Parallel Computing 7 (1988) 163–185
North-Holland

An analysis of the computational and parallel complexity of the Livermore Loops

John T. FEO

Computing Research Group, Lawrence Livermore National Laboratory, Livermore, CA 94550, U.S.A.

Received December 1986
Revised April 1987

Abstract. This paper presents and analyzes the computational and parallel complexity of the Livermore Loops. The Loops represent the type of computational kernels typically found in large-scale scientific computing and have been used to benchmark computer system since the mid-60's. On parallel systems, a process's computational structure can greatly affect its efficiency. If the Loops are to be used to benchmark such systems, their computations must be understood thoroughly, so that efficient implementations may be written. This paper addresses that concern.

Keywords. Parallel processing, complexity analysis, Livermore Loops, benchmarking parallel systems, efficient implementation.

1. Introduction

The Livermore Loops [6] are 24 FORTRAN loops from actual production codes run at Lawrence Livermore National Laboratory. The loops represent the type of computation kernels typically found in large-scale scientific computing. They range from common mathematical operations, such as inner product and matrix multiplication, to intricate searching and storing algorithms, such as the Monte Carlo Search Loop and the 2-D Particle-in-Cell Loop.

The Loops have been used to evaluate the performance of computer systems since the mid-60's [7]. Most large-scale, scientific computer systems have run the Loops; their execution rates are typically given in terms of the millions of floating-point operations per second (Mflops) achieved on the Loops [8]. As computer systems have evolved from strictly sequential machines to vector processes, and now to parallel systems consisting of tens or even hundreds of processors, the importance of the Loops as benchmarks has not diminished; however, the meaning of their execution rates and the significance of the variation in speed among the loops has changed.

When computers were classical Von Neumann machines, benchmarking was equivalent to evaluating the speed of hardware components. But with the advent of vector processors, this is no longer the case. The execution speed of a vector machine depends greatly on the computational nature of the processes it executes. If these processes are highly vectorized, execution rates will be high; if the processes are essentially scalar, execution rates will be low. In benchmarking a vector machine it is essential to execute a set of procedures with differing amounts of vectorization in order to determine the machine's speed over a broad range of

process types. Since the Loops contain code fragments that range from strictly sequential to 100% vectorizable, they are an appropriate means to measure performance across this spectrum.

The Loops can also be used to evaluate vector compilers. These compilers analyze the interdependency of array instructions, particularly in loops, to find vector operations and then they rewrite these operations as vector instructions. (Greater the number of vector operations found by a compiler, greater is the speed of its implementation of the procedure.) By comparing the speed of a compiler's implementation of the Loops on a particular vector machine to the optimal vector implementation on that machine, one can measure the compiler's adaptness a finding vector operations [2].

Now that parallel computing systems are a reality, the implications of benchmarking have again changed. The resources of a parallel system are interconnected by a communication network over which they exchange data and synchronization information. The similarity between the geometry of the communication network and a procedure's computational structure greatly affects the execution speed of the procedure. In benchmarking a parallel system it is important to execute a set of processes with diverse computational structures. As there was a spectrum of vectorization to consider in vector machines, there is a diversity of computational geometries to consider in parallel machines. The Loops embody a wide range of computational structures, including sequential processes, grids, pipelines, and wavefronts; therefore, they make an excellent collection of kernels on which to evaluate the performance of parallel architectures.

Parallel processing has many more degrees of freedom than either sequential or vector processing. This makes evaluating parallel systems much more difficult. The execution speed of a parallel process is very sensitive to the geometry of the underlying interconnection network. One type of parallel process running quickly on a particular system gives little information about how quickly another parallel process will run. In fact, even two similar processes, such as two grid algorithms, may run at different speeds due to their specific communication characteristics. Thus, while running the Loops gives an indication of how fast a particular parallel system executes, great care must be taken when generalizing these results to an arbitrary workload.

As the computational properties of the Loops become more important in determining how efficiently they will execute on a system, it is paramount that these properties be well understood. It is exceedingly unlikely that compilers for parallel systems will be able to save the user from poorly written code. The difference between efficient and inefficient implementations of an algorithm on such systems may require more than the rearrangement of a few lines of code; fundamentally different approaches are often necessary. If a system designer were to benchmark his machine using an inappropriate implementation of the Loops, he could drastically underestimate the computational power of this machine. This paper investigates the properties and explains the computations and mathematics of the Loops. We hope that the analysis presented in this paper will allow users to write clean and efficient implementations of the Loops for their systems.

The complexity of each loop is discussed in the second section. Complexity may be defined in terms of execution time, number of computations, cost of processing, words of memory, memory accesses, and so on. Traditionally, an algorithm's complexity has been defined as the number of computations required to complete the procedure as a function of the input size [1]. On parallel systems, where many instructions can be executed simultaneously, it is more meaningful to talk about complexity in terms of execution time, or logical time steps. Since the Loops have been used historically to determine the speed of computer systems, it is particularly appropriate to define their complexity in terms of execution time.

An algorithm's parallel execution trace is two-dimensional: its height is the number of time steps required for it to complete, and the width of each step is the number of operations that can be computed concurrently at that time. The height of an algorithm, herein, is defined as its

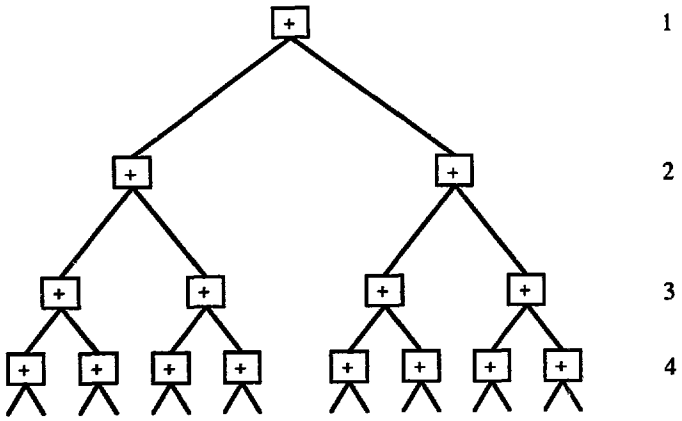

Fig. 1. A binary tree computation.

order. Thus, an algorithm is $O(n)$ if it completes in n logical steps, where a step includes all operations that can be executed at a particular instant of time. For example, the binary computation tree shown in Fig. 1 has a height of 4 and the width of each step is 8, 4, 2, and 1 from bottom to top. Notice that the tree's order is equivalent to $\log_2 n$, where n is the number of inputs.

Usually, an algorithm's height and width are inversely related—decreasing one increases the other. In redesigning algorithms it is important that the height does not decrease at a slower rate than the width expands. While the former is an indication of the time it takes a process to complete on a parallel machine, the latter is an indication of the number of resources required to execute the algorithm. There is little to be gained by reducing an algorithm's execution time from n^2 to n if the number of resources required increases from n to n^3. There are few computing environment in which it is cheaper for a process to run n times faster but consume n^2 more resources.

2. The Livermore Loops [1]

2.1. Loop 1

Loop 1 is a fragment from a hydrodynamic code. The value of $X(k)$, $k = \{1,\ldots,n\}$, is set to

$$Q + [Y(k) * [R * Z(k+10) + T * Z(k+11)]].$$

Since X does not appear in the equation, there are no data dependencies. Consequently, all n values can be computed in parallel and the loop's complexity is O(1).

From a parallel standpoint, the loop is straightforward and not very interesting. From a vector standpoint, it is very interesting for it includes: scalar-vector operations, vector-vector operations and chaining. Chaining is the hardware capability of sending the output of one vector functional unit to the input of another functional unit element-by-element, so that the two vector units run simultaneously. Optimally, one would compute

$$R * Z[11,\ldots,n+10] \quad \text{and} \quad T * Z[12,\ldots,n+11]$$

[1] For the FORTRAN programs of the 24 loops, see Appendix A.

(both scalar-vector multiplies) in parallel, and send the corresponding output pairs to an adder (a vector-vector add). The output vector from the ader would be multiplied by Y (a vector-vector multiply), and as the results emerged from the multiply unit, they would be incremented by Q (a scalar-vector add) and stored in X. Loop 1 is therefore a ideal loop to test the efficiency of vector operations and their chaining.

2.2. Loop 2

Loop 2 is a fragment from an Incomplete Cholesky-Conjugate Gradient code. The X array is divided into a left- and a right-hand side (lhs and rhs, respectively). The lhs consists of the first n elements of the array, and the rhs consists of the remaining elements. The elements of the lhs are used to compute new values for the first $\frac{1}{2}n$ elements of the rhs. The rhs is then divided into two parts: a lhs consisting of its first $\frac{1}{2}n$ elements (i.e., the elements changed on the first iteration) and a rhs consisting of the remaining elements. The process is repeated until the rhs is reduced to a single element. IL is the length of the lhs; $IPNT$ and $IPNTP$ are the indices of the first and last element of the lhs, respectively; and $IPNTP + 1$ is the index of the first element of the rhs (Fig. 2).

Since the elements changed on the ith iteration become the lhs of the $(i + 1)$st iteration, the iterations must be performed sequentially. However, since the new rhs values are functions only of lhs values (with possibly one exception), the calculations at each iteration may be performed in parallel. The exception occurs whenever the length of the lhs is even. For then on the last iteration of the DO-loop, $k = IPNTP$ and the equation for $X(i)$ includes $X(IPNTP + 1)$, the first element of the rhs. This data dependency forces the first and last computation of the iteration to be executed sequentially. Since the size of the rhs is halved each time, there are $\log n$ iterations, where n is the length of the first lhs. Since each iteration can be computed in at most two steps and in constant time, the complexity of the loop is $O(\log n)$.

The inner loop can be vectorized if the last value calculated on each iteration is computed separately. Only two memory reads are necessary to implement the loop: one to load the first n elements of X, and one to load V. Since the values calculated on the ith iteration become the lhs of the $(i + 1)$st iteration, there is no need to read from memory every iteration.

2.3. Loop 3

Loop 3 is the standard Inner Product function of linear algebra. Because it appears frequently in scientific codes, an efficient implementation is important. On a vector machine, the two arrays can be sent pairwise to a multiply unit and the resulting products summed in an accumulator. On a parallel machine, the products can be calculated in parallel and summed in a binary fashion. The complexity of the sum and, thus, the complexity of the function is $O(\log n)$, where n is the size of the arrays.

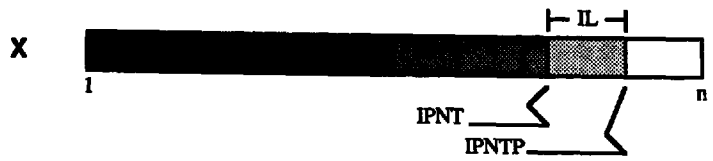

Fig. 2. The data structures of loop 2.

2.4. Loop 4

Loop 4 is a fragment from a Banded Linear Equations routine. The values of $X(k-1)$, $k = \{7, 504, 1001\}$, is set to

$$Y(5) * \left[X(k-1) - \sum_{i=1}^{[n/5]} [X(k-7+i) * Y(5*i)] \right].$$

Since both X and Y are dimensioned 1001, n must be $\leqslant 35$. Thus, the X elements used in the three calculations are disjoint and the three values of X can be computed concurrently. The $X(k-1)$ term appearing explictly in the equation refers to the old value and is of no consequence. However, a $X(k-1)$ term in the sum refers to the new value and creates a data dependency. The term appears in the sum if $n > 25$. In such cases, the summation must be stopped after the fifth step and the accumulated sum substituted for $X(k-1)$. In either case, the loop is logically an inner product of X and Y, and is O(log n).

2.5. Loop 5

Loop 5 is taken from a Tridiagonal Elimination routine. The value of $X(i)$, $i = \{2, \ldots, n\}$, is set to

$$Z(i) * [Y(i) - X(i-1)].$$

The $X(i-1)$ term in the equation creates a data dependency across iterations forcing a sequential execution. The loop as written is O(n).

However, the dependency can be eliminated by recursively substituting for X in the equation. For example, the expression for $X(5)$ can be written as

$$X_5 = Z_5Y_5 - Z_5Z_4Y_4 + Z_5Z_4Z_3Y_3 - Z_5Z_4Z_3Z_2Y_2 + Z_5Z_4Z_3Z_2X_1.$$

Since only $X(1)$ appears in the equations, there are no data dependencies and all new values of X can be computed simultaneously. However, calculating each unrolled equation individually is very expensive requiring O(n^3) multiplications and O(n^2) additions. Many of these operations are redundant and it is possible to solve the equations in a structured manner so that there is no wasted effort. The method is based on a concept known as recursive doubling and is described in [3] and [4].

Figure 3 is the computation graph for $X(5)$ using this method. The products of Z's are computed simultaneously in a similar manner. The computation trees for the X's can be overlayed resulting in the parallel computation depicted in Fig. 4. The implementation requires n processors and generates O(n log n) multiplications and additions. Its complexity is obviously O(log n). The method is certainly less expensive than executing the unrolled equations individually. Whether or not it is cheaper than the sequential implementation, which is O(n) but uses only 1 processor, is likely to be system dependent.

2.6. Loop 6

Loop 6 is an example of a general linear recurrence equation. The value of $W(i)$, $i = \{2, \ldots, n\}$, is set to

$$W(i) + \sum_{k=1}^{i-1} [B(i, k) * W(i-k)].$$

Since the sum includes $W(i-1)$, the outer loop over i must be executed sequentially. However, the inner loop (the sum) is an inner product of known values and is O(log i). Since i is of O(n) and there are n executions of the inner loop, the function is O(n log n).

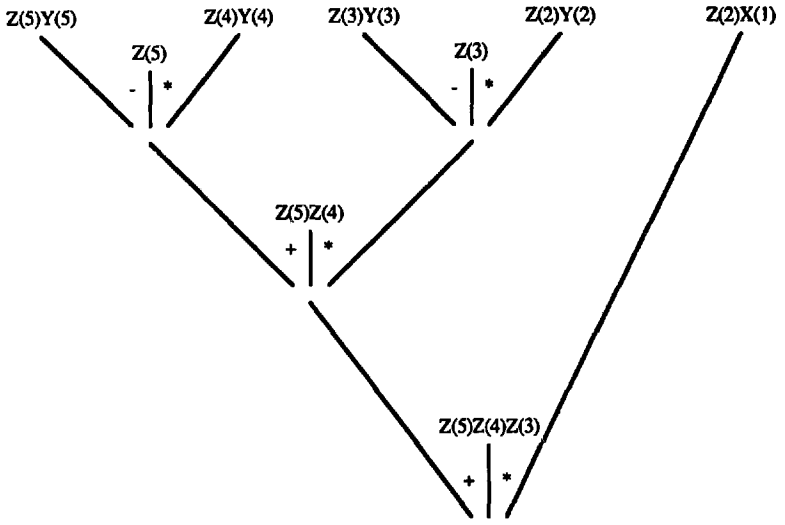

Fig. 3. Parallel computation graph for $X(5)$.

The loop as written is not optimal; it can be implemented in $O(n)$. Notice that $W(i)$ appears in the sum of all elements whose indices are greater than its own; i.e., it contributes to the sum of $W(i+1)$, $W(i+2),\ldots,W(n)$. Therefore at iteration i, instead of summing elements $W(1)$ through $W(i-1)$ to calculate $W(i)$, add $W(i-1)$ multiplied by the appropriate elements of $\boldsymbol{B}$ to $W(i)$ through $W(n)$. Since the additions can be done in parallel, the inner loop is now O(1) and the function's overall complexity is $O(n)$. The FORTRAN version of the $O(n)$ algorithm is given in Fig. 5. The change is more than a simple loop interchange, since both the loop bounds and the array indices have changed.

2.7. Loop 7

Loop 7 is an Equation of State fragment. Since $\boldsymbol{X}$ does not appear on the equation's rhs, the iterations are data independent and the function's logical complexity is O(1). The loop is an important benchmark for it has the highest ratio of floating-point operations to memory accesses compared to the other loops. Consequently, it should execute fastest. Since the loop's equation consists of 8 scalar-vector multiplications and 8 vector-vector additions, it is an

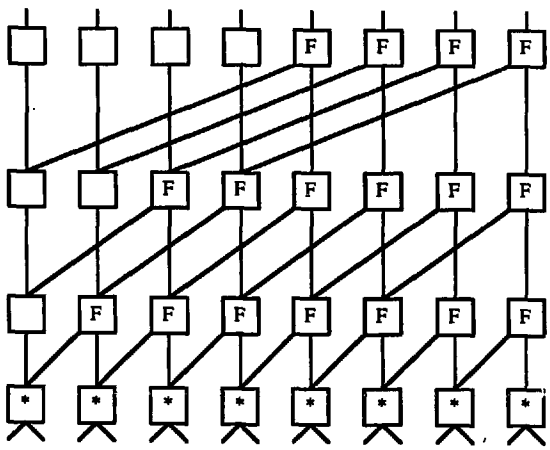

Fig. 4. Parallel computation graph for Loop 5.

```
      subroutine Loop6(n,B,W)
      integer i,k,n
      double precision B(64,64),W(1001)
      Do 6 k = 1,n-1
      Do 6 i = k+1,n
    6    W(i) = W(i) + (B(i,i-k) * W(k))
      return
      end
```

Fig. 5. O(n) implementation of Loop 6.

excellent test of a vector machine's ability to execute and chain such vector operations. From a parallel standpoint, the equation can be broken into a number of subexpressions which can be computed in parallel and summed. Thus, the loop is an excellent test of how well a parallel system finds and executes the fine-grain parallelism within computational tasks.

2.8. Loop 8

Loop 8 is part of an Alternating Direction, Implicit Integration code. All terms on the right-hand sides of the equations are either constants or functions of elements in the first planes of the U matrices (third index equal to 1). Since only elements in the second planes of the U matrices (third index equal to 2) are changed, all rhs values are known at the start of the loop. Thus, the iterations can be executed concurrently and the function's complexity is O(1).

The body of the loop consists of six equations. The last three are each a sum of three terms. For example, the equation for $U1(kx, ky, 2)$ is

$$\begin{aligned} &U1(kx, ky, 1) \\ &\quad + A11 * DU1(ky) + A12 * DU2(ky) + A13 * DU3(ky) \\ &\quad + \mathrm{SIG} * [U1(kx-1, ky, 1) - 2.0 * U1(kx, ky, 1) + U1(kx+1, ky, 1)]. \end{aligned}$$

The loop is easily vectorized over ky. ***DU1***, ***DU2***, and ***DU3*** are computed (each a vector-vector subtraction), multiplied by the appropriate scalar, and added together. If multiple functional units are available, the first term should be included in the add to even out the number addends. There is no need to store ***DU1***, ***DU2***, and ***DU3*** after the first iteration, since they are recomputed on the second iteration.

The equation's third term is a Laplacian. The middle addend can be unrolled and the sum written as

$$[[U1(kx-1, ky, 1) - U1(kx, ky, 1)] - [U1(kx, ky, 1) - U1(kx+1, ky, 1)]].$$

On a vector machine with multiple functional units the unrolled version should be faster. The two inner expressions can be computed simultaneously and the resultant vectors chained to a third functional unit. If only one functional unit is available, executing the rolled up expression may be faster since it consists of a shift operation (the multiplication by 2.0) and two subtractions, as opposed to three subtractions.

2.9. Loop 9

Loop 9 is an Integrate Predictor code. The first n values of ***PX***'s first row are recomputed. Since the equation is independent of the first row, there are no data dependencies across the iterations and all can be computed in parallel. The loop is therefore O(1). The loop trivially vectorizes over i. On a parallel machine, the equation should be broken into its natural components, and the components computed in parallel and summed. As with Loop 7, this loop

is an excellent test of a parallel system's ability to find and to execute the fine-grain parallelism buried in complex equations.

2.10. Loop 10

Loop 10 is a Difference Predictor code. The value of $PX(i, j)$, $i = \{5,\ldots,14\}$, $j = \{1,\ldots,n\}$, is set to $CX(5, j)$ minus the sum of the ***PX*** elements in the rows $i-1$ through 5 of column j. Mathematically,

$$PX(i, j) = \begin{cases} CX(5, j), & i = 5, \\ CX(5, j) - \sum_{k=5}^{i-1} PX(i, k), & i > 5. \end{cases}$$

The ***PX*** elements in the sum are the original elements, not the new ones computed by the loop. Because all terms on the rhs are known at the start of the loop, each iteration can be executed simultaneously. Thus, the loop's complexity is that of the summation, or O(log i). Notice the order is independent of the loop variable n; increasing or decreasing n (the column size of the arrays) should not affect the execution time of the loop, assuming a sufficient number of resources are available. Since i is a constant in the official release, the loop's order is constant.

On a parallel machine with $10n$ processors, each computation can be executed as an independent process. However, if the number of processors is close to n, it may be more efficient to implement the loop as n individual processes each computing a different row. The loop would them still execute in constant time proportional to i. The loop can be vectorized over i by replacing the scalar variables, AR, BR and CR, with array variables, a technique known as scalar expansion [5]. Figure 6 gives a vector version of the algorithm written in FORTRAN 8X-like instructions—the next generation Fortran, which includes many array features and vector instructions. The two vector variables, $V1$ and $V2$, are the vestiges of the scalars, AR, BR, and CR. Most vectorizing compilers fail to vectorize this loop.

```
subroutine Loop10(n,CX,PX)
double precision CX(25,101),PX(25,101)
double precision V1(101),V2(101)
V1(1:n) = CX(5,1:n) - PX(5,1:n)
PX(5,1:n) = CX(5,1:n)
V2(1:n) = V1(1:n) - PX(6,1:n)
PX(6,1:n) = V1(1:n)
V1(1:n) = V2(1:n) - PX(7,1:n)
PX(7,1:n) = V2(1:n)
V2(1:n) = V1(1:n) - PX(8,1:n)
PX(8,1:n) = V1(1:n)
V1(1:n) = V2(1:n) - PX(9,1:n)
PX(9,1:n) = V2(1:n)
V2(1:n) = V1(1:n) - PX(10,1:n)
PX(10,1:n) = V1(1:n)
V1(1:n) = V2(1:n) - PX(11,1:n)
PX(11,1:n) = V2(1:n)
V2(1:n) = V1(1:n) - PX(12,1:n)
PX(12,1:n) = V1(1:n)
PX(14,1:n) = V2(1:n) - PX(13,1:n)
PX(13,1:n) = V2(1:n)
return
end
```

Fig. 6. Loop 10 expressed in vector instructions.

2.11. Loop 11

Loop 11 is a First Sum. The value of $X(k)$, $k = \{1,\ldots,n\}$, is set to the sum of $Y(i)$, $i = \{1,\ldots,k\}$. Obviously, the sums are data independent and can be computed in parallel. The loop's complexity is O(log n). However, calculating the n sums individually is wasteful since both $X(i)$ and $X(i+1)$ calculate the sum of $Y(j)$, $j = \{1,\ldots,i\}$. An efficient parallel implementation is possible based on the method of recursive doubling described in [3]. The procedure avoids any redundant work by adding $Y(i)$ and $Y(i+1)$, $1 \leqslant i \leqslant n-1$, at the start and then, combining these sums in an ordered fashion to generate all n sums in log n steps. The algorithm is depicted in Fig. 7. As in Loop 5, whether or not such a parallel implementation is less expensive than the sequential implementation, which executes in n steps but uses only 1 processor, will be system dependent.

2.12. Loop 12

Loop 12 is a First Difference. The value of $X(k)$, $k = \{1,\ldots,n\}$, is set to $Y(k+1) - Y(k)$. The equation is independent of X and so, all n values can be calculated simultaneously. The loop's complexity is O(1). Both the vector and parallel implementations are trivial.

2.13. Loop 13

Loop 13 is fragment from a 2-D Particle-in-Cell code. The loop is interesting because some array indices are indirect—a function of P. On the ith iteration, the value of the ith column of P and an element of H are changed. The changes to $P(1, i),\ldots, P(4, i)$ are a function of B, C, Y, and Z. The indices of the elements used from these arrays, as well as the indices of the elements of E and F used in calculating the index of the H element to be changed are functions of $P(1, i)$ and $P(2, i)$. In fact, the index function for H includes $P(1, i)$ and $P(2, i)$ explicitly.

All but the last calculation of the loop can be both parallelized and vectorized (the scalars can be eliminated by substitution). Since the value of P is unknown at compile time, it is not known a priori how many times a particular element of H will be incremented; therefore, the n instances of the last computation must be executed sequentially. Consequently, the function is O(n). If it is known that an element of H can be incremented at most once, then each iteration in its entirety could execute concurrently and the function's complexity would be O(1).

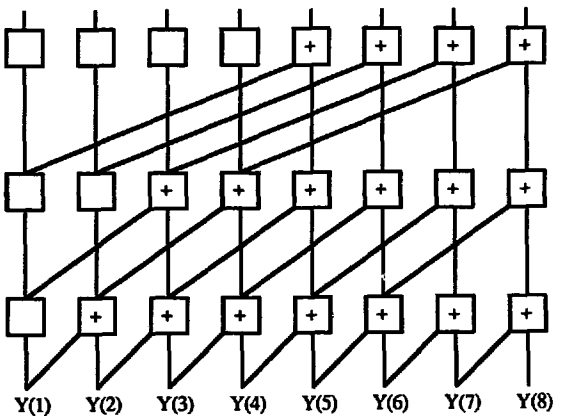

Fig. 7. Parallel computation graph for Loop 11.

2.14. Loop 14

Loop 14 is a part of a 1-D Particle-in-Cell code. The first two DO-loops can be combined into a single loop because the kth iteration of both loops involves only the kth element of the different arrays. Moreover, the iterations of the loop can be computed in parallel. The last DO-loop augments elements of ***RH*** indexed indirectly trhough ***IR***. Since the value of ***IR*** is unknown at a compile time, the third loop must be executed sequentially. This makes the function's complexity O(n). However, as in the case of Loop 13, if it is known that each element of ***RH*** is augmented at most once, the third loop can be combined with the first two. The iterations of this super loop would be data independent and could execute in parallel. The function's complexity would then be O(1).

2.15. Loop 15

Loop 15 is a prime example of how casually FORTRAN can be written. A compiler attempting to optimize, vectorize, or parallelize this loop would have to spend considerable effort just understanding what it does. Essentially, the loop has two parts: the top half recomputes the subarray $VY(2,\ldots,n,2,\ldots,7)$; and the bottom half recomputes the subarray $VS(2,\ldots,n,2,\ldots,6)$. The affected elements of ***VY*** are set to the expression on Line 20, except for those in the 7th column which are set to 0.0 (the effect of lines 7 and 8); and the affected elements of ***VS*** are set to the expression on Line 32, except for those in the nth row which are set to 0.0 (the effect of lines 21 and 22). The other statements of the function determine the values of the scalars R, S, and T used in the two expressions. A clean FORTRAN version is given in Fig. 8. Since the ith iteration uses no value computed on an earlier iteration, there are no data dependencies across iterations. Thus, the loop's complexity is O(1).

It is not too difficult to vectorize the clean version of the loop, although some compilers may fail to do so. The trick is to pull the statements for R, S, and T outside the loop and place them in a loop of their own. The new loop calculates all the values of R, S, and T used in the two equations and creates a vector for each. Once R, S, and T are vectors, the equations for ***VY*** and ***VS*** vectorize easily.

2.16. Loop 16

Loop 16 is the search loop from a Monte Carlo code. The use of multiple arithmetic-ifs makes the FORTRAN look unnecessarily fiendish. The loop searches a grid of arbitrary geometry for the location of a particular particle. The grid is divided into groups of n zones. The number of groups is stored in $ZONE(1)$ and each zone is described by two parameters stored in successive elements of ***ZONE***. The parameters of the first zone are in $ZONE(2)$ and (3), those of the second zone are in $ZONE(4)$ and (5), and so on. The zones are searched sequentially for the particle. On exit, m and $j4$ hold the identificaiton number of the group and the zone in which the particle was found. If the particle is not found, m is set to 1 before exiting (line 24).

The k loop searches over the zones of a group, and lines 22–25 control the search over the groups. The interior of the k loop consists of three parts. The first segment (lines 12–14) decides which boolean expression is used in the second segment (lines 15–17). The zone's second parameter, $ZONE(j4)$, is compared to n. If the parameter is less than $\frac{1}{3}n$, less than $\frac{2}{3}n$, or less than n, then $PLAN(ZONE(j4))$ is compared to T, S, or R, respectively, in the second segment. If it is greater than n, the boolean expression starting on line 20 is used; and, if it is equal to n, the loop terminates. The second segment either terminates the loop (whenever the boolean's lhs is 0.0) or chooses the positive or the negative value of the zone's first parameter,

```
      subroutine Loop15(n,VF,VG,VH,VS,VY)                                1
          integer j,k                                                    2
          double precision R,S,T,VF(101,7),VG(101,7),                    3
         .    VH(101,7),VS(101.7),VY(101,7)                             4
          Do 150 k = 2,n                                                 5
      150    VY(k,7) = 0.0d0                                             6
          Do 151 k = 2,6                                                 7
      151    VS(n,k) = 0.0d0                                             8
          Do 152 j = 2,6                                                 9
          Do 152 k = 2,n                                                10
             T = 0.073d0                                                11
             if(VH(k,j+1) .gt. VH(k,j)) T = 0.053d0                     12
             if(VF(k,j) .ge. VF(k-1,j)) then                            13
                R = max(VH(k-1,j), VH(k,j+1))                           14
                S = VF(k,j)                                             15
             else                                                       16
                R = max(VH(k-1,j), VH(k-1,j+1))                         17
                S = VF(k-1,j)                                           18
             endif                                                      19
      152    VY(k,j) = dsqrt(VG(k,j)**2 + R * R) * T / S                20
          Do 153 j = 2,6                                                21
          Do 153 k = 2,n-1                                              22
             if(VF(k,j) .ge. VF(k+1,j-1)) then                          23
                R = max(VG(k,j), VG(k+1,j))                             24
                S = VF(k,j)                                             25
                T = 0.053d0                                             26
             else                                                       27
                R = max(VG(k,j-1), VG(k+1,j-1))                         28
                S = VF(k,j-1)                                           29
                T = 0.073d0                                             30
             endif                                                      31
      153    VY(k,j) = dsqrt(VG(k,j)**2 + R * R) * T / S                32
          return                                                        33
          end                                                           34
```

Fig. 8. A clean FORTRAN version of Loop 15.

ZONE($j4-1$). If the loop does not terminate, the value chosen is compared to 0.0 in the third segment (lines 18–19). If the value is negative, the loop jumps to the next group of zones; if it is 0.0, the loop terminates; and if it is positive, the loop moves to the next zone within the same group. A clean FORTRAN version of the loop is given in Fig. 9.

The loop can not be vectorized, although some use can be made of vector registers and instructions. However, the loop can be parallelized since each zone can be searched simultaneously. The zone which finds the particle can return a 1 and all others can return a 0. Then m and $j4$ can be set appropriately. Even if more than one zone should be return a 1 (multiple identical particles), the sequential and parallel versions can be made equivalent by setting m and $j4$ to the lowest numbered zone returning a 1. Since the zones can be searched in parallel, the loop's complexity is O(1).

2.17. *Loop 17*

Loop 17 is an example of an implicit conditional computation. The loop calculates new values for elements 2 through n of ***VE3***, ***VXND***, and ***VXNE***. The function has three parts: a selector (lines 15–20), a step model (lines 9–14), and a linear model (lines 21–26). On iteration i, the selector chooses which of the two models to use to compute the $(i+1)$st value of the three arrays. If either *XNM* or *VXNE*(i) is greater than $\frac{5}{3}$*XNC*, the step model is used; else, the linear model is used. Although it is not obvious, the loop's iterations are data dependent.

```
      subroutine Loop16(n,m,j4,R,S,T,D,PLAN,ZONE)                          1
      integer m,j2,j4,j5,C1,ZONE(300)                                      2
      double precision C,R,S,T,D(300),PLAN(300)                            3
      Do 163 m = 0,ZONE(1)-1                                               4
       j2 = (2 * n * m) + 1                                                5
       Do 162 j4 = j2+2, j2+(2*n),2                                        6
          j5 = ZONE(j4)                                                    7
          if(j5 .lt. n/3) then                                             8
             C = T                                                         9
          else if(j5 .lt. 2*n/3) then                                     10
             C = S                                                        11
          else if(j5 .lt. n) then                                         12
             C = R                                                        13
          else if(j5 .eq. n) then                                         14
             goto 164                                                     15
          else                                                            16
             goto 160                                                     17
          endif                                                           18
          if(PLAN(j5) .lt. C) C1 = ZONE(j4-1)                             19
          if(PLAN(j5) .eq. C) goto 164                                    20
          if(PLAN(j5) .gt. C) C1 = - ZONE(j4-1)                           21
          goto 161                                                        22
160       C1 = - ZONE(j4-1)                                               23
          if(D(j5) - (D(j5-1) * (T-D(j5-2))**2 +                          24
            (S-D(j5-3))**2 + (R-D(j5-4))**2) .lt. 0.0) C1 = ZONE(j4-1)    25
161       if(C1 .lt. 0) goto 163                                          26
          if(C1 .eq. 0) goto 164                                          27
          if(C1 .gt. 0) goto 162                                          28
162       continue                                                        29
163   continue                                                            30
      m = 1                                                               31
164   return                                                              32
      end                                                                 33
```

Fig. 9. A clean FORTRAN version of Loop 16.

Notice the expressions for ***VE3***, ***VXND***, and ***VXNE*** are all functions of *XNM*; and that the expression for *XNM* is recurrent. In the step model, the expression is

$$XNM = [XNM * VSP(i)] + VSTP(i);$$

and in the linear model, it is

$$XNM = 2.0 * [[XNM * VLR(i)] + VSTP(i)] - XNM.$$

Since the loop's iterations must be executed sequentially, the function is O(n).

2.18. Loop 18

Loop 18 is a fragment from a 2-D Explicit Hydrodynamic code. It consists of three parts: the first recomputes the subarrays $ZA(2,\ldots,n,2,\ldots,6)$ and $ZB(2,\ldots,n,2,\ldots,6)$; the second recomputes the subarrays $ZU(2,\ldots,n,2,\ldots,6)$ and $ZV(2,\ldots,n,2,\ldots,6)$; and, the third recomputes the subarrays $ZR(2,\ldots,n,2,\ldots,6)$ and $ZZ(2,\ldots,n,2,\ldots,6)$. The three sections must be executed sequentially, since the expressions for ***ZU*** and ***ZV*** are functions of ***ZA*** and ***ZB***, and the expressions for ***ZR*** and ***ZZ*** are functions of ***ZU*** and ***ZV***. However, the iterations in each individual section are data independent and can be executed concurrently. The function is logically a three step procedure in which each step is O(1); therefore, the entire loop is O(1).

The loop is especially interesting because it is a coarse-grain pipeline procedure. Its sections logically form a three-stage pipeline. Machines that can pipe or stream the results of one

subprocedure to another should perform extremely well on this loop. The computations contain many vector operations and there is ample opportunity for vector chaining. On a parallel system, the three double DO-loops must be inverted to execute across the rows instead of down the columns to obtain optimal efficiency. If this is done, the second loop can execute two steps behind the first, i.e., iteration $(j, k-2)$ of the second loop can execute concurrently with iteration (j, k) of the first loop; and the third loop can execute 7 steps behind the second, i.e., iteration $(j-1, k-1)$ of the third loop can execute concurrently with iteration (j, k) of the second loop.

2.19. Loop 19

Loop 19 is a general Linear Recurrence Equation. It consists of two subloops: the first computes $\boldsymbol{B5}$ in a forward sweep from 1 to n, and the second recomputes $\boldsymbol{B5}$ in a reverse sweep from n to 1. Since the second loop recomputes the entire array, the computation for $\boldsymbol{B5}$ in the first loop may be deleted as long as the rhs of the expression on Line 5 is substituted for the instance of $\boldsymbol{B5}$ on Line 6. Both subloops are recurrent and both can be solved using recursive doubling. Since only the last computation of the first loop is used, it is not necessary to compute the entire computation graph, but only those operations in the subtree rooted at the last value (Fig. 10). The loop's complexity is O(log n). Loop 19 is a pipeline procedure in the sense that the first subloop feeds the second; but unlike Loop 18, there is no concurrency since only the last result of the first subloop is used in the second subloop.

2.20. Loop 20

Loop 20 is a fragment from a Discrete Ordinates Transport program. Both the expressions for $X(k)$ and $XX(k+1)$ are functions of $XX(k)$. Since $XX(k)$ and $XX(k+1)$ are computed on successive iterations, the iterations must be executed in strict sequential order. Thus, the loop is O(n). The loop can neither be vectorized or parallelized. It is even impossible to build a vector mask for the conditional expression on Line 8 since it is a function of DI, which is a function of $XX(k)$.

2.21. Loop 21

Loop 21 is a simple matrix calculation. It computes the vector equation $\boldsymbol{P} + (\boldsymbol{VY} * \boldsymbol{CX})$. If the dimension of $\boldsymbol{VY}$ are (i, k) and the dimensions of $\boldsymbol{CX}$ are (k, j), then their product consists of $i+j$ inner products, all of which can be computed in parallel. The complexity of the

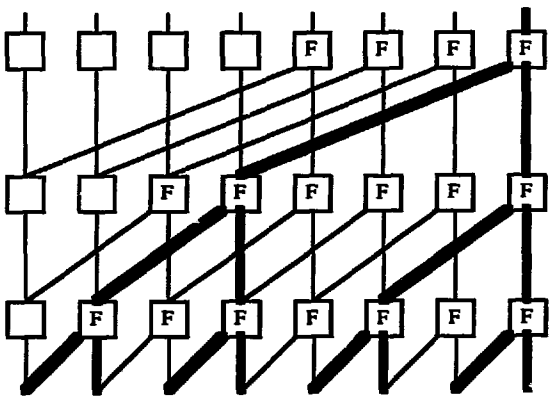

Fig. 10. Computation graph for the first subloop of Loop 19.

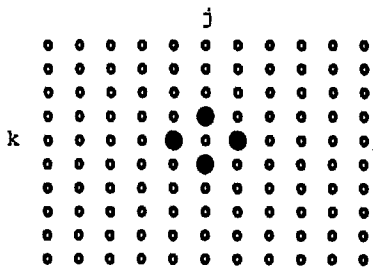

Fig. 11. Elements of ZA required to calculate $ZA(k, j)$.

multiplication, and therefore of the loop, is equivalent to that of the inner product of two k-element vectors, or O(log k). As with Loop 10, the complexity is independent of the loop variable n (the row and column size of VY and CX, respectively), and so, the loop should execute in constant time, assuming a sufficient number of processors.

2.22. Loop 22

Loop 22 is from a Planckian Distribution procedure. It returns the values of two one-dimensional arrays, Y and W. Although $W(i)$ is a function of $Y(i)$, neither are functions of values calculated on earlier iterations. Since there are no dependencies across the iterations, the loop is O(1). The loop can be vectorized by building a vector mask for the boolean expression on line 6.

2.23. Loop 23

Loop 23 is a 2-D Implicit Hydrodynamics fragment. The subarray $ZA(2,\ldots,n, 2,\ldots,6)$ is recomputed. It is an interesting loop because the computation progresses as a series of wavefronts from the array's top left-hand corner to its lower right-hand corner. The new value of $ZA(k, j)$ is a function of the element above it, $ZA(k-1, j)$, below it, $ZA(k+1, j)$, to its left, $ZA(k, j-1)$, and to its right, $ZA(k, j+1)$. Furthermore, $ZA(k-1, j)$ and $ZA(k, j-1)$ refer to new values computed by the loop, while $ZA(k+1, j)$ and $ZA(k, j+1)$ refer to old values (Fig. 11). Since $ZA(k-1, j)$ and $ZA(k, j-1)$ are in the same column and row as $ZA(k, j)$, neither the elements along a row nor a column can be computed in parallel. The function as written is O(n^2).

However, notice that the computation for $ZA(k, j)$ is independent of $ZA(k-1, j+1)$ and $ZA(k+1, j-1)$; in fact, any element whose indexes sum to $k+j$ is independent of any other element whose index sum is $k+j$. Such elements define the left-to-right diagonals of ZA (Fig. 12). The computations for the elements lying along such diagonal are data independent and can be computed in parallel. The only constraint is that the diagonal just above be new and the one

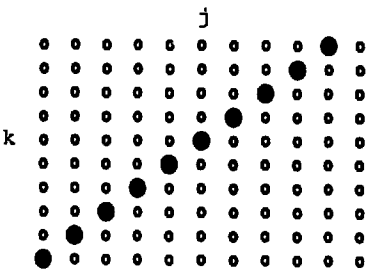

Fig. 12. A left-to-right diagonal of ZA.

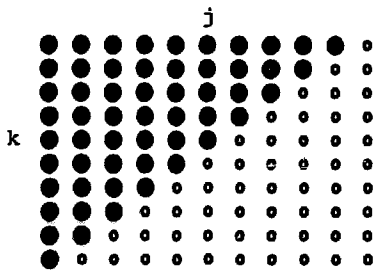

Fig. 13. Parallel execution of Loop 23.

just below be old. This can be guaranteed by executing the loop from the matrix's upper left-hand coerner to its lower right-hand corner along the diagonals (Fig. 13). So by rewriting the loop to execute across the diagonals instead of down the rows, the computations at each iteration (i.e., along each diagonal) can execute in parallel. Since there are $n + m$ diagonals, the loop's complexity is linear.

The loop can be partially vectorized over k by factoring the inner loop into two separate loops (Fig. 14). The first computes a vector of modified QA values and the second computes $ZA(k, j)$, $2 \leqslant k \leqslant n$. By removing the term $ZA(k-1, j) * ZV(k, j)$ from the QA equation and placing it in the ZA equation, the QA equation is made data independent in k without changing the loop's final results. The first subloop can be vectorized; the second subloop is however still sequential. Since the bulk of the computations have been vectorized, the modified loop should execute faster than the original loop.

2.24. Loop 24

Loop 24 finds the location of the first minimum in X. That is to say, Loop 24 returns the index of the first occurrence of x, where x is the minimum value in X. Any algorithm which finds the minimum value of an array can be extended to find the first occurrence of the minimum. An array's minimum value an be found by constructing a binary tree from leaves to root. The leaves are initialized to the values of the array, and each interior node is set to the minimum of its two descendents. At the end, the root will hold the array's minimum value. Obviously, the tree has log n levels and so, the algorithm is O(log n).

There are two ways to extend this parallel minimum-value algorithm to find the location of the first minimum. One way is to initialize each leaf node to a tuple, [value, index]; and set each intermediate node to its descendent's tuple with the smaller value, or if the values are equal to the tuple with the smaller index. The second approach is to append the indexes to the values forming an array of composite numbers (value · index), and run the minimum-value algorithm

```
      subroutine Loop23(n,ZA,ZB,ZR,ZU,ZV,ZZ)                              1
      integer j,k,n                                                        2
      double precision QA(101),ZA(101,7),ZB(101,7)                         3
      double precision ZR(101,7),ZU(101,7),ZV(101,7),ZZ(101,7)             4
      Do 233 j = 2,6                                                       5
         Do 231 k = 2,n                                                    6
231         QA(k) = ZA(k,j+1)*ZR(k,j) + ZA(k,j-1)*ZB(k,j) +               7
     .              ZA(k+1,j)*ZU(k,j) + ZZ(k,j) - ZA(k,j)                  8
         Do 232 k = 2,n                                                    9
232         ZA(k,j) = ZA(k,j) + 0.175d0*(QA(k) + ZA(k-1,j)*ZV(k,j))        10
233   continue                                                            11
      return                                                              12
      end                                                                 13
```

Fig. 14. Loop 23 with split inner loop.

on this array. Once the minimum is found, the high-order bits corresponding to the value can be removed leaving just the index.

3. Summary

This plaper has presented the 24 Livermore Loops. The content of each loop is described and the complexity of each is given. Complexity has been defined as the number of computational steps requireed to complete a loop, assuming that all actions computable at a particular instant are actually executed concurrently. We hope that this paper gives a clearer picture of the contents and mathematics of the Loops than was previously avaialble, and will allow users to achieve cleaner and more efficient implementation of the loops for their particular systems. In turn, this should result in more accurate evaluation of computer systems, especially parallel machines.

Acknowledgment

The author wishes to thank Frank McMahon for his insights on the inner workings of the Loops, and Steve Skedzielewski and Rod Oldehoeft for their help in preparing this paper. This work was supported (in part) by the Applied Mathematical Sciences subprogram of the Office of Energy Research, U.S. Department of Energy, by Lawrence Livermore National Laboratory under contract No. W-7405-Eng-48.

Appendix A. The Livermore Loops

```
      Subroutine Loop1(n,Q,R,T,X,Y,Z)                                    1
      integer n,k                                                        2
      double precision Q,R,T,X(1001),Y(1001),Z(1001)                     3
      Do 1 k = 1,n                                                       4
1        X(k) = Q + (Y(k) * ((R * Z(k+10)) + (T * Z(k+11))))             5
      return                                                             6
      end                                                                7


      subroutine Loop2(n,V,X)                                            1
      integer n,k,i,IL,IPNT,IPNTP                                        2
      double precision V(1001),X(1001)                                   3
      IL = n                                                             4
      IPNTP = 0                                                          5
21    IPNT = IPNTP                                                       6
      IPNTP = IPNTP + IL                                                 7
      IL = IL / 2                                                        8
      i = IPNTP                                                          9
      Do 20 k = IPNT+2,IPNTP,2                                           10
         i = i + 1                                                       11
20       X(i) = X(k) - (V(k) * X(k-1)) + (V(k+1) * X(k+1))               12
      If(IL .GT. 1)  GOTO 21                                             13
      return                                                             14
      end                                                                15
```

```
      subroutine Loop3(n,Q,X,Z)                                          1
      integer n                                                          2
      double precision Q,X(1001),Z(1001)                                 3
      Q = 0.0d0                                                          4
      Do 3 k = 1,n                                                       5
3        Q = Q + (Z(k) * X(k))                                           6
      return                                                             7
      end                                                                8


      subroutine Loop4(n,X,Y)                                            1
      integer n,k,j,lw                                                   2
      double precision X(1001),Y(1001)                                   3
      Do 40 k = 7,1001,(1001 - 7)/2                                      4
         lw = k - 6                                                      5
         Do 41 j = 5,n,5                                                 6
            X(k-1) = X(k-1) - (X(lw) * Y(j))                             7
41          lw = lw + 1                                                  8
40       X(k-1) = Y(5) * X(k-1)                                          9
      return                                                            10
      end                                                               11


      subroutine Loop5(n,X,Y,Z)                                          1
      integer i,n                                                        2
      double precision X(1001),Y(1001),Z(1001)                           3
      Do 5 i = 2,n                                                       4
5        X(i) = Z(i) * (Y(i) - X(i-1))                                   5
      return                                                             6
      end                                                                7


      subroutine Loop6(n,B,W)                                            1
      integer n                                                          2
      double precision B(64,64),W(1001)                                  3
      Do 6 i = 2,n                                                       4
      Do 6 k = 1,i-1                                                     5
6        W(i) = W(i) + (B(i,k) * W(i-k))                                 6
      return                                                             7
      end                                                                8


      subroutine Loop7(n,R,T,U,X,Y,Z)                                    1
      integer n                                                          2
      double precision R,T,U(1001),X(1001),Y(1001),Z(1001)               3
      Do 7 k = 1,n                                                       4
         X(k) = U(k) +                                                   5
     .      (R * (Z(k) + (R * Y(k)))) +                                  6
     .      (T * (U(k+3) + (R * (U(k+2) + (R * U(k+1)))) +               7
     .           (T * (U(k+6) + (R * (U(k+5) + (R * U(k+4)))))))))       8
7        Continue                                                        9
      return                                                            10
      end                                                               11


      subroutine Loop8(n,A11,A12,A13,A21,A22,A23,                        1
     .                 A31,A32,A33,SIG,U1,U2,U3)                         2
      integer n,kx,ky,nl1,nl2                                            3
```

```
      double precision DU1(101),DU2(101),DU3(101)                          4
      double precision U1(5,101,2),U2(5,101,2),U3(5,101,2)                 5
      double precision A11,A12,A13,A21,A22,A23,A31,A32,A33,SIG             6
      nl1 = 1                                                              7
      nl2 = 2                                                              8
      Do 8 kx = 2,3                                                        9
      Do 8 ky = 2,n                                                       10
         DU1(ky) = U1(kx,ky+1,nl1) - U1(kx,ky-1,nl1)                      11
         DU2(ky) = U2(kx,ky+1,nl1) - U2(kx,ky-1,nl1)                      12
         DU3(ky) = U3(kx,ky+1,nl1) - U3(kx,ky-1,nl1)                      13
         U1(kx,ky,nl2) = U1(kx,ky,nl1) + (A11 * DU1(ky)) + (A12 *         14
     .     DU2(ky)) + (A13 * DU3(ky)) + (SIG * (U1(kx+1,ky,nl1) -         15
     .      (2.0d0 * U1(kx,ky,nl1)) + U1(kx-1,ky,nl1)))                   16
         U2(kx,ky,nl2) = U2(kx,ky,nl1) + (A21 * DU1(ky)) + (A22 *         17
     .     DU2(ky)) + (A23 * DU3(ky)) + (SIG * (U2(kx+1,ky,nl1) -         18
     .      (2.0d0 * U2(kx,ky,nl1)) + U2(kx-1,ky,nl1)))                   19
         U3(kx,ky,nl2) = U3(kx,ky,nl1) + (A31 * DU1(ky)) + (A32 *         20
     .     DU2(ky)) + (A33 * DU3(ky)) + (SIG * (U3(kx+1,ky,nl1) -         21
     .      (2.0d0 * U3(kx,ky,nl1)) + U3(kx-1,ky,nl1)))                   22
8     Continue                                                            23
      return                                                              24
      end                                                                 25
```

```
      subroutine Loop9(n,CO,DM22,DM23,DM24,DM25,DM26,DM27,DM28,PX)         1
      integer i,n                                                          2
      double precision CO,DM22,DM23,DM24,DM25,DM26,DM27,DM28,PX(25,101)    3
      Do 9 i = 1,n                                                         4
         PX(1,i) = PX(3,i) +  (CO * (PX(5,i) + PX(6,i))) +                 5
     .             (DM28 * PX(13,i)) + (DM27 * PX(12,i)) +                6
     .             (DM26 * PX(11,i)) + (DM25 * PX(10,i)) +                7
     .             (DM24 * PX(9,i))  + (DM23 * PX(8,i))  +                8
     .             (DM22 * PX(7,i))                                       9
9     Continue                                                           10
      return                                                             11
      end                                                                12
```

```
      subroutine Loop10(n,CX,PX)                                          1
      integer i,n                                                         2
      double precision AR,BR,CR,CX(25,101),PX(25,101)                     3
      Do 10 i = 1,n                                                       4
         AR = CX(5,i)                                                     5
         BR = AR - PX(5,i)                                                6
         PX(5,i) = AR                                                     7
         CR = BR - PX(6,i)                                                8
         PX(6,i) = BR                                                     9
         AR = CR - PX(7,i)                                               10
         PX(7,i) = CR                                                    11
         BR = AR - PX(8,i)                                               12
         PX(8,i) = AR                                                    13
         CR = BR - PX(9,i)                                               14
         PX(9,i) = BR                                                    15
         AR = CR - PX(10,i)                                              16
         PX(10,i) = CR                                                   17
         BR = AR - PX(11,i)                                              18
         PX(11,i) = AR                                                   19
         CR = BR - PX(12,i)                                              20
         PX(12,i) = BR                                                   21
         PX(14,i) = CR - PX(13,i)                                        22
10       PX(13,i) = CR                                                   23
```

```
      return                                                          24
      end                                                             25


      subroutine Loop11(n,X,Y)                                         1
      integer k,n                                                      2
      double precision X(1001),Y(1001)                                 3
      X(1) = Y(1)                                                      4
      Do 11 k = 2,n                                                    5
11       X(k) = X(k-1) + Y(k)                                          6
      return                                                           7
      end                                                              8


      subroutine Loop12(n,X,Y)                                         1
      integer k,n                                                      2
      double precision X(1001),Y(1001)                                 3
      Do 12 k = 1,n                                                    4
12       X(k) = Y(k+1) - Y(k)                                          5
      return                                                           6
      end                                                              7


      subroutine Loop13(n,E,F,B,C,H,P,Y,Z)                             1
      integer n,ip,i1,j1,i2,j2,E(96),F(96)                             2
      double precision B(64,64),C(64,64),H(64,64)                      3
      double precision P(4,512),Y(1001),Z(1001)                        4
      Do 13 ip = 1,n                                                   5
         i1 = P(1,ip)                                                  6
         j1 = P(2,ip)                                                  7
         i1 = 1 + MOD2N(i1,64)                                         8
         j1 = 1 + MOD2N(j1,64)                                         9
         P(3,ip) = P(3,ip) + B(i1,j1)                                 10
         P(4,ip) = P(4,ip) + C(i1,j1)                                 11
         P(1,ip) = P(1,ip) + P(3,ip)                                  12
         P(2,ip) = P(2,ip) + P(4,ip)                                  13
         i2 = P(1,ip)                                                 14
         j2 = P(2,ip)                                                 15
         i2 = MOD2N(i2,64)                                            16
         j2 = MOD2N(j2,64)                                            17
         P(1,ip) = P(1,ip) + Y(i2+32)                                 18
         P(2,ip) = P(2,ip) + Z(j2+32)                                 19
         i2 = i2 + E(i2+32)                                           20
         j2 = j2 + F(j2+32)                                           21
13       H(i2,j2) = H(i2,j2) + 1.0d0                                  22
      return                                                          23
      end                                                             24


      subroutine Loop14(n,FLX,DEX,DEX1,EX,EX1,                         1
                        GRD,IR,IX,RH,RX,VX,XI,XX)                      2
      integer n,k,IR(1001),IX(1001)                                    3
      double precision FLX,DEX(1001),DEX1(1001)                        4
      double precision EX(1001),EX1(1001),GRD(1001)                    5
      double precision RH(1001),RX(1001),VX(1001),XI(1001),XX(1001)    6
      Do 141 k = 1,n                                                   7
         VX(k) = 0.0d0                                                 8
         XX(k) = 0.0d0                                                 9
         IX(k) = INT(GRD(k))                                          10
```

```
      XI(k) = FLOAT(IX(k))                                                  11
      EX1(k) = EX(IX(k))                                                    12
      DEX1(k) = DEX(IX(k))                                                  13
141   Do 142 k = 1,n                                                        14
      VX(k) = VX(k) + EX1(k) + (DEX1(k) * (XX(k) - XI(k)))                  15
      XX(k) = XX(k) + VX(k) + FLX                                           16
      IR(k) = XX(k)                                                         17
      RX(k) = XX(k) - IR(k)                                                 18
      IR(k) = MOD2N(IR(k),512) + 1                                          19
142   XX(k) = RX(k) + IR(k)                                                 20
      Do 140 k = 1,n                                                        21
      RH(IR(k)) = RH(IR(k)) - RX(k) + 1.0d0                                 22
140   RH(IR(k) + 1) = RH(IR(k) + 1) + RX(k)                                 23
      return                                                                24
      end                                                                   25


      subroutine Loop15(n,VF,VG,VH,VS,VY)                                   1
      integer j,k                                                           2
      double precision R,S,T,VF(101,7),VG(101,7)                            3
      double precision VH(101,7),VS(101,7),VY(101,25)                       4
      do 1500 j = 2,7                                                       5
      do 1500 k = 2,n                                                       6
      if(j-7) 1502,1501,1501                                                7
1501  VY(k,j) = 0.0d0                                                       8
      goto 1500                                                             9
1502  if(VH(k,j+1) - VH(k,j)) 1504,1504,1503                                10
1503  T = 0.053d0                                                           11
      goto 1505                                                             12
1504  T = 0.073d0                                                           13
1505  if(VF(k,j) - VF(k-1,j)) 1506,1507,1507                                14
1506  R = max(VH(k-1,j), VH(k-1,j+1))                                       15
      S = VF(k-1,j)                                                         16
      goto 1508                                                             17
1507  R = max(VH(k,j), VH(k,j+1))                                           18
      S = VF(k,j)                                                           19
1508  VY(k,j) = dsqrt(VG(k,j)**2 + R * R) * T / S                           20
1509  if(k-n) 1511,1510,1510                                                21
1510  VS(k,j) = 0.0d0                                                       22
      goto 1500                                                             23
1511  if(VF(k,j) - VF(k,j-1)) 1512,1513,1513                                24
1512  R = max(VG(k,j-1), VG(k+1,j-1))                                       25
      S = VF(k,j-1)                                                         26
      T = 0.073d0                                                           27
      goto 1514                                                             28
1513  R = max(VG(k,j), VG(k+1,j))                                           29
      S = VF(k,j)                                                           30
      T = 0.053d0                                                           31
1514  VS(k,j) = dsqrt(VH(k,j)**2 + R * R) * T / S                           32
1500  continue                                                              33
      return                                                                34
      end                                                                   35


      subroutine Loop16(n,m,j4,R,S,T,D,PLAN,ZONE)                           1
      integer n,m,i1,j2,j4,j5,k,II,LB,ZONE(300)                             2
      double precision R,S,T,D(300),PLAN(300)                               3
      m = 1                                                                 4
      i1 = m                                                                5
      II = n / 3                                                            6
      LB = II + II                                                          7
```

```
1601 j2 = (n + n) * (m - 1) + 1                                          8
     do 1613 k = 1,n                                                     9
        j4 = j2 + k + k                                                  10
        j5 = ZONE(j4)                                                    11
        if(j5 - n) 1603,1614,1609                                        12
1602    if(j5 - n + II) 1605,1604,1604                                   13
1603    if(j5 - n + LB) 1606,1602,1602                                   14
1604    if(PLAN(j5) - R) 1608,1615,1607                                  15
1605    if(PLAN(j5) - S) 1608,1615,1607                                  16
1606    if(PLAN(j5) - T) 1608,1615,1607                                  17
1607    if(ZONE(j4 - 1)) 1610,1616,1613                                  18
1608    if(ZONE(j4 - 1)) 1613,1616,1610                                  19
1609    if(D(j5) - (D(j5-1) * (T - D(j5-2))**2 +                         20
     .     (S - D(j5-3))**2 + (R - D(j5-4))**2)) 1608,1615,1607          21
1610    m = m + 1                                                        22
        if(m - ZONE(1)) 1612,1612,1611                                   23
1611    m = 1                                                            24
1612    if(i1 - m) 1601,1615,1601                                        25
1613 continue                                                           26
1614 continue                                                           27
1615 continue                                                           28
1616 continue                                                           29
     return                                                             30
     end                                                                31


     subroutine Loop17(n,VE3,VLIN,VLR,VSP,VSTP,VXND,VXNE)               1
     integer i,n                                                        2
     double precision E3,E6,XNC,XNEI,XNM,VE3(101),VLIN(101)             3
     double precision VLR(101),VSP(101),VSTP(101),VXND(101),VXNE(101)   4
     i = n                                                              5
     XNM = 1.0d0 / 3.0d0                                                6
     E6 = 1.03d0 / 3.07d0                                               7
     goto 172                                                           8
171  E6 = XNM * VSP(i) + VSTP(i)                                        9
     VXNE(i) = E6                                                       10
     XNM = E6                                                           11
     VE3(i) = E6                                                        12
     i = i - 1                                                          13
     if(i .eq. 1) goto 170                                              14
172  E3 = XNM * VLR(i) + VLIN(i)                                        15
     XNC = 5.0d0 / 3.0d0 * E3                                           16
     XNEI = VXNE(i)                                                     17
     VXND(i) = E6                                                       18
     if(XNM .gt. XNC) goto 171                                          19
     if (XNEI .gt. XNC) goto 171                                        20
     VE3(i) = E3                                                        21
     E6 = E3 + E3 - XNM                                                 22
     VXNE(i) = E3 + E3 - XNEI                                           23
     XNM = E6                                                           24
     i = i - 1                                                          25
     if(i .ne. 1) goto 172                                              26
170  continue                                                           27
     return                                                             28
     end                                                                29


     subroutine Loop18(n,S,T,ZA,ZB,ZM,ZP,ZQ,ZR,ZU,ZV,ZZ)               1
     integer k,j,n                                                      2
     double precision S,T,ZA(101,7),ZB(101,7),ZM(101,7),ZP(101,7)       3
     double precision ZQ(101,7),ZR(101,7),ZU(101,7),ZV(101,7),ZZ(101,7) 4
```

```
      do 181 k = 2,6                                                    5
      do 181 j = 2,n                                                    6
         ZA(j,k) =                                                      7
     .   (ZP(j-1,k+1) + ZQ(j-1,k+1) - ZP(j-1,k) - ZQ(j-1,k)) *         8
     .   (ZR(j,k) + ZR(j-1,k)) / (ZM(j-1,k) + ZM(j-1,k+1))             9
         ZB(j,k) =                                                      10
     .   (ZP(j-1,k) + ZQ(j-1,k) - ZP(j,k) - ZQ(j,k)) *                 11
     .   (ZR(j,k) + ZR(j,k-1)) / (ZM(j,k) + ZM(j-1,k))                 12
181   continue                                                          13
      do 182 k = 2,6                                                    14
      do 182 j = 2,n                                                    15
         ZU(j,k) = ZU(j,k) + S *                                        16
     .             (ZA(j,k)   * (ZZ(j,k) - ZZ(j+1,k)) -                17
     .              ZA(j-1,k) * (ZZ(j,k) - ZZ(j-1,k)) -                18
     .              ZB(j,k)   * (ZZ(j,k) - ZZ(j,k-1)) +                19
     .              ZB(j,k+1) * (ZZ(j,k) - ZZ(j,k+1)))                 20
         ZV(j,k) = ZV(j,k) + S *                                        21
     .             (ZA(j,k)   * (ZR(j,k) - ZR(j+1,k)) -                22
     .              ZA(j-1,k) * (ZR(j,k) - ZR(j-1,k)) -                23
     .              ZB(j,k)   * (ZR(j,k) - ZR(j,k-1)) +                24
     .              ZB(j,k+1) * (ZR(j,k) - ZR(j,k+1)))                 25
182   continue                                                          26
      do 183 k = 2,6                                                    27
      do 183 j = 2,n                                                    28
         ZR(j,k) = ZR(j,k) + T * ZU(j,k)                                29
         ZZ(j,k) = ZZ(j,k) + T * ZV(j,k)                                30
183   continue                                                          31
      return                                                            32
      end                                                               33
```

```
      subroutine Loop19(n,STB5,B5,SA,SB)                                1
      integer i,k,n                                                     2
      double precision STB5,B5(101),SA(101),SB(101)                     3
      do 191 k = 1,n                                                    4
         B5(k) = SA(k) + STB5 * SB(k)                                   5
191      STB5 = B5(k) - STB5                                            6
      do 193 i = 1,n                                                    7
         k = n - i + 1                                                  8
         B5(k) = SA(k) + STB5 * SB(k)                                   9
193      STB5 = B5(k) - STB5                                            10
      return                                                            11
      end                                                               12
```

```
      subroutine Loop20(n,DK,S,T,G,U,V,VX,W,X,XX,Y,Z)                   1
      integer k,n                                                       2
      double precision DK,DI,DN,S,T,G(1001),U(1001),V(1001),VX(1001)    3
      double precision W(1001),X(1001),XX(1001),Y(1001),Z(1001)         4
      do 20 k = 1,n                                                     5
         DI = Y(k) - (G(k) / (XX(k) + DK))                              6
         DN = 0.2d0                                                     7
         if(DI .NE. 0.0d0) DN = max(s, min(Z(k)/DI, T))                 8
         X(k) = ((W(k) + V(k) * DN) * XX(k) + U(k)) /                   9
                (VX(k) + V(k) * DN)                                     10
         XX(k+1) = (X(k) - XX(k)) * DN + XX(k)                          11
20     continue                                                         12
       return                                                           13
       end                                                              14
```

```
      subroutine Loop21(n,CX,PX,VY)                                      1
      integer i,j,k,n                                                    2
      double precision CX(25,101),PX(25,101),VY(101,25)                  3
      do 21 k = 1,25                                                     4
      do 21 i = 1,25                                                     5
      do 21 j = 1,n                                                      6
         PX(i,j) = PX(i,j) + VY(i,k) * CX(k,j)                           7
21    continue                                                           8
      return                                                             9
      end                                                               10


      subroutine Loop22(n,U,V,W,X,Y)                                     1
      integer k,n                                                        2
      double precision U(1001),V(1001),W(1001),X(1001),Y(1001)           3
      do 22 k = 1,n                                                      4
         Y(k) = 20.0d0                                                   5
         if(U(k) .lt. 20.0d0 * V(k)) Y(k) = U(k) / V(k)                  6
         W(k) = X(k) / (dexp(Y(k)) - 1.0d0)                              7
22    continue                                                           8
      return                                                             9
      end                                                               10


      subroutine Loop23(n,ZA,ZB,ZR,ZU,ZV,ZZ)                            1
      integer j,k,n                                                      2
      double precision QA,ZA(101,7),ZB(101,7),ZR(101,7)                  3
      double precision ZU(101,7),ZV(101,7),ZZ(101,7)                     4
      do 23 j = 2,6                                                      5
      do 23 k = 2,n                                                      6
         QA = ZA(k,j+1) * ZR(k,j) + ZA(k,j-1) * ZB(k,j) +                7
              ZA(k+1,j) * ZU(k,j) + ZA(k-1,j) * ZV(k,j) + ZZ(k,j)        8
23       ZA(k,j) = ZA(k,j) + 0.175d0 * (QA - ZA(k,j))                    9
      return                                                            10
      end                                                               11


      subroutine Loop24(n,max24,X)                                       1
      integer k,max24,n                                                  2
      double precision X(1001)                                           3
      max24 = 1                                                          4
      do 24 k = 2,n                                                      5
24       if(X(k) .lt. X(max24)) max24 = k                                6
      return                                                             7
      end                                                                8
```

References

[1] A.V. Aho, J.E. Hopcroft and J.D. Ullman, *The Design and Analysis of Computer Algorithms* (Addison-Wesley, Reading, MA, 1974).

[2] C.N. Arnold, Performance evaluation of three automatic vectorizer packages, *Proc. IEEE Spring Computer Conference* (1982) 235–242.

[3] H.S. Kogge, Parallel solution of recurrence problems, *IBM J. Res. Development* (March 1974) 138–148.

[4] H.S. Kogge and P.M. Stone, A parallel algorithm for the efficient solution of a general class of recurrence equations, *IEEE Trans. Comput.* **22** (8) (1973) 786–793.

[5] D.J. Kuck, R.H. Kuhn et al., Dependence graphs and compiler optimizations, *Conference Record of the 8th ACM Symposium on Principles of Programming Languages* (1981).

[6] F.H. McMahon, L.L.N.L. FORTRAN kernels: MFLOPS, Lawrence Livermore National Laboratory, 1986.

[7] F.H. McMahon, Livermore FORTRAN kernels: A computer test of the numerical performance range, UCRL-53745, University of California, Lawrence Livermore National Laboratory, in publication.

[8] J.P. Riganati and P.B. Schneck, Supercomputing, *Computer* **17** (10) (1984) 97–113.