Parallelism in Loops Containing Recurrences

Bill Pottenger

June 12, 1996

1 Introduction

Extensive analysis of Grand Challenge codes from NCSA, codes from the SPEC CFP95 Benchmark Suite, and codes from the Perfect Club Suite, has revealed the presence of a number of loops in which recurrences prevent DOALL parallelization.

During a recent review of the Polaris restructurer [BDE+96] approximately 500 loops from programs in the aforementioned test suite were identified as serial. Of these, approximately 35% involve an explicitly coded recurrence, reduction, or induction for which Polaris was unable to determine a parallel form. Approximately 70% of the loops in this 35% subset were determined to be partially or fully parallelizable based on a manual inspection of the codes.

As part of our research in the automatic parallelization of FORTRAN programs, techniques are being developed for solving these recurrences in DOALL and DOACROSS fashions. In the following section, several techniques will be presented which have proven effective in the parallelization of loops containing recurrences.

2 Techniques for Parallelizing Recurrences

2.1 Symbolic Computation of Closed Form

[Pot94] includes a survey of symbolic techniques for determining closed forms for recurrences involving scalar induction variables. The extension of similar symbolic techniques to the solution of linear recurrences is under investigation.

2.2 Hoisting

There are at least two cases in the SPEC CFP95 benchmark su2cor where recurrences occur in an explicit form. An example from the routine trngv is portrayed below.

```
IFIRST=0
IMAX=2147483647
DO 100 N=1,N103
    DO 10 I=IFIRST,IFIRST+102
        IREG(I+250)=IREG(I+147)-IREG(I)
        IF (IREG(I+250).LE.0) IREG(I+250)=IREG(I+250)+IMAX
    10 CONTINUE
    IFIRST=IFIRST+103
100 CONTINUE
```

Trngv is called from within several of the major serial loops, including SU2COR_do60 and SWEEP_do200. This section of code has an explicit recurrence relation of the form $a_i = a_{i-103} +$

$a_{i-250} \; mod \; 2^{31} - 1$. It is an implementation of a lagged-Fibonacci pseudorandom number generator with recursion parameters of $(103, 250)$ [PCMR94]. The conditional subtraction performs a modulo operation which results in an integer in the range $[0, 2^{31} - 2]$, a member of the *representative residue class* $[0, 2^{31} - 2]$.

One approach to solving linear homogeneous recurrences of this nature is to symbolically calculate the closed form of the recurrence during compilation as was discussed briefly in the previous section. However, although the techniques for solving this class of recurrence are well known, in this case the actual closed form is numerically unstable due to the fact that the computation must be done in the complex realm. Nonetheless, there are many cases where symbolic techniques such as this have been successfully employed [PE95].

For this particular recurrence, partial *hoisting* was employed to break the dependence arcs in two of the enclosing loops. Simply put, the computation of the pseudorandom numbers was hoisted out of inner loops and placed in an outer loop. Naturally this transformation required that additional elements (pseudorandom numbers) be computed and stored for later use in the inner loops. This was accomplished by extracting the loop-control flow from SWEEP_do200/2 (a perfectly nested loop inside SWEEP_do200/1) and pre-computing (in do200/1) the set of pseudorandom numbers that would later be needed inside do200/2.

However, since our target for DOACROSS parallelization was SWEEP_do200/1, there remained unbroken dependence arcs in this, the outermost loop. These were handled by explicitly synchronizing access to *ireg*.

2.3 Overlapping Loop Execution using Semi-private Variables

Another interesting transformation involved what we have termed *semi-private* variables. The privatization of variables often allows dependence arcs to be broken [Tu95]. In SU2COR_do60 a pattern occurs in which a variable is private from a certain program point in a loop to the end of the loop. Consider the following example:

```
DO K=1,N
    DO I=2,N
        DO J=1,N
            ...
            ... = ... A(J,I-1) ...
            ...
            A(J,I) = ...
            ...
        ENDDO
    ENDDO
    ...
    DO M=2,N
        DO J=1,N
            ... = ... A(J,M) ...
        ENDDO
    ENDDO
ENDDO
```

Here we have a case where the writes to the variable A in the I loop completely cover the reads to A in the M loop. Since A is read-only in the M loop, a private copy of A can be made upon termination of the I loop, allowing the M loop to proceed in tandem with a subsequent invocation of the I loop. A similar pattern occurs in SU2COR_do60 where the I and M loops are actually the outermost loops of subroutines called from within do60.

2.4 Doacross Parallelism

The loop SWEEP_do200 in su2cor is a candidate for doacross parallelism for two reasons: one, there is work done at the head of the loop which does not involve accesses to variables with loop-carried dependences; and two, since the loop-carried dependences present in this loop are conditional in nature, it is possible to take advantage of the case(s) where no actual dependences exist.

A doacross loop was implemented for do200/1 using synchronization to enforce the dependence relationships on sections of the main data structure u. The issue of granularity of synchronization did arise, but will not be discussed here due to space limitations. Suffice it to say that the optimum granularity of synchronization was on a panel of u made up of 4096 32-byte "elements", and the doacross loop do200/1 accesses four of these panels every iteration.

2.5 Structure Access Pipelining

In order to gain additional parallelism in SWEEP_do200/1, a technique termed *structure access pipelining* was employed. Akin to vector chaining, structure access pipelining overlaps contiguous invocations of a parallel loop. For example, consider the following loop:

```
DO K=1,N
    CALL EXAMPLE
ENDDO
SUBROUTINE EXAMPLE
    DO I=2,N
        DO J=1,N
            ...
            ... = ... A(J,I-1) ...
            ...
            A(J,I) = ...
            ...
        ENDDO
    ENDDO
END
```

The loop-carried dependences on A serialize the outer I loop in EXAMPLE. However, using synchronization this loop can be executed as a doacross loop in parallel. Additional parallelism can be obtained across invocations of the outer I loop (i.e., across subroutine calls) when the control flow in the caller requires that EXAMPLE be reinvoked (as shown above). This can be accomplished, for example, by coalescing the K loop in the caller with the I loop in EXAMPLE and running this new, coalesced loop as a doacross.

This transformation accomplishes two things: one, it mitigates the overhead of spawning a parallel loop each time EXAMPLE is called; and two, like vector chaining, it hides the latency of access to A by allowing processors to proceed with the next (logical) iteration of the caller's loop without waiting for all other processors to finish the previous iteration. This is precisely the situation which arises in SU2COR_do60, the caller of SWEEP which invokes the loop SWEEP_do200.

2.6 Loop Rotation

In the CFP96 code applu several loop-carried dependences exist in the main time-stepping loop in subroutine SSOR. However, through a technique dubbed *loop rotation*, the loop-carried dependences on the array variable *rsd* can be broken via privatization. The technique is akin to a partial peeling operation which involves rotating the body of the loop in order to move the write of *rsd* to the head of the loop, thereby allowing it to be privatized. The portion of the first iteration which was

partially peeled is duplicated in a prologue to the loop, and the corresponding portion (partially peeled from the final iteration) is likewise duplicated in an epilogue.

2.7 Min/Max Reductions

An interesting recurrence occurs in the SPEC CFP95 benchmark tomcatv in the loop MAIN_do80. The following portrays this loop:

```
DO 80 J = 2,N-1
    DO 80 I = 2,N-1
        RXM(ITER) = MAX(RXM(ITER), ABS(RX(I,J)))
        RYM(ITER) = MAX(RYM(ITER), ABS(RY(I,J)))
80 CONTINUE
```

As can be seen from this example, do80 is a max() reduction on two variables, rx and ry. $ITER$ is the index of the outermost timestepping loop (MAIN_do140).

Polaris has the ability to recognize and transform additive reductions into a parallel form. However, the current reduction recognition pass does not recognize max or min reductions. Nonetheless, the transformation is straightforward, and is pictured below based on the technique employing *expanded reductions* [PE95]:

```
do j = 1, procs
    rxm_e(j) = 0.0
    rym_e(j) = 0.0
enddo
do parallel k = 2, n-1
    do i = 2, n-1
        rxm_e(thread-num()) = max(rxm_e(thread-num()),abs(rx(i, k)))
        rym_e(thread-num()) = max(rym_e(thread-num()),abs(ry(i, k)))
    enddo
enddo
do m = 1, procs
    rxm(iter) = max(rxm(iter),rxm_e(m))
    rym(iter) = max(rym(iter),rym_e(m))
enddo
```

In the above, rxm_e and rym_e are expanded versions of the original *single address* reduction variables rxm(iter) and rym(iter)[1]. They are expanded by the number of concurrent threads, initialized to zero, and indexed by the thread-id of each thread participating in the computation. Following the parallel reduction in the k loop, the partial results are summed across threads in the final m loop.

2.8 Loop Chaining

The structure access pipelining discussed in section 2.5 has a corresponding application in tomcatv which we term *loop chaining*. Consider the following example abstracted from tomcatv:

```
DO 140 ITER=1,ITACT
```

[1] Note that the index iter is invariant in the do80 loop nest

```
    ...
    DO 60 J=2,N-1
        DO 50 I=2,N-1
            ...
            RX(I,J) = A*PXX+B*PYY-C*PXY
            RY(I,J) = A*QXX+B*QYY-C*QXY
        50 CONTINUE
    60 CONTINUE
    DO 80 J=2,N-1
        DO 80 I=2,N-1
            RXM(ITER) = MAX(RXM(ITER), ABS(RX(I,J)))
            RYM(ITER) = MAX(RYM(ITER), ABS(RY(I,J)))
    80 CONTINUE
    ...
140 CONTINUE
```

Both the do60 and do80 loops access rx and ry in their entirety during each step of the outermost loop 140 (index ITER). The dependences on rxm and rym in do140 are not loop-carried - i.e., rxm and rym are loop-private variables which are written in do80 and read prior to the exit of do140.

As a result, it is possible to overlap the access to rx and ry in do60 and do80 such that these two loops can be executed concurrently. This is accomplished by synchronizing on columns of rx and ry as pictured below:

```
DO 140 ITER=1,ITACT
    ...
process_1
    DO parallel 60 J=2,N-1
        DO 50 I=2,N-1
            ...
            RX(I,J) = A*PXX+B*PYY-C*PXY
            RX_p(I,J) = RX(I,J)
            RY(I,J) = A*QXX+B*QYY-C*QXY
            RY_p(I,J) = RY(I,J)
        50 CONTINUE
        post(j)
    60 CONTINUE
process_2
    DO 80 J=2,N-1
        wait(j)
        DO 80 I=2,N-1
            RXM(ITER) = MAX(RXM(ITER), ABS(RX_p(I,J)))
            RYM(ITER) = MAX(RYM(ITER), ABS(RY_p(I,J)))
    80 CONTINUE
    ...
140 CONTINUE
```

In this example, loops do60 and do80 are executed as two separate processes which share the same address space. Loop do60 is a parallel, doall loop, and do80 may be executed either as a parallel reduction loop or as a serial loop depending on the ratio of the respective execution times for the loops. The term *loop chaining* is derived from the nature of the access pattern of the two loops: do60 executes one iteration (writing one entire column of both RX_p and RY_p), then posts to notify do80 that these private variables are available. Loop do80 waits for the post then accesses

the same columns of RX_p and RY_p in turn, with the result that access to the Jth columns of rx and ry are "chained". In much the same way that vector chaining enhances performance in a vector processor, loop chaining hides the execution time of the do80 loop in its entirety.

3 Results to Date

In this section performance results will be discussed for two benchmarks from the SPEC CFP95 suite which have been studied and manually transformed. The discussion will be organized on a loop-by-loop basis for computationally important loops in these two codes.

3.1 su2cor

The computationally key loops in su2cor are delineated in Table 1.

Subroutine	Loop	Depth	% T_{seq}	S/P
SU2COR	do60	1	99.9809	S
SWEEP	do200	2	63.1459	S
LOOPS	do900	2	36.8664	S
SWEEP	do220	4	34.3754	P
LOOPS	do400	3	29.6517	P
MATMAT	do10	5	18.3314	P
SWEEP	do310	4	16.1173	S
INT2V	do100	6	13.9536	S
SWEEP	do311	5	12.7783	P

Table 1: Loops > 10% of Sequential Time

The S/P markings in Table 1 denote whether the given loop is parallel (P) or serial (S). The Depth column designates the (interprocedural) nesting level of the given loop, and % T_{seq} is the percentage of the sequential execution time for the loop (including nested loops).

As these figures indicate, many time-consuming outer loops are serial in nature. Our task has been to determine if additional loop-level parallelism is of importance in these loops.

3.1.1 Results

The implementation of the four techniques discussed above in Sections 2.2, 2.3, 2.4, and 2.5 required sophisticated synchronization, including the implementation of a non-blocking barrier to allow structure access pipelining. Two additional transformations were necessary as well: the solution of reductions in parallel and the solution of induction variables with discontinuous closed forms.

The combination of these six transformations yielded an overall program speedup of 3.63 using between 10 and 12 processors. Runs were made on a 12-processor SGI Challenge in a real-time scheduling mode.

3.2 tomcatv

The computationally key loops in tomcatv are delineated in Table 2.

The S/P markings in Table 2 denote whether the given loop is parallel (P) or serial (S). The Depth column designates the (interprocedural) nesting level of the given loop, and % T_{seq} is the percentage of the sequential execution time for the loop (including nested loops). As these figures indicate, many of the more time-consuming loops in tomcatv are serial in nature.

Subroutine	Loop	Depth	% T_{seq}	S/P
MAIN	do140	1	98.4012	S
MAIN	do60	2	64.5203	P
MAIN	do100	2	12.6117	S
MAIN	do120	2	8.67711	S
MAIN	do130	2	7.24051	P
MAIN	do80	2	5.32165	S

Table 2: Loops > 5% of Sequential Time

3.2.1 Results

The initial solution for parallelizing the reduction operations in do80 was straightforward. However, the resulting performance was very poor due to false-sharing of reduction variables between threads running on different processors. A second version solved this problem by making each element in the expanded reduction variables rxm_e and rym_e the size of a cacheline[2]. The performance of this version was quite good, with a speedup of over 7.2 for the loop, and an overall program speedup of 4.29. Table 3 summarizes these results.

Performance results for loop chaining vs. privatized reductions also appear in Table 3. The speedup for this technique actually exceeds that achieved for the parallel reductions. This result is partially due to better cache utilization in the loop-chaining version.

Transformation	speedup
Polaris	3.49
Polaris (with do80 reduction in parallel)	4.29
Polaris (with loop chaining transformation)	4.96

Table 3: Tomcatv speedups on 8 processors

4 Conclusions

The results for su2cor indicate that significant additional parallelism beyond doall parallelism is available. A project is underway to determine if the newly uncovered parallelism can be effectively combined with the existing doall parallelism on an architecture which supports multi-level parallelism.

The results for tomcatv are promising in that the *loop chaining* technique has proven to increase overall program speedup by a factor of 1.4. It remains to be investigated how often this pattern can be effectively taken advantage of in Fortran programs.

References

[BDE+96] William Blume, Ramon Doallo, Rudolf Eigenmann, John Grout, Jay Hoeflinger, Thomas Lawrence, Jaejin Lee, David Padua, Yunheung Paek, William Pottenger, Lawrence Rauchwerger, and Peng Tu. *Advanced Program Restructuring for High-Performance*

[2] 128 bytes on the Challenge

Computers with Polaris. Univ. of Illinois at Urbana-Champaign, Center for Supercomputing Res. & Dev., January 1996.

[PCMR94] D. Pryor, S. Cuccaro, M. Mascagni, and M. Robinson. Implementation of a Portable and Reproducible Parallel Pseudorandom Number Generator. *IEEE*, 1994.

[PE95] Bill Pottenger and Rudolf Eigenmann. Idiom Recognition in the Polaris Parallelizing Compiler. *Proceedings of the 9th ACM International Conference on Supercomputing, Barcelona, Spain*, July 1995.

[Pot94] William Morton Pottenger. Induction Variable Substitution and Reduction Recognition in the Polaris Parallelizing Compiler. Master's thesis, Univ of Illinois at Urbana-Champaign, Cntr for Supercomputing Res & Dev, December 1994.

[Tu95] Peng Tu. *Automatic Array Privatization and Demand-Driven Symbolic Analysis*. PhD thesis, Univ. of Illinois at Urbana-Champaign, Center for Supercomputing Res. & Dev., May 1995.