Array Regions for Interprocedural Parallelization and Array Privatization

Béatrice CREUSILLET*

Centre de Recherche en Informatique
École des mines de Paris
Internal Report A/279/CRI

Abstract

Three demonstrations are presented, that highlight the need for interprocedural analyses such as preconditions *and exact* array regions*, in order to parallelize loops that contain subroutine calls or temporary arrays. These analyses are provided by PIPS in an interactive environment.*

1 Interprocedural Parallelization

`AILE` is an application from the ONERA, the French institute of aerospatial research. It has more than 3000 lines of FORTRAN code. It has been slightly modified to test the coherence of some input values.

The aim of this demonstration is to show that interprocedural analyses are necessary for an automatic parallelization.

For that purpose, we have chosen the subroutine (or *module*) `EXTR`, which is called by the module `GEOM`, itself called by the main routine `AILE`. An excerpt is given in Figure 1 (without the intermediate call to `GEOM`).

1.1 EXTR

`EXTR` contains a `DO` loop that has several characteristics:

1. There are several read and write references to elements of the array `T`. This induces dependences that cannot be disproved if we don't know the relations between index expressions, and more precisely between `J` and `JH`. We already know that `JH=J1+J2-J`, but we don't know the values of `J1`, `J2` and `JA`, which are global variables initialized in `AILE`. Thus, we can disprove the loop-carried dependences between `T(J,1,NC+3)` and `T(JH,1,NC+3)` for instance, only if we interprocedurally propagate the values of `J1`, `J2` and `JA` from `AILE`. This type of information is called *precondition* in PIPS [4, 3].

*e-mail: <creusil@cri.ensmp.fr>, web: <http://www.cri.ensmp.fr/~creusil>

```
      PROGRAM AILE
      DIMENSION T(52,21,60)
      COMMON/CT/T
      COMMON/CI/I1,I2,IMAX,I1P1,I1P2,I2M1,I2M2,IBF
      COMMON/CJ/J1,J2,JMAX,J1P1,J1P2,J2M1,J2M2,JA,JB,JAM1,JBP1
      COMMON/CK/K1,K2,KMAX,K1P1,K1P2,K2M1,K2M2
      COMMON/CNI/L
      ...
      READ(NXYZ) I1,I2,J1,JA,K1,K2
C
      IF(J1.GE.1.AND.K1.GE.1) THEN
         N4=4
         J1=J1+1
         J2=2*JA+1
         JA=JA+1
         K1=K1+1
         ...
         CALL EXTR(NI,NC)
      ENDIF
      END

      SUBROUTINE EXTR(NI,NC)
      DIMENSION T(52,21,60)
      COMMON/CT/T
      COMMON/CI/I1,I2,IMAX,I1P1,I1P2,I2M1,I2M2,IBF
      COMMON/CJ/J1,J2,JMAX,J1P1,J1P2,J2M1,J2M2,JA,JB,JAM1,JBP1
      COMMON/CK/K1,K2,KMAX,K1P1,K1P2,K2M1,K2M2
      COMMON/CNI/L
      L=NI
      K=K1
      DO 300 J=J1,JA
         S1=D(J,K  ,J,K+1)
         S2=D(J,K+1,J,K+2)+S1
         S3=D(J,K+2,J,K+3)+S2
         T(J,1,NC+3)=S2*S3/((S1-S2)*(S1-S3))
         T(J,1,NC+4)=S3*S1/((S2-S3)*(S2-S1))
         T(J,1,NC+5)=S1*S2/((S3-S1)*(S3-S2))
         JH=J1+J2-J
         T(JH,1,NC+3)=T(J,1,NC+3)
         T(JH,1,NC+4)=T(J,1,NC+4)
         T(JH,1,NC+5)=T(J,1,NC+5)
 300  CONTINUE
      END

      REAL FUNCTION D(J,K,JP,KP)
      DIMENSION T(52,21,60)
      COMMON/CT/T
      COMMON/CNI/L
C
      D=SQRT((T(J,K,L  )-T(JP,KP,L  ))**2
     1     +(T(J,K,L+1)-T(JP,KP,L+1))**2
     2     +(T(J,K,L+2)-T(JP,KP,L+2))**2)
      END
```

Figure 1: Excerpt from program AILE.

2. There are three calls to the function D in EXTR. D contains several read references to the global array T. So, we must assume that the whole array is potentially read by each call to D. This induces dependences in EXTR between the calls to D and the other statements. In order to disprove these dependences, we need a way to represent the set of array elements read by any invocation of D, and be able to use this information at each call site. These sets are called *array regions* in PIPS [2].

3. S1, S2, S3 and JH are defined and used at each iteration. This induces loop-carried dependences. But we may notice that each use is preceded by a definition in the same iteration. These variables can be privatized (this means that a local copy is assigned to each iteration) to remove the spurious dependences.

1.2 D

As written before, there are several references to elements of the array T in D. Our aim is to represent this set of elements, such that it can be used at each call site to help disproving dependences.

If we know nothing about the relations between the values of K and KP or between J and JP, all we can deduce is that the third index of all the array elements ranges between L and L+2. This is represented by the region:

$$\texttt{<T(}\phi_1\texttt{,}\phi_2\texttt{,}\phi_3\texttt{)-R-MAY-}\{\texttt{L<=}\phi_3\texttt{<=L+2}\}\texttt{>}$$

The ϕ variables represent the dimensions of the array; R means that we consider the *read* effects on the variable; and MAY means that the region is an over-approximation of the set of elements that are actually read.

The relations between the values of K and KP or J and JP are those that exist between the real arguments. At each call site, we have JP==J and KP==K+1. These contidions hold true before each execution of D; we call them *preconditions*. Under these conditions, we can now recompute the region associated to the array T:

$$\texttt{<T(}\phi_1\texttt{,}\phi_2\texttt{,}\phi_3\texttt{)-R-MUST-}\{\phi_1\texttt{==J, K<=}\phi_2\texttt{<=K+1, L<=}\phi_3\texttt{<=L+2}\}\texttt{>}$$

Notice that this is a MUST region, because it exactly represents the set of array elements read by any invocation of function D.

1.3 Parallelisation of EXTR

We can now parallelize EXTR by:

1. privatizing the scalar variables;

2. using *array regions* to summarize the read effects on the array T by each invocation of D;

3. using the *preconditions* induced by the initialization of global scalar variables (in AILE) to disprove the remaining dependences.

This leads to the parallelized version of Figure 2.

```
SUBROUTINE EXTR(NI,NC)
DIMENSION T(52,21,60)
COMMON/CT/T
COMMON/CI/I1,I2,IMAX,I1P1,I1P2,I2M1,I2M2,IBF
COMMON/CJ/J1,J2,JMAX,J1P1,J1P2,J2M1,J2M2,JA,JB,JAM1,JBP1
COMMON/CK/K1,K2,KMAX,K1P1,K1P2,K2M1,K2M2
COMMON/CNI/L
L = NI
K = K1
DOALL J = J1, JA
   PRIVATE S1,S2,S3
   S1 = D(J, K, J, K+1)
   S2 = D(J, K+1, J, K+2)+S1
   S3 = D(J, K+2, J, K+3)+S2
   T(J,1,NC+3) = S2*S3/((S1-S2)*(S1-S3))
   T(J,1,NC+4) = S3*S1/((S2-S3)*(S2-S1))
   T(J,1,NC+5) = S1*S2/((S3-S1)*(S3-S2))
ENDDO
DOALL J = J1, JA
   PRIVATE JH
   JH = J1+J2-J
   T(JH,1,NC+3) = T(J,1,NC+3)
   T(JH,1,NC+4) = T(J,1,NC+4)
   T(JH,1,NC+5) = T(J,1,NC+5)
ENDDO
END
```

Figure 2: Parallelized version of EXTR.

2 Array Privatization

Array privatization is not yet implemented in PIPS, but the information needed to perform the transformation is already available: *IN and OUT regions* [2, 1].

To illustrate the characteritics of these regions, we will consider two examples: `NORM` is another excerpt from `AILE`, and `RENPAR6` is a contrived example that highlights some details of the computation of regions and the possibilities opened up by IN and OUT regions.

2.1 NORM

This is a very simple example (see Figure 3) that shows the necessity of array privatization, and the need for IN and OUT array regions.

In the loop of subroutine `NORM`, the references to the array `T` do not induce loop-carried dependences. Furthermore, there are only read-read dependences on `S`. However, notice that the array `TI` is a real argument in the call to `PVNMUT`, and that there are 3 read references to array `TI`. This induces potential interprocedural dependences. We have seen with the previous example that these dependences can sometimes be disproved with array regions.

We must first compute the regions of array `TI` that are referenced in `PVNMUT`. In `PVNMUT`, `TI` is called `C`. And the 3 elements of `C` are written, but not read. This leads to:

$$\mathtt{<C(}\phi_1\mathtt{)\text{-}W\text{-}MUST\text{-}}\{\mathtt{1<=}\phi_1\mathtt{<=3}\}\mathtt{>}$$

(`W` means that this is a *write* effect)

At the call site, `C` is translated into `TI`, which gives the region:

$$\mathtt{<TI(}\phi_1\mathtt{)\text{-}W\text{-}MUST\text{-}}\{\mathtt{1<=}\phi_1\mathtt{<=3}\}\mathtt{>}$$

And finally, the regions corresponding to the whole body of the loop nest are:

$$\mathtt{<TI(}\phi_1\mathtt{)\text{-}W\text{-}MUST\text{-}}\{\mathtt{1<=}\phi_1\mathtt{<=3}\}\mathtt{>}$$
$$\mathtt{<TI(}\phi_1\mathtt{)\text{-}R\text{-}MUST\text{-}}\{\mathtt{1<=}\phi_1\mathtt{<=3}\}\mathtt{>}$$

These regions are identical, which means that each iteration of loops `K` and `J` reads and writes to the same memory locations of array `TI`. Thus, there are loop-carried dependences, and the loop cannot be parallelized.

However, these dependences are false dependences, because if we allocate a copy of array `TI` to each iteration (in fact to each processor), there are no more dependences. This is what is called array privatization. In order to privatize an array, we must be sure that, in each iteration, no element is read before being written in the same iteration. Thus, there are no loop-carried producer-consumer dependences.

This last property cannot be verified by using READ regions, because they contain all the elements that are read, and not only those that are read before being written. This is represented in PIPS by IN regions. In our case, we must verify that no element of `TI` belongs to the IN region corresponding to the loop body, which is the case.

We must also be sure that no element of `TI` that is initialized by a single iteration is used in the subsequent iterations or after the loops. This information is provided in PIPS by the OUT regions. They represent the set of live array elements, that is to say those that are used in the continuation.

We can now parallelize `NORM` by:

```
      PROGRAM AILE
      DIMENSION T(52,21,60)
      COMMON/CT/T
      COMMON/CI/I1,I2,IMAX,I1P1,I1P2,I2M1,I2M2,IBF
      COMMON/CJ/J1,J2,JMAX,J1P1,J1P2,J2M1,J2M2,JA,JB,JAM1,JBP1
      COMMON/CK/K1,K2,KMAX,K1P1,K1P2,K2M1,K2M2
      COMMON/CNI/L
      DATA N1,N3,N4,N7,N10,N14,N17/1,3,4,7,10,14,17/

      READ(NXYZ) I1,I2,J1,JA,K1,K2
C
      IF(J1.GE.1.AND.K1.GE.1) THEN
         N4=4
         J1=J1+1
         J2=2*JA+1
         JA=JA+1
         K1=K1+1
         CALL NORM(N10,N7,N4,N14,N17,I2)
      ENDIF
      END

      SUBROUTINE NORM(LI,NI,MI,NN,NC ,I)
      DIMENSION T(52,21,60)
      DIMENSION TI(3)

      COMMON/T/T
      COMMON/I/I1,I2,IMAX,I1P1,I1P2,I2M1,I2M2,IBF
      COMMON/J/J1,J2,JMAX,J1P1,J1P2,J2M1,J2M2,JA,JB,JAM1,JBP1
      COMMON/K/K1,K2,KMAX,K1P1,K1P2,K2M1,K2M2
      COMMON/IO/LEC ,IMP,KIMP,NXYZ,NGEO,NDIST

C ....
DO 300 K=K1,K2
      DO 300 J=J1,JA

      CALL PVNMUT(TI)
      T(J,K,NN  )=S*TI(1)
      T(J,K,NN+1)=S*TI(2)
      T(J,K,NN+2)=S*TI(3)
  300 CONTINUE
C ....
      END

      SUBROUTINE PVNMUT(C)
      DIMENSION C(3), CX(3)
      CX(1)= 1
      CX(2)= 2
      CX(3)= 3
      R=SQRT(CX(1)*CX(1)+CX(2)*CX(2)+CX(3)*CX(3))
      IF(R.LT.1.E-12) R=1.
      DO I = 1,3
      C(I) = CX(I)/R
      ENDDO
      RETURN
      END
```

Figure 3: Another excerpt from AILE: NORM

1. using *array regions* to perform the dependence analysis;
2. using *IN* and *OUT array regions* to privatize the array `TI`.

This leads to the parallelized version of Figure 4.

```
      SUBROUTINE NORM(LI,NI,MI,NN,NC ,I)
      DIMENSION T(52,21,60)
      DIMENSION TI(3)

      COMMON/CT/T
      COMMON/I/I1,I2,IMAX,I1P1,I1P2,I2M1,I2M2,IBF
      COMMON/J/J1,J2,JMAX,J1P1,J1P2,J2M1,J2M2,JA,JB,JAM1,JBP1
      COMMON/K/K1,K2,KMAX,K1P1,K1P2,K2M1,K2M2
      COMMON/IO/LEC ,IMP,KIMP,NXYZ,NGEO,NDIST

C     ....
      DOALL K = K1, K2
         PRIVATE J
         DOALL J = J1, JA
            PRIVATE TI
            CALL PVNMUT(TI)
            T(J,K,NN) = S*TI(1)
            T(J,K,NN+1) = S*TI(2)
            T(J,K,NN+2) = S*TI(3)
         ENDDO
      ENDDO
C     ....
      END
```

Figure 4: Parallelized version of `NORM`.

2.2 RENPAR6

`RENPAR6` is a contrived example (see Figure 5) designed to show on a very simple program the power of READ, WRITE, IN and OUT regions, and some particular details of their computations, especially when integer scalar variables that appear in array indices are modified.

The main purpose is to see that array `WORK` is only a temporary and can be privatized. Notice that the value of `K` is unknown on entry to the loop `I`, and that its value is modified by a call to `INC1` at each iteration (`INC1` simply increments its value by 1).

We are interested in the sets of array elements that are referenced in each iteration. However, since the value of `K` is not the same in the two written references, we cannot summarize the write accesses if we do not know the relation that exists between the two values of `K`. This is achieved in PIPS by using transformers, that here show how the new value of `K` is related to the value before the `CALL` (`K#init`):

`T(K) {K==K#init+1}`

And the transformer of the loop shows how the value of `K` at each step is related to the values of `I` and `K#init` (value of `K` before the loop):

```
SUBROUTINE RENPAR6(A,N,K,M)
INTEGER N,K,M,A(N)
DIMENSION WORK(100,100)
K = M * M
DO I = 1,N
   DO J = 1,N
      WORK(J,K) = J + K
   ENDDO

   CALL INC1(K)

   DO J = 1,N
      WORK(J,K) = J * J - K * K
      A(I) = A(I) + WORK(J,K) + WORK(J,K-1)
   ENDDO
ENDDO
END

SUBROUTINE INC1(I)
I = I + 1
END
```

Figure 5: Contrived example: `RENPAR6`

```
T(I,K) {K==I+K#init-1}
```

This previous information is used to summarize the sets of elements that are read or written by each program structure. In order to compute the summary for the loop `I`, we must merge the sets for the two `J` loops. Be careful that the value of `K` is not the same for these two loops. We must use the transformer of the `CALL` to translate the value of `K` in the second region into the value of `K` before the CALL. At this step, we have a summary of what is done by a single iteration. We then compute the regions for the whole loop `I`. This is done with the help of the transformer of the loop that gives the relation between `K` and `I`.

However, as we have seen with `NORM`, READ and WRITE regions are not sufficient for array privatization, because we must verify that every element of `WORK` that is read by an iteration is previously written in the same iteration. This is achieved by the IN region. Then OUT regions allow us to verify that no element of `WORK` is used in the subsequent iterations or in the continuation of the loop.

We can now try to parallelize `RENPAR6` by:

1. using *transfomers* to compute *array regions*;
2. using *array regions* to perform the dependence analysis;
3. using *IN* and *OUT array regions* to privatize the array `WORK`.

This leads to the parallelized version of Figure 6. The array `WORK` is privatized in loop `I`. However, the loop is not parallelized, because automatic induction variable substitution is not available in PIPS. This transformation has been performed by hand. This leads to the subroutine `RENPAR6_2` in figure 7. And after array privatization, PIPS is able to parallelize the loop `I` (see Figure 8).

```
SUBROUTINE RENPAR6(A,N,K,M)
INTEGER N,K,M,A(N)
DIMENSION WORK(100,100)
K = M*M
DO I = 1, N
   PRIVATE WORK,I
   DOALL J = 1, N
      PRIVATE J
      WORK(J,K) = J+K
   ENDDO
   CALL INC1(K)
   DOALL J = 1, N
      PRIVATE J
      WORK(J,K) = J*J-K*K
   ENDDO
   DO J = 1, N
      PRIVATE J
      A(I) = A(I)+WORK(J,K)+WORK(J,K-1)
   ENDDO
ENDDO
END
```

Figure 6: Parallelized version of RENPAR6.

```
SUBROUTINE RENPAR6_2(A,N,K,M)
INTEGER N,K,M,A(N)
DIMENSION WORK(100,100)
K0 = M * M
DO I = 1,N
   K = K0+I-1
   DO J = 1,N
      WORK(J,K) = J + K
   ENDDO

   CALL INC1(K)

   DO J = 1,N
      WORK(J,K) = J * J - K * K
      A(I) = A(I) + WORK(J,K) + WORK(J,K-1)
   ENDDO
ENDDO
END
```

Figure 7: RENPAR6_2.

```
SUBROUTINE RENPAR6_2(A,N,K,M)
INTEGER N,K,M,A(N)
DIMENSION WORK(100,100)
K0 = M*M
DOALL I = 1, N
   PRIVATE WORK,J,K,I
   K = K0+I-1
   DOALL J = 1, N
      PRIVATE J
      WORK(J,K) = J+K
   ENDDO
   CALL INC1(K)
   DOALL J = 1, N
      PRIVATE J
      WORK(J,K) = J*J-K*K
   ENDDO
   DO J = 1, N
      PRIVATE J
      A(I) = A(I)+WORK(J,K)+WORK(J,K-1)
   ENDDO
ENDDO
END
```

Figure 8: Parallelized version of RENPAR6_2.

In fact, IN and OUT regions could also be used to reduce the set of elements of array WORK to allocate to each processor, because each iteration only accesses a sub-array. These regions provide an exact representation of the set of elements that are actually needed.

References

[1] Béatrice Creusillet. IN and OUT array region analyses. In *Fifth International Workshop on Compilers for Parallel Computers*, June 1995.

[2] Béatrice Creusillet and François Irigoin. Interprocedural array regions analyses. In *Language and Compilers for Parallel Computing*, August 1995.

[3] François Irigoin. Interprocedural analyses for programming environments. In *Workshop on Environments and Tools for Parallel Scientific Computing*, September 1992.

[4] François Irigoin, Pierre Jouvelot, and Rémi Triolet. Semantical interprocedural parallelization: An overview of the PIPS project. In *International Conference on Supercomputing*, June 1991.