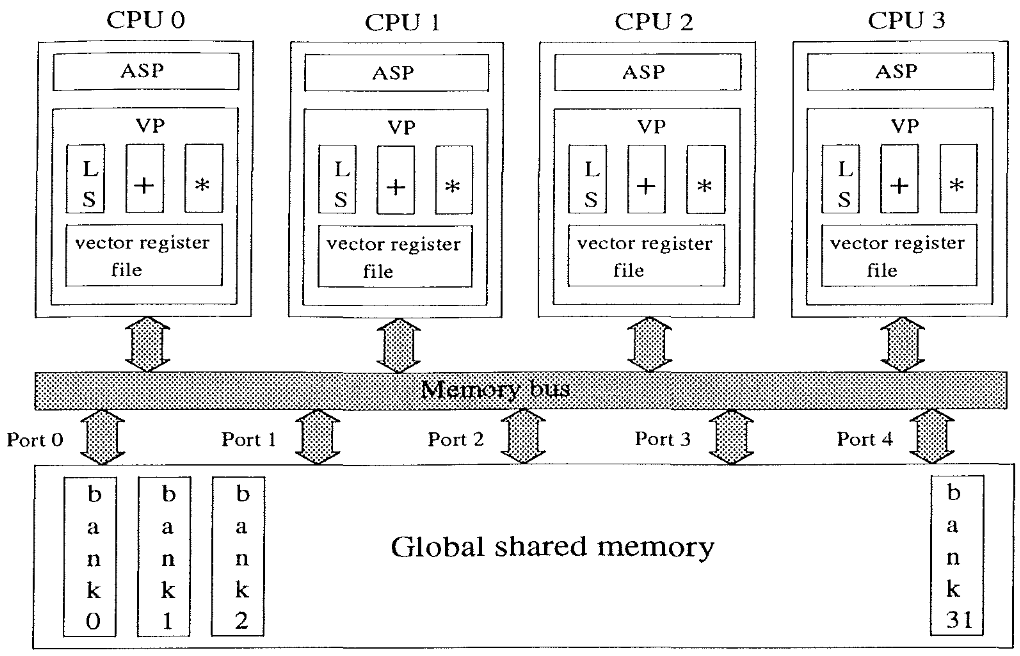

Figure 3: A programmer's overview of Convex C240 architecture.

elements with each element having 64 bits. The main memory consists of a total of 32 independent memory banks. Memory *interleaving* is provided on the 32 memory banks[1] to support a pipelined access rate of one long word (64 bits) read/write cycle per clock.

Implementing the regular schedule (Algorithm 3.1) on a vector computer with the architecture above involves mapping the processors in Algorithm 3.1 onto the processing elements of the vector computer and placing the data across the memory banks so as to obtain maximum memory throughput. In our mapping, a processor in Algorithm 3.1 corresponds to a vector register element in the vector processor. The pipelined functional units provide the major mechanism to exploit parallelism in programs in a vector processor and depend on the vector registers supplying data to be kept fully utilized. Moreover, the number of data elements to be fetched into a vector register (i.e., the *vector length*) can be specified and controlled in the source code. Thus the number of processing elements in a vector computer is equal to the number of vector processors multiplied by vector register length. We shall map Algorithm 3.1 to up to 128 processing elements corresponding to a single vector processor. One can easily map our algorithm to multiple vector processors in the same fashion. However, this mapping does not mean that speedup on the order of ((# of CPU's) $\times$ (vector register length)) can be obtained. In this paper, we are only concerned with parallel programming in high-level languages and performance tuning in source code. Issues such as utilization of chaining will be handled by the Convex compiler.

With this mapping scheme, it seems intuitive that using full vector register length 128, i.e., mapping our algorithm to 128 processing elements, would yield the best performance. However, such intuition is deceiving, because of a special property of our algorithm and memory interleaving. The question of choosing the optimal register length will be addressed in the remainder of this section.

[1]The *degree* of memory interleaving is thus 32.

As an example, the portion of the source code in Convex C of the regular schedule for 2nd-order LR is given below. The labels corresponding to the labels in Algorithm 3.1 are added for convenience. The coefficient matrices are stored in a sparse format. Referring to Equation 1 of $R\langle N, m\rangle$ in Example 3.1, coefficients $a_{k,k-2}$ (those in the top row of the coefficient matrices) are stored in array $a[]$, coefficients $a_{k,k-1}$ (those in the second row of the coefficient matrices) are stored in array $b[]$, constants c_k are stored in array $c[]$.

Our C code expands Algorithm 3.1 (which is expressed in terms of matrix multiplications) with arithmetic operations. The correspondence of Algorithm 3.1 to the C code for $R\langle N, 2\rangle$ is: loop 1 to L1, loop 2 to L2.1 and L2.2, loop 3 to L3, statement 4 to code blocks L4.1 and L4.2, loop 5 to L5, loop 6 to L6 and statement 7 to L7. Note that loop 2 of Algorithm 3.1 is split into two loops L2.1 and L2.2 in our C code. That is because in a chain multiplication of $p > 2$ coefficient matrices for $R\langle N, 2\rangle$, the first matrix multiplication has 5 arithmetic operations as expanded in code block L4.1 and the second through $(p-1)$th matrix multiplications have 10 arithmetic operations each as expanded in code block L4.2. In general, for a chain multiplication of $p > m$ coefficient matrices for $R\langle N, m\rangle$, the jth $(1 \leq j \leq m-1)$ has $j(2m+1)$ operations respectively, and the $(j+1)$th through $(p-1)$th matrix multiplications have $m(2m+1)$ operations each.

```
main ()
{
  /*
   * regular schedule for computing 2nd-order linear recurrence.
   */
  #define N1 1000000 /* problem size. */
  #define RTH  121   /* redundant tree height. */
  #define PI RTH*RTH /* period size. */
  #define N2  N1/PI*PI+PI+2 /* padding to make array size a multiple of
                      RTH*RTH plus 2. */
  register int k, m, i, j, l;
  double x[N2], a[N2], b[N2], c[N2], ar[N2], br[N2];

  ....../* initialize arrays a[N2], b[N2], c[N2]. */

  x[1]=c[1];
  x[2]=c[2];
 L1: for  (m=2; m<=N2; m+=PI) /* do computation of one period at a time. */
       {
         /* do simultaneously 1st matrix multiplications on p redundant trees
            in a period.  vectorizable. */
 L2.1:   for  (i=m+1; i<=m+1+RTH* (RTH-1); i+=RTH)
           {
 L4.1:       ar[i+1]=a[i]*a[i+1]+b[i+1];
             br[i+1]=b[i]*a[i+1];
             c[i+1]=c[i]*a[i+1]+c[i+1];
             a[i+1]=ar[i+1];
             b[i+1]=br[i+1];
           }

         /* do sequentially 2nd thru  (p-1)th coeff. matrix multiplications
            on p redundant trees in a period. */
 L2.2:   for  (j=m+2; j<=m+RTH-1; j++)
           /* do in parallel jth coeff. matrix multiplications on p
              redundant trees in a period.  vectorizable. */
 L3:       for  (k=j; k<=j+RTH* (RTH-1); k+=RTH)
             {
 L4.2:         ar[k+1]=a[k]*a[k+1]+a[k-1]*b[k+1];
               br[k+1]=b[k]*a[k+1]+b[k-1]*b[k+1];
               c[k+1]=c[k]*a[k+1]+c[k-1]*b[k+1]+c[k+1];
               a[k+1]=ar[k+1];
              b[k+1]=br[K+1];
            }

        /* do sequentially final results on p redundant trees in a period. */
L5:     for  (i=m; i<=m+RTH* (RTH-1); i+=RTH)
          /* do in parallel final results on current redundant tree.
             vectorizable. */
L6:       for  (l=1; l<=RTH; l++)
L7:         x[i+l]=c[i+l]+a[i+l]*x[i]+b[i+l]*x[i-1];
      }
}
```

As commented in our C code, loop L2.1, L3 and L6 can be vectorized by the Convex C parallelizing compiler. The symbolic constant RTH (as defined in the code) in these three loops will serve as the vector length. In Loop L2.1 and L3, RTH will be used as the vector stride measured in long words of 8 bytes on Convex C200 Series (the stride is 8×RTH measured in bytes as in Convex assembly code). That is because consecutive iterations in these two loops operate on data that reside in the global memory (RTH $\pmod{32}$) banks apart (recall that each iteration does a coefficient matrix multiplication and all RTH iterations work on RTH redundant trees in parallel). Hence, RTH as the vector stride will determine how memory accesses are interleaved among the 32 memory banks, which is a key factor affecting the performance. The following claim provides an analytical method for choosing an optimal vector stride based on the architectural features given.

Claim 4.1 Given that

1. the startup time for a vector load instruction is $T_{startup}$ (at the end of last cycle of the startup duration, the first element of the vector is available in the vector register),
2. the time for a *complete round* [2] of accesses to memory banks is equal to $T_{round} = \frac{\text{lcm}(I_m,S_v)}{S_v}$, where I_m is the degree of memory interleaving, S_v is vector stride measured in long word and lcm is least common multiple, and
3. d is the minimum number of cycles between loads to the same memory bank,

the time for a vector load with vector stride S_v and vector length L_v is

$$\begin{aligned}
T_{vld} &= T_{startup} + (\text{time for complete rounds of accesses}) + (\text{time for trailing incomplete round of accesses}) \\
&= T_{startup} + \\
&\quad + \left\lfloor \frac{L_v S_v}{\text{lcm}(I_m,S_v)} \right\rfloor \left(\frac{\text{lcm}(I_m,S_v)}{S_v} + \max\{d, \frac{\text{lcm}(I_m,S_v)}{S_v}\} - \frac{\text{lcm}(I_m,S_v)}{S_v} \right) + \qquad (3) \\
&\quad + \left(L_v - \left\lfloor \frac{L_v S_v}{\text{lcm}(I_m,S_v)} \right\rfloor \frac{\text{lcm}(I_m,S_v)}{S_v} \right) + \\
&\quad + \begin{cases} -(\max\{d, \frac{\text{lcm}(I_m,S_v)}{S_v}\} - \frac{\text{lcm}(I_m,S_v)}{S_v}) & \text{if } L_v \equiv 0 \pmod{\frac{\text{lcm}(I_m,S_v)}{S_v}} \text{ and } \lfloor \frac{L_v S_v}{\text{lcm}(I_m,S_v)} \rfloor \geq 1 \\ 0 & \text{otherwise} \end{cases}
\end{aligned}$$

[2]If a VP has fetched in sequence data from memory banks $B_{S_v \bmod I_m}$, $B_{2S_v \bmod I_m}$, ..., $B_{(k-1)S_v \bmod I_m}$ such that $kS_v \equiv S_v \bmod I_m$, then the VP is said to have done a *complete round* of accesses.

All time intervals are measured in machine clock cycles. Our architecture can achieve a rate of one vector element per cycle with full memory interleaving during pipelined access stage (for example, in Convex C200 Series and several other vector computers). This claim is also applicable to vector store instructions.

Proof: The idea of Formula 3 is self-explanatory from its first line.

The term $(\max\{d, \frac{\text{lcm}(I_m,S_v)}{S_v}\} - \frac{\text{lcm}(I_m,S_v)}{S_v})$ gives the waiting interval, explained as follows. When the time for issuing a round is less than the time for completing the first access in a round (i.e., $d > \frac{\text{lcm}(I_m,S_v)}{S_v}$), the first fetch (thus the following fetches) in the successive round have to wait $(\max\{d, \frac{\text{lcm}(I_m,S_v)}{S_v}\} - \frac{\text{lcm}(I_m,S_v)}{S_v}) = (d - \frac{\text{lcm}(I_m,S_v)}{S_v})$ cycles for the current round to finish. Else, when $d \leq \frac{\text{lcm}(I_m,S_v)}{S_v}$, there is no waiting by the successive rounds, i.e., $(\max\{d, \frac{\text{lcm}(I_m,S_v)}{S_v}\} - \frac{\text{lcm}(I_m,S_v)}{S_v}) = 0$.

The time for all complete rounds of accesses is equal to the number of complete rounds multiplied by the number of cycles taken by a complete round, given by

$$\left\lfloor \frac{L_v S_v}{\text{lcm}(I_m, S_v)} \right\rfloor \left(\frac{\text{lcm}(I_m, S_v)}{S_v} + \max\{d, \frac{\text{lcm}(I_m, S_v)}{S_v}\} - \frac{\text{lcm}(I_m, S_v)}{S_v} \right),$$

minus a waiting interval because all complete rounds except for the first one may need a waiting interval, when $\lfloor \frac{L_v S_v}{\text{lcm}(I_m,S_v)} \rfloor \geq 1$. Note that the startup cycles for a round (until the first data is available) are not counted above because they get overlapped. The time for the trailing incomplete round of accesses is given by

$$(L_v - \left\lfloor \frac{L_v S_v}{\text{lcm}(I_m, S_v)} \right\rfloor \frac{\text{lcm}(I_m, S_v)}{S_v}),$$

plus the waiting time when there is a preceding complete round. □

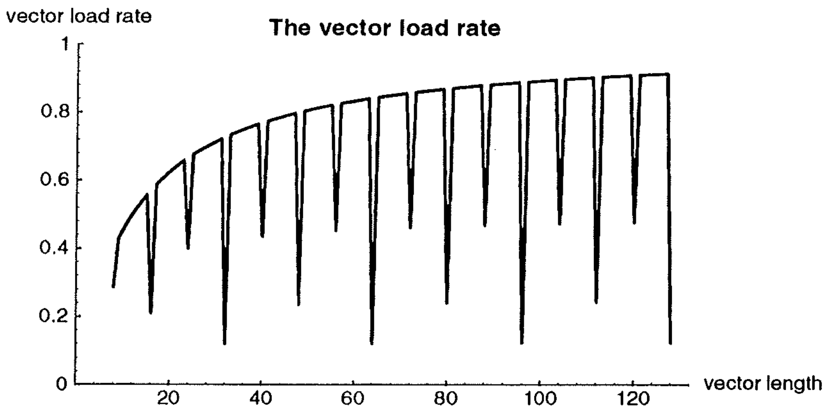

Figure 4: The vector load rate (number of vector elements fetched per cycle) for a vector load instruction vs. vector length for the regular schedule on Convex C240 based on our analysis.

Plugging into Formula 3 the Convex C240 parameters [8], $d = 8$ cycles and $T_{startup} = 12$ cycles and $L_v = S_v =$RTH, we get the time for a vector load. In Figure 4, we plotted the curve for vector load rate L_v/T_{vld}, measured by number of vector elements fetched per cycle, for L_v of 8 through 128. Figure 4 indicates that when the vector length is a multiple of 32, 16 (but not of 32) and 8 (but not of 16 nor 32), the vector load rate drops to the lowest, the second lowest and the third lowest. The peak vector load rate is obtained for $L_v = 127$. The curve also indicates that when accesses can be fully interleaved, the vector load rates are virtually indistinguishable in the range between $L_v = 127$ and $L_v = 121$.

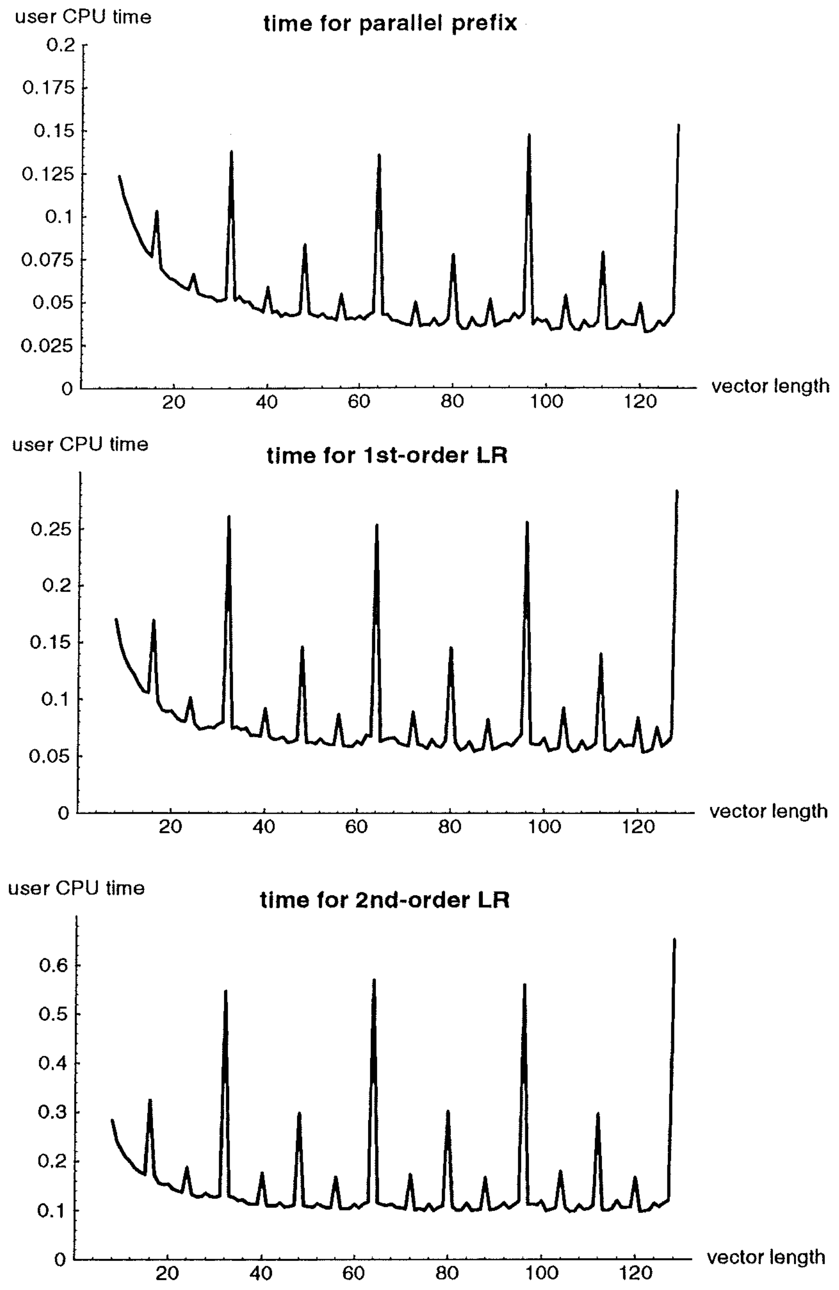

Figure 5: The user CPU time (in seconds) vs. vector length for parallel prefix (top), 1st-order (middle) and 2nd-order (bottom) LR using Regular Schedule measured on Convex C240.

The other important aspect that affects choosing the optimal vector length (or vector stride) is how the operating system (OS) works. ConvexOS [8], like UNIX, is not a real time system. Factors such as context switching, paging and interrupt servicing may also significantly affect the actual running time with different values of L_v. Since these factors are far too complex to model analytically, we have run experiments to determine the optimal vector register length. In Figure 5, we show the curves for user CPU time vs. vector lengths of 16 through 128 for 3 programs (parallel prefix, 1st-order LR and 2nd-order LR) with 100,000 equations on Convex C240. The experimental results in Figure 5 are consistent with our analytical results in that the user CPU time with vector strides of multiple of 32, 16 (but not 32) and 8 (but not 32 nor 16) were the longest, the second longest and the third longest respectively. All 3 programs obtained the shortest execution time with vector length 121.

5 Programming Based on Regular Schedule

For standard BLR's (band linear recurrences), the results in Section 3 and 4 can be applied directly to generate parallel programs for vector supercomputers. There are many application programs in which BLR's are mixed with other computations. The mixture of recurrence with other code poses more challenges in applying the Regular Schedule to it than to the standard BLR. Sometimes even recognition of the core recurrence becomes non-trivial. However, it is in this kind of situation that fully parallelizing the code will yield better performance than separating the core recurrence from other code and replacing it with some prescribed library routine. In this section, we outline our approach, based on the Regular Schedule, for parallelizing such programs in this section. Our approach can be used both for parallelizing programs with BLR's and for compiler parallelization of such programs in conjunction with recurrence recognition techniques.

Our approach for parallelizing programs with BLR's utilizing the Regular Schedule consists of 3 steps:

---

1. Recognizing the BLR's and the dependence of other computations on the core BLR structure in the program given;
2. Construct the matrix chain multiplication for the BLR's and their dependent computation;
3. Applying the Regular Schedule.

There exist several methods for compiler recognition of recurrences in loops that can be used in Step 1 of our approach described above (since this is not the topic of this paper, we briefly overview these works). Banerjee et al [3] showed how the data dependence graph can be used to isolate recurrences so that pattern matching techniques can be applied. Ammarguellat and Harrison [2] put forth a method for recognizing recurrence relations automatically. Pinter and Pinter [25] gave a graph-rewriting technique for recognizing recurrences by unfolding loops. Tanaka [31] provided a framework for both recognition and vectorization of 1st-order linear recurrence. Callahan [5] gave an algebraic approach to combining bounded recurrences and generating parallel code.

We illustrate our approach by going through it with Livermore Kernel 19— general linear recurrence equations as shown below.

```
main ()
```

```
{   /* Livermore Kernel 19---general linear recurrence equations. */
    long argument , k , l , i, kb5i;
    double sa[N], sb[N], b5[N], stb5;

    kb5i = 0;

    for  ( k=0 ; k<N ; k++ ) {
        b5[k+kb5i] = sa[k] + stb5*sb[k];
        stb5 = b5[k+kb5i] - stb5;
    }
    for  ( i=1 ; i<=N ; i++ ) {
        k = 101 - i ;
        b5[k+kb5i] = sa[k] + stb5*sb[k];
        stb5 = b5[k+kb5i] - stb5;
    }
}
```

The first loop can be rewritten into an equivalent loop as follows:

```
for  ( k=0 ; k<N ; k++ ) {
    b5[k] = sa[k] + stb5[k]*sb[k];
    stb5[k+1] = sa[k] + stb5[k]* (sb[k] - 1);
}
```

The scalar `stb5` in the original loop has been privatized [13] into array `stb5[N+1]`, which enables us to recognize that the core recurrence is on the second statement in the new loop. We then construct a matrix chain multiplication for the core recurrence as follows:

$$[stb5[k]\ 1] = [stb5[1]\ 1]\begin{bmatrix} sb[1]-1 & 0 \\ sa[1] & 1 \end{bmatrix}\begin{bmatrix} sb[2]-1 & 0 \\ sa[2] & 1 \end{bmatrix}\begin{bmatrix} sb[3]-1 & 0 \\ sa[3] & 1 \end{bmatrix}\cdots\begin{bmatrix} sb[k]-1 & 0 \\ sa[k] & 1 \end{bmatrix}. \quad (4)$$

where $1 < k \leq N+1$.

Once we have constructed the matrix chain multiplication for the core recurrence, we can easily apply our Regular Schedule to the chain for computing `stb5[k]`. We organize the core recurrence with the computation around it according to locality of reference. The resulting loop is given as follows.

```
main ()
{
  /* 11/14/92
   * Livermore Loop Kernel LL19 ---- general linear recurrence equations
   * my parallel program uses the regular schedule.
   */
  register int k, m, i, j, l;
  double sa[N2], sb[N2], b5[N2], stb5[N2], ta[N2], tb[N2];

  ...... /* initialization of arrays. */
  for  (m=0; m<=NPI; m+=PI)
    {
      for  (i=m+1; i<=m+1+PI-RTH; i+=RTH)
        {
          ta[i]=sa[i]; tb[i]=sb[i]-1;
        }

      for  (j=m+2; j<=m+RTH; j++)
          for  (k=j; k<=j+PI-RTH; k+=RTH)
            {
              ta[k]= (sb[k]-1)*ta[k-1]+sa[k];
              tb[k]= (sb[k]-1)*tb[k-1];
            }
      for  (i=m; i<=m+PI-RTH; i+=RTH)
        for  (l=1; l<=RTH; l++)
          {
            stb5[i+l]=tb[i+l]*stb5[i]+ta[i+l];
            b5[i+l]=sa[i+l]+stb5[i+l]*sb[i+l];
          }
    }
  ...... /* second loop. */
}
```

Similarly, the second loop can be rewritten into an equivalent loop and the matrix chain multiplication for the new loop can be constructed, except that the induction variable in the second loop decrements.

In our experiment, all 6 benchmarks containing BLR intermixed with other code (1st and 2nd-order infinite-impluse response filters, tridiagonal elimination below diagonal, general linear recurrence equations, tridiagonal and pentadiagonal linear system solver) were programmed using our approach as illustrated in this section and substantial gains in performance were obtained for all of them. We have preliminary evidence that our Regular Schedule can facilitate parallel programming code containing BLR intermixed with other code in an integrated fashion. Such a comprehensive treatment, where locality of reference is better observed, will gain us more performance than treating BLR and its dependent code in a loop separately, as evidenced by our experimental results. More detail of our approach is given in [34].

6 Results

We show in Table 1 the CPU performance of benchmark programs containing BLR implemented using the BLAS routines of the Convex C240 VECLIB [11] in comparison with the same programs implemented using our regular schedule. The CPU performance was measured by the user CPU time in seconds on a single processor of Convex C240. Programs using Regular Schedule were implemented in Convex C. The BLAS routines [11] of the Convex C240 VECLIB are coded in assembly and highly optimized to the architecture details, and are the fastest linear algebra routines installed at UCI. In comparison, the corresponding routines in the Convex SCILIB (scientific library) perform noticeably worse (the Convex SCILIB consists of a collection of FORTRAN-callable routines identical in name and operation to those found in Cray Research Inc.'s UNICOS Math and Scientific Library V5.0 and optimized for the Convex family of supercomputers).

The first column lists the names of the programs containing BLR. Each program was tested for problem sizes of 1 through 4 million (which covers the typical size of some large scale computations [1]). Double precision (long word of 8 bytes) floating-point numbers were used for all implementations. The column titled "BLAS routine used" lists the BLAS routines used in coding the benchmark programs. The columns titled "user CPU time of programs using BLAS" and "user CPU time of programs using regular schedule" give the user CPU time of programs using BLAS routines and using our regular schedule respectively. The last column calculates the improvements in performance of programs using the regular schedule over the same programs using BLAS. To ensure a fair comparison, each pair of implementations (one using BLAS and one using the Regular Schedule) for a benchmark were submitted to a single batch job in order to run them in the same system load. A reported

program	problem size N	BLAS routine used	user CPU time of programs using BLAS	user CPU time of programs using reg. sch.	improvement= $(\frac{BLAS\ time}{Our\ time} - 1) \times 100$
Livermore Kernel 11— first sum (parallel prefix)	1M	dflrlc_	0.2717208	0.2319761	17.13%
	2M	dflrlc_	0.5541139	0.4570663	21.23%
	3M	dflrlc_	0.8211107	0.6857052	19.75%
	4M	dflrlc_	1.0998956	0.9043271	21.63%
1st-order variable coefficient linear recurrence	1M	dflrlp_	0.4384309	0.3741395	17.18%
	2M	dflrlp_	0.9645480	0.7856930	22.76%
	3M	dflrlp_	1.4249004	1.1785394	20.90%
	4M	dflrlp_	1.9071744	1.5490257	23.12%
2nd-order variable coefficient linear recurrence	1M	dslr3_	1.1777273	0.9467815	24.40%
	2M	dslr3_	2.4732273	1.9313057	28.05%
	3M	dslr3_	3.8393909	2.9726960	29.15%
	4M	dslr3_	5.1466683	3.9572833	30.05%
1st-order constant coefficient linear recurrence	1M	dflrlc_	0.2715408	0.2361474	14.97%
	2M	dflrlc_	0.5015998	0.4307869	16.44%
	3M	dflrlc_	0.8279383	0.7015832	18.01 %
	4M	dflrlc_	1.0907467	0.9137528	19.37 %
2nd-order constant coefficient linear recurrence	1M	dslr3_	1.1059383	0.4539557	143.62%
	2M	dslr3_	2.1921038	0.8800565	149.09%
	3M	dslr3_	3.3299868	1.3073640	154.71%
	4M	dslr3_	4.4565868	1.7151933	159.83%
1st-order infinite-impulse response (IIR) digital filter	1M	dflrlc_	0.4922081	0.3805755	29.33%
	2M	dflrlc_	0.9748273	0.7471519	30.47%
	3M	dflrlc_	1.4668962	1.1153408	31.52%
	4M	dflrlc_	1.9444695	1.4618972	33.01%
2nd-order infinite-impulse response (IIR) digital filter	1M	dslr3_	1.11110735	0.5384003	106.37%
	2M	dslr3_	2.1928480	1.0609556	106.69%
	3M	dslr3_	3.2946523	1.5891628	107.32%
	4M	dslr3_	4.3767078	2.1047936	107.94%
Livermore Kernel 5— tridiagonal elimination below diagonal	1M	dflrlp_	0.7994302	0.4843439	65.05%
	2M	dflrlp_	1.7427578	1.0074353	72.98%
	3M	dflrlp_	2.6141367	1.5208398	71.88%
	4M	dflrlp_	3.4695270	2.0390878	70.15%
Livermore Kernel 19— general linear recurrence equations	1M	dflrlp_	2.6560849	2.1149698	25.58%
	2M	dflrlp_	5.5512174	4.3145384	28.66%
	3M	dflrlp_	8.3932282	6.5352567	28.43%
	4M	dflrlp_	10.9961915	8.5021786	29.33%
tridiagonal linear system solver (positive definite)	1M	dftsl_	2.6014798	2.2223868	17.06%
	2M	dftsl_	5.1509300	4.3114304	19.47%
	3M	dftsl_	7.8304542	6.5782649	19.03%
	4M	dftsl_	10.6140376	8.7562039	21.21%
pentadiagonal linear system solver	1M	dgbfa_, dgbsl_	43.2499651	4.4920359	862.81%
	2M	dgbfa_, dgbsl_	84.2986218	8.7232152	866.37%
	3M	dgbfa_, dgbsl_	128.134129	13.2161077	869.53%
	4M	dgbfa_, dgbsl_	165.5952909	17.0281436	872.48%

Table 1: Performance measured by user CPU time (in seconds) of programs based on the regular schedules in comparison with programs based on BLAS routines of the Convex C240 VECLIB.

user CPU time for a program is the average of 100 runs. The vector length 121 for vector register was used in all our implementations.

Our benchmarks are among the most frequently used code (or code segments) containing BLR in scientific and engineering computations (3 Livermore Kernels are also found to be such kind of code). Note that, for benchmark Livermore Kernel 11—first sum (i.e., parallel prefix), we compare the program implemented using BLAS routine "dflrlc_" with the program based on Regular Schedule, where "dflrlc_" computes 1st-order constant coefficient LR (which involves multiplication whereas parallel prefix does not). This is a fair comparison since the performance difference between involving multiplications and no multiplications is negligible because the multiplications and additions in "dflrlc_" are well chained.

We used "dslr3_", the BLAS routine for 2nd-order variable coefficient LR, in implementing 2nd-order constant coefficient LR (which is the best that can be done using BLAS), since BLAS does not provide routines for constant coefficient LR beyond 1st-order. The performance gain (in range of 100%) of Regular Schedule over BLAS implementation for 2nd-order constant coefficient LR is large because "dslr3_" does not allow pre-computation of partial products of coefficient whereas both "dflrlc_" and Regular Schedule do. For the same reason, large performance gain (in range of 100%) of the Regular Schedule versus the BLAS implementation for 2nd-order infinite-impluse response (IIR) filter was observed.

The 1st-order and 2nd-order IIR filters are widely used in digital signal processing (DSP) applications. They contain 1st-order and 2nd-order constant coefficient LR's with some dependent code respectively. The Regular Schedule is extremely well suited for this type of real-time DSP computation, because the inputs to filters are usually grouped in windows to feed the parallel processors and this matches the period organization of the Regular Schedule.

Our tridiagonal solver using Regular Schedule is based on the LU decomposition method [22], and is numerically stable for userful classes of tridiagonal matrices including positive definite and diagonally dominant. We compared our solver against the BLAS tridiagonal solver "dftsl_".

Our implementation of pentadiagonal solver using Regular Schedule is based on the parallel prefix formulation in [22] and is numerically stable for some useful classes of pentadiagonal matrices. We implemented the BLAS version using "dgbfa_" and "dgbsl_", which respectively factors a band coefficient matrix and solves the equation. Although the BLAS version is numerically stable for more classes of pentadiagonal matrices than the prefix formulation, the large performance gain (in the range of 800%) of the Regular Scheudle over the BLAS implementation makes it a feasible choice for the classes of matrices for which it is stable.

We can observe the following two phenomena in the results. First, when the order of LR increases, the performance improvements of Regular Schedule over the BLAS routine also increase, as demonstrated by the variable coefficient LR's, constant coefficient LR's and infinite-impluse response (IIR) digital filters. The performance gain of Regular Schedule over BLAS implementation for higher order LR's would be larger, since BLAS does not provide routines for variable coefficient LR's beyond 2nd-order nor for constant coefficient LR's beyond 1st-order, and thus higher order LR's have to be implemented using BLAS routines for 1st-order or 2nd-order LR's. Second, for benchmarks containing BLR and dependent code such as Livermore Kernel 5 and Livermore Kernel 19, the performance improvements of Regular-Schedule-based over BLAS-based implementation are higher than for the standard BLR.

We expect to have 5% to 10% more improvement in performance than reported here, if programs using Regular Schedule are coded in assembly.

7 Conclusion and Future Work

We have presented a new scalable algorithm, called the *Regular Schedule*, for parallel evaluation of general BLR's (i.e., mth-order BLR's for $m \geq 1$). The schedule's regular organization of the computation for BLR makes it extremely well suited for vector supercomputers. Moreover, the fact that Regular Schedule organizes the computation in *periods* of the same regular structure makes it a good choice for real-time applications (e.g., digital signal processing), whereas algorithms presented in [19, 6, 17, 15] do not have such desirable properties. We described our implementation of the Regular Schedules and discussed how to obtain maximum memory throughput in implementing the schedule using analytical and experimental methods on vector supercomputers (our experiments were performed on Convex C240 vector supercomputer). We also illustrated our approach, based on our Regular Schedule for BLR, to parallelizing programs containing BLR and other kinds of code. Our approach enables a comprehensive parallelization of programs containing core BLR and dependent code, and thus allows a better utilization of locality of reference in programs than separate treatment of BLR code and its dependent code. Significant improvements in CPU performance for a range of programs containing BLR implemented using the Regular Schedule in C over the same programs implemented using highly-optimized coded-in-assembly BLAS routines [11] are demonstrated on Convex C240. Our approach can be used both at user level in parallel programming code containing BLR's and in compiler parallelization of such code combined with recurrence recognition techniques.

The Regular Schedule (Algorithm 3.1) is scalable in terms of p, the number of processors. When mapped to vector supercomputers, it is also scalable in terms of both number of vector processors and vector register length. We have demonstrated this for a single vector processor in this paper. Note that implementations of Regular Schedule on a single vector processor does not require synchronization since it operates in SIMD mode. For multiple vector processors operating in SIMD mode, if r is the number of vector processors and s is the vector register length in a vector processor, then the time bound (as stated in Theorem 3.1) of the Regular Schedule holds with $p = rs$. Extending Regular Schedule to MIMD multiple vector processors will involve synchronization, an issue which we are currently investigating. Future work also includes adapting Regular Schedule onto massively parallel machines.

Since we have converted BLR to the problems of parallel prefix by rewritting them in terms of matrix chain multiplication, the numerical properties of Regular Schedule are critical. Recent results [32] on numerical properties of parallel algorithms for LR's provide evidence for the numerical stability of our Regular Schedule. The results in [32] states that the parallel schedule for LR's by Sameh and Brent[26] using unlimited resources is equivalent to the best sequential algorithm for LR's in numerical stability. The precise analysis of the numerical properties of Regular Schedule is beyond the scope of this paper and is left for future work.

Acknowledgement

The authors wish to thank Lyle Wiedeman and Don Frederik of UCI Office of Academic Computing for their technical help with Convex programming and to thank Dave Dodson, Scott Garee and Doug Scott of the Convex Computer Corporation for answering many of our questions about the Convex architecture and BLAS.

References

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

[2] Z. Ammarguellat and W. Harrison III, "Automatic recognition of induction variable and recurrence relations by abstract interpretation", *Proc. of ACM SIGPLAN 1990 Conf. on Programming Language Design and Implementation*, pp. 283-295, White Plains, New York, June 20-22, 1990.

[3] U. Banerjee, S. C. Chen, D. Kuck and R. Towle, "Time and parallel processor bounds for Fortran-like loops", *IEEE Trans. on Computers*, C-28(9), pp. 660-670, September 1979.

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

[5] D. Callahan, "Recognizing and parallelizing bounded recurrences", *Lecture notes in computer science—languages and compilers for parallel computing*, pp. 169-185, Springer-Verlag, 1992.

[6] S. C. Chen, D. Kuck and A. H. Sameh, "Practical Parallel Band Triangular System Solvers", ACM Transactions on Math. Software, Vol. 4, pp. 270-277, Sept, 1978.

[7] H. Conn and L. Podrazik, "Parallel recurrence solvers for vector and SIMD supercomputers", *Proc. 1992 International Conf. on Parallel Processing*, pp. 88-95, Vol. 3, Aug. 17-21, 1992.

[8] Convex Computer Corporation, *Convex architecture reference*, Richardson, Texas, 1991.

[9] Convex Computer Corporation, *Convex theory of operation*(C200 Series), Document No. 081-005030-000, 2nd Edition, Richardson, Texas, Sept. 1990.

[10] Convex Computer Corporation, *Convex SCILIB user's guide*, Document No. 710-013630-001, 1st Edition, Richardson, Texas, August 1991.

[11] Convex Computer Corporation, *Convex VECLIB user's guide*, Document No. 710-011030-001, 6th Edition, Richardson, Texas, August 1991.

[12] J. J. Dongarra, C. B. Moler, J. R. Bunch and G. W. Stewart, *Linpack Users' Guide*, Chapter 7, SIAM, Philadelphia, 1979.

[13] R. Eigenmann, J. Hoeflinger, G. Jaxon, Z. Li and D. Padua, "Restructuring Fortran programs for Cedar", In *Proc. of ICPP*, August, 1991.

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

[15] D. Gajski, "An Algorithm for Solving Linear Recurrence Systems on Parallel and Pipelined Machines", IEEE Transactions on Computers, Vol. c-30, No.3, March 1981.

[16] K. A. Gallivan, R. J. Plemmons, A. H. Sameh, "Parallel Algorithms for Dense Linear Algebra Computations", SIAM Review, Vol. 32, No. 1, pp. 54-135, March 1990.

[17] L. Hyafil and H. T. Kung, "The Complexity of Parallel Evaluation of Linear Recurrence", JACM, Vol. 24, No.3, pp. 513-521, July 1977

[18] R. M. Karp, R. E. Miller, and S. Winograd, "The organization of computations for uniform recurrence equations". Journal of the ACM, 14:563-590, 1967.

[19] 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.

[20] D. Kuck, *The structure of computers and computations*, pp. 118-119, Vol. 1, John Wiley & Sons, 1978.

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

[22] F. T. Leighton, *Introduction to Parallel Algorithms and Architectures: Arrays-Trees-Hypercubes*, Section 1.3, Morgan Kaufmann Publishers, Inc., San Mateo, California, 1992.

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

[24] V.P. Roychowdhury, "Derivation, Extensions and Parallel Implementation of Regular Iterative Algorithms", Ph.D. Dissertation, Stanford University, December 1988.

[25] S. Pinter and R. Pinter, "Program optimization and parallelization using idioms", *In conference record of the 18th ACM Symposium on the Principles of Programming Languages*, Jan. 1991.

[26] A. Sameh, R. Brent, "Solving triangular systems on a parallel computer", SIAM J. Num. Anal. 14(1977), 1101-1113.

[27] Shang, W. and Fortes, J.A.B., "On time mapping of uniform dependence algorithms into lower dimensional processor arrays", IEEE Transactions on Parallel and Distributed Systems, May 1992, vol.3, (no.3):350-63.

[28] Shang, W. and Fortes, J.A.B. "Independent partitioning of algorithms with uniform dependencies", IEEE Transactions on Computers, Feb. 1992, vol.41, (no.2):190-206.

[29] Shang, W. and Fortes, J.A.B. "Time optimal linear schedules for algorithms with uniform dependencies", IEEE Transactions on Computers, June 1991, vol.40, (no.6):723-42.

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

[31] Y. Tanaka, "Compiling techniques for first-order linear recurrence", *The journal of supercomputing*, 4(1):63-82, March 1990.

[32] N. K. Tsao, "Solving triangular system in parallel is accurate", pp. 633-638, *Numerical Linear Algebra, Digital Signal Processing and Parallel Algorithms*, Edited by Gene Golub and Paul Van Dooren, NATO Series F: Computer and Systems Sciences, Vol. 70, Springer-Verlag, 1991.

[33] H. Wang, A. Nicolau, "Speedup of Band Linear Recurrences in the Presence of Resource Constraints", *Proc. 1992 6th ACM International Conf. Supercomputing*, pp. 466-477, Washington, D. C., July 19-23, 1992.

[34] H. Wang, A. Nicolau, "Computing programs containing band linear recurrences on vector supercomputers", TR 92-113, Department of Computer Science, UC Irvine, December 1992.