On the Automatic Parallelization of Sparse and Irregular Fortran Codes

Rafael Asenjo, Eladio Gutierrez, Yuan Lin, David Padua, Bill Pottenger, Emilio Zapata

Center for Supercomputing Research and Development
University of Illinois at Urbana-Champaign
1308 West Main Street, Urbana, Illinois 61801-2307
217-333-6578, fax: 217-244-1351
potteng,yuanlin,padua@csrd.uiuc.edu *

Dept. Arquitectura de Computadores
E.T.S. Ingenieros de Telecomunicacion
Complejo Tecnologico
Campus de Teatinos
Apdo. 4114 - 29080 MALAGA
+34-(9)5-213-27-91, fax: +34-(9)5-213-27-90
asenjo,eladio,ezapata@ac.uma.es

February 28, 1997

1 Introduction

Irregular memory access patterns have traditionally caused difficulties in the automatic detection of parallelism, and in many cases parallelization is prevented. These problems are nonetheless important in that a significant fraction of current applications are irregular in nature.

Some work has been done in the past, but very few studies have been made of complete codes [21]. This paper studies how well automatic parallelization techniques work on a collection of real codes with sparse and irregular access patterns. In conducting this work, we have compared existing technology in the commercial parallelizer PFA from SGI with the Polaris restructurer [7]. In cases

*This work is supported by U.S. Army contract #DABT63-95-C-0097 and is not necessarily representative of the positions or policies of the Army or the Government.

where performance was poor, we have done manual analysis and determined the techniques necessary for automatic parallelization.

2 The Benchmark Suite

Benchmark	Description	Origin	# lines	Serial exec (seconds)
CHOLESKY	Sparse Cholesky Factorization	HPF-2	1284	265
DSMC3D	Direct Simulation Monte Carlo	HPF-2	1794	482
EULER	Euler equations on 3-D grid	HPF-2	1990	303
GCCG	Computational fluid dynamics	Vienna	407	739
LANCZOS	Eigenvalues of symmetric matrices	Malaga	269	868
MVPRODUCT	Basic matrix operations	Malaga	342	477
NBFC	Molecular dynamics kernel	HPF-2	206	375
SpLU	Sparse LU Factorization	HPF-2	363	471

Table 1: Benchmark Codes

Table 1 summarizes the eight codes in the benchmark suite employed in our experiments. The suite consists of a collection of sparse and irregular application programs as well as several kernels representing key computational elements present in sparse codes. Several of the benchmarks in our suite are derived from the set of motivating applications for the HPF-2 effort [11]. Exceptions include the kernels MVPRODUCT and LANCZOS which were developed as part of this project. The sparse CFD code GCCG was developed at the Institute for Software Technology and Parallel Systems at the University of Vienna, Austria.

2.1 CHOLESKY

The sparse cholesky factorization of a symmetric positive definite sparse matrix A produces a lower triangular matrix L such that $A = LL^T$. This factorization is used in direct methods to solve systems of linear equations. An example of the type of access pattern seen in CHOLESKY is depicted below:

```
do s = 1,nsu
   do j = isu(s),isu(s+1)-1
      snhead(j) = isu(s)
      nafter(j) = isu(s+1) - 1 - j
   enddo
 enddo
```

The indirectly referenced loop bounds of the inner j loop vary across iterations of the outer i loop. The Harwell-Boeing matrix BCSSTK30 was used as input for this benchmark [10].

2.2 DSMC3D

DSMC3D is a modification of the DSMC (Direct Simulation Monte Carlo) benchmark in 3 dimensions. DSMC implements a simulation of the behavior of particles of a gas in space using the Monte Carlo method [5]. An example of one of the access patterns occurring in this application is abstracted below:

```
 do i = 1, NM
    if (mcell(i)+1 .eq. ncell(i)) then
          cellx(mcell(i)) = cellx(mcell(i)) + 1
    endif
  enddo
```

In the above accumulation into `cellx`, subscripted subscripts occur on both the left and right-hand sides of assignment statements. Other more complex indirection patterns that we find in this irregular code include the following segment excerpted from the SELECT subroutine:

```
      k=int(rnd()*ic(2,n,nn))
      l=ir(k)
      k=int(rnd()*iscg(2,msc,mm))
      m=ir(k)
      do 200 j=1,3
        vrc(j)=pv(j,l)-pv(j,m)
200   continue
```

2.3 EULER

EULER is an application which solves the Euler equations on an irregular mesh. The computation is based on an indirectly referenced description of the grid. In addition, indirection is employed on both sides of assignment statements. The following code abstract exemplifies this two-level pattern of indirection:

```
     do  ng=1,ndegrp
      do  i=ndevec(ng,1),ndevec(ng,2)
      n1        = nde(i,1)
      n2        = nde(i,2)
      pw(n1,1)  = pw(n1,1) + qw(n2,1)*eps(i)
      pw(n2,1)  = pw(n2,1) + qw(n1,1)*eps(i)
     enddo
    enddo
```

2.4 GCCG

GCCG is an example of a computational fluid dynamics solver. The access pattern is similar to that found in finite element methods where the value of an element is determined by the contribution of neighbors selected using subscripted subscripts. As a result, indirection occurs on the right-hand-side of the computed expressions.

```
       do  nc=nintci,nintcf
        direc2(nc)=bp(nc)*direc1(nc)
     *            -bs(nc)*direc1(lcc(nc,1))
     *            -bw(nc)*direc1(lcc(nc,4))
     *            -bl(nc)*direc1(lcc(nc,5))
       enddo
```

2.5 LANCZOS

The lanczos algorithm with full reorthogonalization determines the eigenvalues of a symmetric matrix [12]. LANCZOS is an implementation of the lanczos algorithm for sparse matrices. The key computational elements are the calculation of a sparse matrix-vector product and the reorthogonalization of a dense work matrix. Access patterns include subscripted subscripts on the right-hand-side of assignment statements as the following excerpt demonstrates:

```
         do  j=1,a_nr
          do  k=ar(j),ar(j+1)-1
           r(j)=r(j)+ad(k)*q(ac(k),i)
          enddo
         enddo
```

The matrix 1138_BUS of Harwell-Boeing collection was used as input for this benchmark.

2.6 MVPRODUCT

MVPRODUCT is a set of basic sparse matrix operations including sparse matrix-vector multiplication and the product and sum of two sparse matrices [3, 12]. The representation of the sparse matrices employs two different schemes: *compressed row storage* (CRS) and *compressed column storage* (CCS) [22]. The access pattern is demonstrated by the following code abstract:

```
        do  i=1,a_nr
         do  k=1,b_nc
          do  ja=ar(i),ar(i+1)-1
           do  jb=bc(k),bc(k+1)-1
             if (ac(ja).eq.br(jb)) THEN
              c(i,k)=c(i,k)
     &               + ad(ja)*bd(jb)
             endif
           enddo
          enddo
         enddo
        enddo
```

Here indirection occurs on the right-hand-side of the computed expressions. The matrix BCSSTK14 from the Harwell-Boeing collection has been used as input to this benchmark.

2.7 NBFC

The calculation of non-bonded forces forms a key element of many molecular dynamics computations [8]. NBFC computes an electro-static interaction between particles where the forces acting on an atom are calculated from a list of neighboring atoms. Similar to the DSMC3D benchmark, the data access pattern in this sparse code has indirection on both sides of the computed expressions:

```
       do k = 1, ntimestep
        do i = 1, natom
         do  j = inblo(i),inblo(i+1)-1
          dx(jnb(j)) = dx(jnb(j)) - (x(i) - x(jnb(j)))
          dx(i)      = dx(i) + (x(i) - x(jnb(j)))
         enddo
        enddo
       enddo
```

2.8 SpLU

SpLU computes the LU factorization of a sparse matrix. The LU factorization is used in several methods which solve sparse linear systems of equations. The factorization of a matrix A results in two matrices, L (lower triangular) and U (upper triangular), and two permutation vectors π and ρ such that: $A_{\pi_i \rho_j} = (LU)_{ij}$.

The pattern of access to arrays in SpLU includes indirectly referenced loop bounds across an iteration space traversed by a loop induction variable:

```
do i=cptr1(j),cptr2(j)
   a(shift)=a(i)
   r(shift)=r(i)
   shift = shift + 1
enddo
```

SpLU is a right-looking sparse LU factorization based on the CCS data structure. This algorithm is somewhat slower than the MA48 code from Harwell Subroutine Library [10], a left-looking standard benchmark for factorization. The motivation for developing a right-looking algorithm derived from the lack of significant parallelism in MA48. This led to the inclusion of the original C version of SpLU in the suite of HPF-2 motivating applications. The version of SpLU included in our benchmark suite is a Fortran implementation by the authors of the original HPF-2 version [2]. The sparse matrix lns_3937 from the Harwell-Boeing collection was used as input for the results reported in this paper.

3 Parallelizing Techniques and Transformations

Several new techniques have been developed and existing techniques employed to parallelize this suite of sparse and irregular codes:

- Histogram Reductions
- Random Number Generator Substitution
- Proving Monotonicity of Index Arrays
- Proving Ranges of Induction Variables Non-Overlapping
- Copy-in and Copy-out
- Loops with Conditional Exits

3.1 Histogram Reductions

The following code portrays a reduction on the array A which involves a loop-variant subscript function $f(i,j)$.

```
 do i=1,n
   do j = 1, m
      k = f(i,j)
      a(k) = a(k) + expression
   enddo
enddo
```

Due to the loop-variant nature of the subscript function f, loop-carried dependences may be present at run-time. This pattern occurs commonly in many codes, both sparse and non-sparse, and is termed a *histogram* reduction [19, 14].

In our study of the benchmark suite we have found that histogram reductions occur in key computational loops in all four of the benchmarks derived from the HPF-2 motivating suite: NBFC, CHOLESKY, DSMC3D, and EULER. The parallelization of histogram reductions is based on a run-time technique which depends on the associativity of the operation being performed. The **Polaris** parallelizing restructurer recognizes and transforms histogram reductions [7].

The parallelizing transformation takes one of three forms: *critical section*, *privatized*, *expanded.* Each approach is discussed and exemplified below. The language used in the examples is based on IBM's Parallel Fortran [13].

- Critical Section

The first approach involves the insertion of synchronization primitives around each reduction statement, making the sum operation atomic. In our example the reduction statement would be enclosed by a lock/unlock pair:

```
parallel loop i=1,n
  do j = 1, m
     k = f(i,j)
     call lock
        a(k) = a(k) + expression
     call unlock
  enddo
enddo
```

This is an elegant solution on architectures which provide fast synchronization primitives.

- Privatized

In *privatized* reductions, duplicates of the reduction variable that are private to each processor are created and used in the reduction statements in place of the original variable. The following code exemplifies this transformation:

```
 parallel loop i=1,n
   private a_p(sz)
   dofirst
     a_p(1:sz) = 0
   doevery
     do j = 1, m
        k = f(i,j)
        a_p(k) = a_p(k) + expression
     enddo
   dofinal lock
     a(1:sz) = a(1:sz) + a_p(1:sz)
enddo
```

Each processor executes the `dofirst` section of the parallel loop once at the beginning of their slice of the iteration space. The `doevery` section of the loop is executed every iteration. The `dofinal` section of the code is executed once by each processor after completion of its slice of the iteration space. The `lock` argument to `dofinal` indicates that the code be enclosed in a critical section.

- Expanded

The third approach employs expansion to accomplish the same functionality as the privatizing transformation. Rather than allocating loop-private copies of the reduction variable, the variable is expanded by the number of threads participating in the computation. All reduction variables are replaced by references to this new, global array, and the newly created dimension is indexed by the processor number executing the current iteration.

The initialization and cross-processor sums take place in separate loops preceding and following the original loop, respectively. In this approach there is no need for synchronization, and both loops can be executed in parallel:

```
parallel loop j=1,threads
   do i=1,n
      a_e(i,j) = 0
   enddo
enddo
```

```
parallel loop i = 1, n
private int tid = thread\_id()
   do j = 1, m
      k = f(i,j)
      a_e(k,tid) = a_e(k,tid) + expression
   enddo
enddo
parallel loop i=1,n
   do j=1,threads
      a(i) = a(i) + a_e(i,j)
   enddo
enddo
```

3.2 Random Number Generator Substitution

Two of the codes in our suite contain calls to pseudo-random number generators (RNGs) within computationally important loops. Calls to routines of this type serialize a loop. Recent work by Bailey and Mascagni involves the development and standardization of robust thread-parallel pseudo-random number generators based on lagged Fibonacci series [20, 18]. RNGs such as these can be used to replace single-threaded generators.

Although the generation of pseudo-random numbers is not an associative operation, it can be said to be associative if we consider the fact that the particular pseudo-random number returned by a generator is not important as long as it is truly "random" [23]. In this sense, the substitution of RNGs can be viewed as the replacement of a non-associative, single-threaded RNG with an associative RNG.

We have come across the following cases involving RNGs in various codes in our test suites:

- Calls to single-threaded library routines
- Calls to single-threaded user routines of one of the following types
  - linear congruential generators
  - lagged Fibonacci generators

3.2.1 Single-threaded library routines

We have determined that calls to RNG library routines occur in two computationally important loops in a test suite used in the evaluation of the FALCON MATLAB compiler [9]. In addition, random() is called in INITIA_DO2000 in the Perfect Club benchmark MDG [4]. The CHOLESKY benchmark in our suite also calls the rand() library routine in an important loop. In three of these cases, the RNG call is the only factor preventing parallelization of the loop after application of the techniques implemented in the current Polaris restructurer. In CHOLESKY, a compile-time analysis of the array access pattern reveals a straightforward test which is sufficient to prove independence if the random number generation can be parallelized (see Section 3.3). In each of these cases, the loop-carried dependence can be broken by replacing these calls with a thread-parallel RNG.

3.2.2 Calls to single-threaded RNGs implemented in the program

Several codes in well-known test suites contain implementations of RNGs of various types. The Perfect Club benchmark QCD, for example, contains the routines PRANF, LADD, and LMULT which together implement a *linear congruential* pseudo-random number generator [15]. Similarly, the DSMC3D benchmark in our sparse/irregular suite implements a linear congruential generator based on work described in [16]. A third example occurs in the SPEC CFP95 benchmark su2cor which implements a lagged Fibonacci generator as described in the introduction to this section.

Lagged Fibonacci generators such as that implemented in su2cor take the form of a recurrence relation. Such relations can be automatically detected using pattern recognition techniques [1]. General techniques for solving linear recurrences of this type are well known [17], and closed-forms for such recurrences can be computed at compile-time thereby breaking loop-carried dependences.

In cases where RNGs are not explicitly coded as linear recurrences, other techniques must be employed. Currently we plan to develop a directive interface which can be used to identify RNGs. This requires input from the programmer, and therefore the parallelization becomes semi-automatic. Fully-automatic techniques which are based on the recognition of a stream of pseudo-random numbers at compile-time are under investigation.

3.3 Proving Monotonicity of Index Arrays

One of the major difficulties in automatically parallelizing sparse codes involves the analysis of subscripted array subscripts. The use of subscripted subscripts normally causes data dependence tests to draw overly conservative conclusions. The following portrays an example of such patterns:

```
do i = 1, n
 k = ia(i)
 a(k) = ...
end do
```

In the general case the subscript array *ia* must contain distinct indices if the outermost *i* loop is to be executed as a doall loop. Another pattern which occurs commonly in our suite involves the use of subscripted array subscripts in loop bounds:

```
do i = 1, n
 do j = ia(i), ia(i+1)-1
  a(j) = ...
 end do
end do
```

To parallelize the outermost loop in this case, the range $[ia(i), ia(i+1)-1]$ must be non-overlapping for all i. Although this condition may not hold generally, we have found that the index arrays in several of our sparse codes are monotonic in nature. This is due to the fact that matrices in sparse codes are often represented in a row-wise or column-wise format where the values of non-zero elements of the matrix are stored in a one dimensional array and pointers to the first and last elements of each row or column are stored in an index array. When this representation is used, the index array is non-decreasing.

The analysis of such access patterns has been considered difficult to accomplish at compile-time. However, using a combination of static, compile-time analysis and simple run-time tests, it is possible to prove that these index arrays are non-decreasing. In CHOLESKY, for example, it can be statically proven that the index array *isu* (initialized in SPARCHOL_GOTO290) is non-decreasing. This in turn is sufficient to prove that SPARCHOL_DO1017 can be executed as a doall loop.

In cases where the index array is read from input, it suffices to test that `ia(i+1).ge.ia(i)` for `i = 1,n` to prove that the ranges do not overlap. As mentioned earlier, the data representation employed in the codes studied in our suite guarantees that $n \equiv m + 1$, where n is the size of the index array and m is the number of columns or rows. In practice $n << \alpha$, where α is the number of non-zero entries in the matrix, and the overhead of this test is small.

When possible, this test should be inserted as part of the initial input operation. However, since this loop is essentially a reduction across ia, it can also be executed in parallel. Using techniques for handling loops with conditional exits discussed in Section 3.6, the loop execution time may be decreased even further.

We have found this pattern occurs in key computational loops in both CHOLESKY and SpLU.

3.4 Proving Ranges of Induction Variables Non-Overlapping

The most time consuming loop in SpLU involves access to arrays via an induction variable which is conditionally incremented. Similar patterns involving induction variables occur in DSMC3D. In both of these cases, static analysis of the code reveals conditions which can be tested at run-time to prove that ranges are independent (non-overlapping). Consider the following example abstracted from SpLU DPFAC_DO50:

```
      do 20 i=1,n
         ia_2(i) = ia_1(i)
 20   continue
      do 100 k=1,n
         shift=mod(k,2)*lfact+1
         do 50 j=k+1,n
            c1=shift
            do 60 i=ia_1(j),ia_2(j)
               a(shift)=a(i)
               shift=shift+1
 60         continue
            c2=shift-1
            if (fill-in) then
               c2=c2+1
               a(shift:shift+positive_inc)= ...
               shift=shift+positive_inc
            endif
            do 95 i=ia_2(j)+1,ia_1(j+1)-1
```

```
               a(shift)=a(i)
               shift=shift+1
95          continue
            ia_1(j)=c1
            ia_2(j)=c2
50       continue
         ia(n+1)=shift
100   continue
```

Loop 100 is the outermost loop, and is executed for the n columns in array a. ia_1 and ia_2 are index arrays. $shift$ is an induction variable which is also used as an index into a. Two facts are sufficient to show that the do_50 loop may execute in parallel: one, for each j in do_50, the range of $shift$ must not overlap the range $ia\_1(j), ia\_1(j+1)-1$ for iterations of do_50 executed on other processors[1]; and two, the range of $shift$ must not overlap the same range of $shift$ for iterations executed on other processors[2].

In order to prove these two points we must first determine that ia_1 and ia_2 are non-decreasing. These index arrays are reassigned in each iteration of do_50, thereby complicating the analysis of the access pattern. However, it is possible to determine statically at compile time simple conditions under which these arrays will be non-decreasing. First, note that the induction variable $shift$ is never decremented in the loop. It is conditionally incremented under the "fill-in" condition by a positive amount. Likewise, the initial conditions (loop 20) guarantee that do_60 will execute at least one iteration. Thus, one of the invariant conditions of this loop is that the induction variable $shift$ is strictly increasing. If ia_1 and ia_2 are initially non-decreasing, by induction this invariant condition is sufficient to guarantee that they will remain non-decreasing across the entire execution of the outermost loop 100. Thus, our task of proving that these index arrays are non-decreasing has been reduced to the complexity of executing the test `ia(i+1).ge.ia(i)` for `i = 1,n` once at run-time.

As a side-effect of this analysis we have proven that no output dependences exist across iterations of the do_50 loop. This is a result of the fact that $shift$ is strictly increasing in do_50.

[1]I.e., no flow or anti-dependences are present
[2]I.e., no output dependences

Given that *ia_1* and *ia_2* are non-decreasing, the next step is to show that for each j in do_50, the range of *shift* does not overlap the range $[ia\_1(j), ia\_1(j+1)-1]$ for iterations of do_50 executed on other processors (in effect, we are proving that there are no flow or anti-dependences across iterations of do_50). This can be accomplished using a simple test which compares the upper bound of *shift* to the lower bound of *ia_1* and the lower bound of *shift* to the upper bound of *ia_1*. As before, the presence of a conditional "fill-in" increment to *shift* complicates the analysis. However, we can use an estimate of the maximum value of *shift* by determining an upper bound across the entire iteration space of the do_50 loop. Based on the initial conditions and the strictly increasing nature of *shift*, $ia\_2(j) \geq ia\_1(j)$. Thus we know the tripcount of do_60, and we can conservatively assume that the "fill-in" is always true. Together these facts lead to the following run-time test to confirm the non-overlapping nature of reads and writes to a:

```
min_i=ia(k+1)
max_i=ia(n+1)-1
min_shift=shift
max_shift=shift+(max_i-min_i)+((n-k)*positive_inc)
if(min_shift.gt.max_i .or. max_shift.lt.min_i) then
       parallel=.true.
else
       parallel=.false.
end if
```

This test is placed outside the do_50 loop, and as a result incurs little overhead. When it is true, do_50 may be executed in parallel. When false, it must (conservatively) be executed serially. What this test actually proves is that writes to a are independent of reads to a across all iterations of do_50. This concludes the proof that iterations of do_50 are independent.

3.5 Copy-in and Copy-out

After the identification of monotonicity in index arrays and the proof that induction variables have non-overlapping ranges, it is often necessary to break remaining loop-carried anti and output dependences by *privatizing* both scalar and array variables which are defined and used within a single iteration [24]. In DSMC3D COLLMR_DO100, for example, variables in the /elast/ common block were privatized. Similarly, several variables in DPFAC_DO50 in SpLU required privatization. However, many such variables have initial values which must be copied into each processor's privatized copy of the variable prior to the start of parallel execution. This requires an extension to the existing techniques implemented in Polaris.

Corresponding to this "copy-in", as it is called, is an operation in which variables' values are copied out on the final iteration of each processors' slice of the iteration space. Termed "copy-out", this transformation is necessary whenever local variables have a *last value* which is used outside the parallel region.

3.6 Loops with Conditional Exits

In various cases in our test suite, while loops and loops with multiple exits are used to conditionally construct and manipulate data structures. Such loops present difficulties in parallelization due to side-effects of iterations which are executed in parallel but would not be executed serially. However, certain types of operations such as reductions can be parallelized despite the presence of side-effects. As discussed in Section 3.1, this is possible for associative operations such as histogram reductions if the reduction variable is either privatized or expanded. However, a subtle complication exists for associative operations in loops with conditional exits. This complication has to do with the fact that although the underlying operator may be both associative and commutative (e.g., +), when a conditional exit exists, the execution of the entire *operation* is non-commutative (e.g., a max operation). This is a result of the fact that the exit condition may be true multiple times, but the only correct result is when the condition is true for the first time. Code generation must, as a result, involve the privatization of reduction variable(s) in order to avoid interleaved (commuted) updates to global variable(s).

One implementation difficulty which arises when parallelizing loops with conditional exits is the need for each processor to "flush" its remaining iterations once the exit has been taken. On the SGI Challenge no mechanism is provided to explicitly take an early exit from a parallel loop. In order to provide an early exit, we strip-mine the loop by creating a new, outer loop which executes one iteration on each processor. In the inner loop, iterations are explicitly interleaved so that processors execute relatively small slices of the iteration space. This enables exits to be detected with an efficiency proportional to the size of each slice. A global, shared variable is used to store the minimum iteration in which the break condition is true. Any iteration greater than this minimum will take an early exit out of the loop. The following depicts this transformation:

```
        geti = n+1
        stagesize = blocksize * maxproc
        do 100 k = 1, maxproc
           do j = (k-1)*blocksize+1, n, stagesize
              do i = j, j+blocksize-1
                 if ( i > geti ) then goto 100
                 ...
                 if ( a(i) ) then
                    call lock
                       if ( i < geti ) then geti = i
                    call unlock
                 end if
                 ...
              end do
           end do
 100      end do
```

This transformation involves the following five steps:

1. The iteration space is divided into stages.
2. Each stage is divided into blocks with one block assigned per processor. For simplicity, we assume here that the iteration space can be evenly divided by the blocksize.
3. Each processor goes through all stages; at each stage it executes the iterations in the block assigned to it.
4. Once a processor finds the exit condition true, it sets *geti* atomically to its current iteration. Note that once *geti* is set, it can only be reset to iterations less than *geti* due to the `if (i < geti)` statement.
5. If a processor finds that it is working on an iteration beyond *geti*, it will exit to 100, thereby flushing its remaining iterations.

Since we have confined ourselves to privatizable associative reductions in this paper, we do not need to provide "roll-back" functionality to restore prior state. This greatly reduces the overhead of the transformation.

4 Results

		Polaris	PFA	Manual	Polaris	PFA	Manual
Benchmark	T_{seq}	T_{par}	T_{par}	T_{par}	Speedup	Speedup	Speedup
CHOLESKY	4:25	6:42	4:20	3:40	0.66	1.02	1.20
DSMC3D	8:02	6:35	7:53	1:45	1.22	1.02	4.95
EULER	5:03	2:34	4:56		1.97	1.02	
GCCG	12:19	1:27	1:57		8.49	6.32	
LANCZOS	14:28	2:01	1:58		7.17	7.36	
MVPRODUCT	7:57	1:42	1:11		4.68	6.72	
NBFC	6:15	1:15	6:20		5.00	0.99	
SpLU	3:54	15:25	3:44	1:01	0.25	1.04	3.84

Table 2: Speedups: PFA, Polaris, and Manual

Table 2 presents a comparison of the speedups obtained by Polaris with those of the commercial parallelizing compiler PFA, provided by SGI. The programs were executed in real-time mode on eight processors on an SGI Challenge with 150 MHz R4400 processors. Figure 2 shows that Polaris delivers, in many cases, substantially better speedups than PFA.

The table also portrays additional speedups obtained using new techniques discussed in Section 3. In these cases the techniques were manually implemented and the resulting transformed program executed in parallel.

In general our results indicate that histogram reductions are one of the most important transformations applied in our suite. However, we have determined that there are both cases where the performance of the transformed codes is excellent, and other cases where we obtain relatively poor speedups. The reasons behind this variance in the performance of histogram reductions is under investigation.

Other techniques which proved crucial to the process of parallelization include both sophisticated analysis of index array and induction variable access patterns, and the substitution of pseudo-random number generators.

In the following sections, techniques applied to each benchmark both manually and automatically will be outlined and compared to those applied by PFA.

4.1 NBFC

NBFC contains one computationally key loop which accounts for over 97% of the sequential execution time. Both histogram and single-address reductions occur in the loop. When a given array is involved in both types of reduction statement, it may be parallelized by applying the histogram reduction transformation at all reduction sites involving the array. The histogram reduction technique was sufficient to parallelize this loop, and excellent speedups were obtained. PFA, however, does not implement histogram reductions and therefore achieved no speedup on this benchmark.

4.2 CHOLESKY

The following techniques were applied to CHOLESKY:

- Histogram reductions
- Loops with conditional exits
- Proving monotonicity of index arrays
- Random number generator substitution

Histogram reductions are performed in the main update loop indexed by the variable *kk* in the UPDATE_DO#3. This loop accounts for approximately 20% of the serial execution time. The transformation, however, did not yield significant speedups. This is due to the additional overhead incurred during the initialization and cross-processor reduction phases of the expanded transformation employed. An alternative approach based on the privatizing transformation is under investigation.

Loops with conditional exits occur in GENQMD_DO400, UPDATE_DO#2, and UPDATE_DO#6. The transformation discussed in Section 3.6 was applied to GENQMD_DO400, a loop which performs a reduction across nodes to determine the minimum degree node. The reduction is terminated by a threshold condition which causes an early exit to be taken from the loop. The loop accounts for about 15% of the sequential execution time. Loop-level speedups of 1.6 were achieved on four processors.

The techniques outlined in Section 3.3 apply in the SPARCHOL_DO1015 and SPARCHOL_DO1020 loops in CHOLESKY. Together these loops account for approximately 24% of the serial execution time. A loop-level speedup of 2.84 was obtained in SPARCHOL_DO1020 on four processors.

The final transformation involved the substitution of a parallelized pseudo-random number generator for the library call in SPARCHOL_DO1015. This loop accounts for approximately 3.6% of the serial execution time. The call to the random number generator is the primary work done in the loop, and a loop-level speedup of 2.5 was achieved on eight processors.

Although both Polaris and PFA find a large number of loops parallel in this code, little high-level parallelism is available due to the nature of the supernode algorithm employed. This is reflected in the results for all three versions of the benchmark: Polaris, PFA, and the manually transformed code. We continue to work on this benchmark.

4.3 DSMC3D

The following techniques were applied to DSMC3D:

- Histogram reductions
- Random number generator substitution
- Proving ranges of induction variables non-overlapping
- Associative operations in list manipulation

Histogram reductions, discussed in Section 3.1, are important in several loops: INDEXM_DO300, INDEXM_DO700, COLLMR_DO100, MOVE3_DO#3, and MOVE3_GOTO100. Together these five loops account for approximately 84% of the sequential execution time. Random number generator substitution, discussed in Section 3.2, is important in COLLMR_DO100, MOVE3_GOTO100, INIT3_DO605, and ENTER3_do4. Together these four loops account for almost 60% of the serial execution time. There are two other loops which contain conditionally incremented induction variables, ENTER3_DO4 and INIT3_DO605. Together these loops account for approximately 7.5% of the sequential execution time. Both of these loops are parallelizable using techniques outlined in [19] for determining the closed-form of induction variables if the induction can be proven to be monotonically increasing. However, the conditional increment poses a problem in that monotonicity may not hold and the induction variable ranges may overlap as a result. Through static analysis of the pattern in these loops, a simple run-time test can be abstracted which determines that the induction variable ranges do not overlap and that the loops may be executed in parallel.

4.3.1 Associative operations in list manipulation

DSMC3D contains a while loop in the MOVE3 subroutine which accounts for approximately 35% of the sequential execution time. This loop computes the movement phase, the first of three phases executed each iteration of the outermost time-stepping loop. Molecules involved in the movement phase are stored in lists comprised of two global arrays. These arrays are indexed almost without exception by the loop induction variable. However, when a molecule leaves the flow, it is deleted from the list and replaced by the last molecule in the list. This creates loop-carried dependences in the do loop. However, the deletion of molecules can be deferred until after the entire list has been processed [26, 6]. Based on this, the following transformation can be made:

```
i = 1
j = n
while (i < j)                  parallel loop i = 1, n
    if (cond(a(i)) then          if (cond(a(i)) then
        a(i) = a(j)                a(i) = mark
        j = j - 1        ⇒       endif
    else                       enddo
        i = i + 1              call remove_marked(a)
    endif
endwhile
```

The current molecule in $a(i)$ is marked for later removal. After exit from the parallel loop, marked elements are removed and the array a is packed. Effectively, the operation of removing and replacing elements in the list is associative, and therefore can be parallelized [25].

The combination of these techniques in the loops mentioned above contributed to the overall program speedup of 4.95 in the manually parallelized version. The speedups reported for Polaris include the histogram reduction in MOVE3_DO#3. PFA, however, does not implement any of these techniques and therefore achieved less of a speedup than Polaris, although both were low.

4.4 EULER

EULER contains five computationally important loops: DFLUX_DO100, DFLUX_DO200, EFLUX_DO100, EFLUX_DO200, and PSMOO_DO20. Together these loops account for over 70% of the serial execution time of the program. In all five loops, the histogram reduction transformation is the only transformation necessary to parallelize the loop. Although the speedup for this benchmark is approximately two, actual loop-level speedups of the transformed reduction loops are quite good (e.g., 3 on 4 processors).

Two factors contributed to the lack of overall program speedup: first, all transformations involved the use of expanded reductions because, based on our study of other codes in the Perfect and SPEC benchmark suites, we have empirically determined that expansion is the most efficient transformation on the SGI. However, no appreciable speedups could be obtained in the initialization and cross-processor reduction loops. Secondly, the transformed reduction loops incurred an overhead of between 10% and 30% in execution time. Together these factors reduced the overall program speedup by a factor of approximately two.

In investigating the reasons behind these results, profiling tools available on the SGI Power Challenge were employed[3]. Based on hardware counters integrated into the R10000 CPU, events such as cache misses, translation lookaside buffer (TLB) misses, cycle counts, etc. can be statistically sampled during program execution. Initial study of the results of profiling experiments has revealed that primary cache misses, secondary cache misses, and TLB misses all increased. Further evaluation of these results is underway.

Despite the difficulties encountered with the implementation of histogram reductions in EULER, the transformation resulted in an overall program speedup of two. No speedup was achieved with PFA due to the fact that PFA does not recognize and solve histogram reductions. In comparison, Polaris did somewhat, but only slightly, better.

[3]Note that our timing experiments were conducted on the SGI Challenge

4.5 GCCG

The primary access pattern in GCCG involves indirections which occur on the right-hand-sides of assignment statements, as discussed in Section 2. These pose no particular dependence problem due to the fact that the array locations are read but not written. Many reductions occur in GCCG, but they are all scalar or *single-address* reductions in which the reduction variable is a single element of an array. Current parallelizing technology is capable of recognizing and transforming such reductions into parallel form. This fact is reflected in the speedup results for PFA as well as Polaris.

4.6 LANCZOS

LANCZOS presents a situation similar to that found in GCCG in that the primary access pattern involves indirection on assignment statements' right-hand-sides during a sparse matrix-vector product operation. The reorthogonalization is computed using dense matrices, and arrays are accessed via loop indices. As a result, no loop-carried dependences prevent parallelization. This fact is reflected in the good speedups achieved by both Polaris and PFA.

4.7 MVPRODUCT

MVPRODUCT has been implemented such that dense matrices result from the combination of sparse matrices. Due to this fact, indirection arises only on the right-hand-sides of assignment statements. This type of indirection poses no particular problem to parallelization, and the speedups achieved by both PFA and Polaris reflect this fact.

4.8 SpLU

The parallelization of SpLU involved the following techniques:

- Proving monotonicity of index arrays
- Proving ranges of induction variables non-overlapping
- Copy-in and Copy-out

Although a loop with a conditional exit exists in the subroutine DPREORD, this subroutine was not called during our experiments. Loop DPFAC_DO50 in SpLU accounts for almost 100% of the serial execution time of the benchmark. As discussed in Section 3.3, it was first necessary to prove that index arrays were non-decreasing, and from this fact it was possible to develop a run-time test capable of proving the independence of induction variable ranges. The final transformation applied in DPFAC_DO50 involved the copy-in and copy-out of the privatized arrays $a, r, cptr1$ and $cptr2$.

Neither PFA nor Polaris implement the functionality present in these three techniques, and the corresponding speedups reflect this fact. Polaris, in particular, applied the histogram reduction transformation to an inner loop with a low tripcount. This resulted in a significant slowdown.

5 Conclusions

In our study of our sparse and irregular benchmark suite we have determined that indirection on the right-hand-sides of assignment statements is not a hindrance to automatic parallelization. We have also identified several new techniques which begin to point to the fact that, although much work remains to be done, automatic parallelization of sparse and irregular codes seems feasible.

References

[1] Zahira Ammarguellat and Luddy Harrison. Automatic Recognition of Induction & Recurrence Relations by Abstract Interpretation. *Proceedings of Sigplan 1990, Yorktown Heights*, 25(6):283–295, June 1990.

[2] R. Asenjo, M. Ujaldón, and E. L. Zapata. *SpLU – Sparse LU Factorization. HPF-2. Scope of Activities and Motivating Applications.* High Performance Fortran Forum, version 0.8 edition, November 1994.

[3] Richard Barrett, Michael Berry, Tony F. Chan, James Demmel, June Donato, Jack Dongarra, Victor Eijkhout, Roldan Pozo, Charles Romine, and Henk van der Vorst. *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods.* SIAM, Philadelphia, PA, 1994.

[4] M. Berry, D. Chen, P. Koss, D. Kuck, L. Pointer, S. Lo, Y. Pang, R. Roloff, A. Sameh, E. Clementi, S. Chin, D. Schneider, G. Fox, P. Messina, D. Walker, C. Hsiung, J. Schwarzmeier, K. Lue, S. Orszag, F. Seidl, O. Johnson, G. Swanson, R. Goodrum, and J. Martin. The Perfect Club Benchmarks: Effective Performance Evalution of Supercomputers. *Int'l. Journal of Supercomputer Applications, Fall 1989*, 3(3):5–40, Fall 1989.

[5] G.A. Bird. *Molecular Gas Dynamics and the Direct Simulation of Gas Flows.* Oxford University Press, Oxford, England, 1994.

[6] Graeme Bird. Personal communication with author, 1996.

[7] William Blume, Ramon Doallo, Rudolf Eigenmann, John Grout, Jay Hoeflinger, Thomas Lawrence, Jaejin Lee, David Padua, Yunheung Paek, Bill Pottenger, Lawrence Rauchwerger, and Peng Tu. Parallel Programming with Polaris. *IEEE Computer*, 29(12):78–82, December 1996.

[8] B.R. Brooks, R.E. Bruccoleri, B.D. Olafson, D.J. States, S. Swaminathan, and M. Karplus. CHARMM: A Program for Macromolecular Energy, Minimization, and Dynamics Calculations. *J. Comp. Chem.*, 4:187–217, 1983.

[9] Luiz DeRose, Kyle Gallivan, Bret Marsolf, David Padua, and Stratis Gallopoulos. FALCON: A MATLAB Interactive Restructuring Compiler. *Proceedings of the 8th International Workshop on Languages and Compilers for Parallel Computing, Columbus, OH*, pages 18.1–18.18, August 1995.

[10] Iain Duff, Nick Gould, John Reid, Jennifer Scott, and Linda Miles. Harwell Subroutine Library. Technical Report http://www.rl.ac.uk/departments/ccd/numerical/hsl/hsl.html, Council for the Central Laboratory of the Research Councils, Department for Computation and Information, Advanced Research Computing Division.

[11] Ian Foster, Rob Schreiber, and Paul Havlak. HPF-2 Scope of Activities and Motivating Applications. Technical Report CRPC-TR94492, Rice University, November 1994.

[12] G.H. Golub and C.F. van Loan. *Matrix Computations.* The Johns Hopkins University Press, 1993.

[13] IBM. Parallel FORTRAN Language and Library Reference, March 1988.

[14] Jee Myeong Ku. The Design of an Efficient and Portable Interface Between a Parallelizing Compiler and its Target Machine. Master's thesis, Univ. of Illinois at Urbana-Champaign, Center for Supercomputing Res. & Dev., 1995.

[15] D.H. Lehmer. Mathematical Methods in Large-scale Computing Units. In *2nd Symposium on Large-Scale Digital Calculating Machinery*, pages 141–146, Cambridge, MA, 1951. Harvard University Press.

[16] P.A.W. Lewis, A.S. Goodman, and J.M. Miller. A Pseudo-Random Number Generator for the System/360. *IBM Systems Journal*, 8(2):136–146, May 1969.

[17] G. Lueker. Some Techniques for Solving Recurrences. *Computing Surveys, Vol. 12, No. 4*, December 1980.

[18] Michael Mascagni and David Bailey. Requirements for a Parallel Pseudorandom Number Generator. Technical Report http://olympic.jpl.nasa.gov/SSTWG/lolevel.msgs.html, Center for Computing Sciences, I.D.A. and NASA Ames Research Center, June 1995.

[19] Bill Pottenger and Rudolf Eigenmann. Idiom Recognition in the Polaris Parallelizing Compiler. *Proceedings of the 9th ACM International Conference on Supercomputing, Barcelona, Spain*, pages 444–448, July 1995.

[20] Daniel V. Pryor, Steven A. Cuccaro, Michael Mascagni, and M. L. Robinson. Implementation of a Portable and Reproducible Parallel Pseudorandom Number Generator. *Proceedings of Supercomputing '94*, Nov. 1994.

[21] Lawrence Rauchwerger and David Padua. The LRPD Test: Speculative Run-Time Parallelization of Loops with Privatization and Reduction Parallelization. *Proceedings of the SIGPLAN'95 Conference on Programming Language Design and Implementation*, June 1995.

[22] L.F. Romero and E.L. Zapata. Data Distributions for Sparse Matrix Vector Multiplication. *J. Parallel Computing*, 21(4):583–605, April 1995.

[23] Gary Sabot. *The Paralation Model.* The MIT Press, Cambridge, Massachusetts, 1988.

[24] Peng Tu and David Padua. Automatic Array Privatization. In Utpal Banerjee, David Gelernter, Alex Nicolau, and David Padua, editors, *Proc. Sixth Workshop on Languages and Compilers for Parallel Computing, Portland, OR. Lecture Notes in Computer Science.*, volume 768, pages 500–521, August 12-14, 1993.

[25] Guhan Viswanathan and James R. Larus. User-defined Reductions for Efficient Communication in Data-Parallel Languages. Technical Report 1293, Univ. of Wisconsin-Madison, Computer Sciences Department, Aug. 1996.

[26] Dick Wilmoth. Personal communication with author, 1996.