Program Transformation for Locality Using Affinity Regions

Bill Appelbe, Charles Hardnett, and Sri Doddapaneni

College of Computing,
Georgia Institute of Technology, Atlanta, GA 30332

Abstract. Affinity regions ensure that a shared processor schedule, mapping loop iterations to processors, is used in consecutive parallel loop nests. Using affinity regions can improve locality without affecting parallelism.
Unlike loop fusion, affinity regions are always safe. Also, unlike parallel regions, affinity regions do not require explicit code for mapping loop iterations to processors. While affinity regions improve locality, there may be cases where more than one affinity region is possible for the same pair of loop nests. Also, loop transformations such as interchange and alignment can affect the profitability of affinity regions. In this paper we discuss the interaction between loop transformations and affinity regions, and the relationship between affinity regions and global optimization for parallelism and locality. The algorithms in this paper are being implemented in the parallelization tool, PAT, targeted at generating of affinity regions for the KSR-1.

1 Introduction

An *affinity region* is a sequence of parallel loop nests which share a common schedule (mapping iterations to processors)[4]. An `AFFINITY REGION` construct ensures that a shared processor scheduling, mapping iterations to processors, is used in consecutive parallel loop nests, that are declared as members of the `AFFINITY REGION`. Using affinity regions can significantly improve locality of reference, and hence performance. Many parallel programs consist of sequences of parallel loops accessing common data arrays with regular access patterns and overlapping references. The primary reason for using affinity regions is to minimize data movement between processors. Scheduling using affinity regions can ensure that processors access the same array sections in successive loop nests.

Affinity regions are targeted at shared-memory or NUMA (Non-Uniform Memory Access) multiprocessors, such as the Kendall Square KSR-1[2]. However, the algorithms used to find affinity regions are applicable to optimization for distributed memory systems as explained below.

In principle, the scheduling of parallel loops in an affinity region can use either self-scheduling or pre-scheduling strategies. Pre-scheduling specifies the iteration to processor mapping statically, whereas self-scheduling is a dynamic mapping. The shared schedule used in an affinity region can either be determined statically, or by the first loop executed in the region. Although, self-scheduling results in proper load balancing, pre-scheduling can result in better locality[6].

In practice, aside from affinity regions, there are two other ways of achieving data locality between loop nests – *loop fusion* and `PARALLEL REGIONS`. However, affinity regions are more general than loop fusion, and a higher level construct than parallel regions[4]. To illustrate the effectiveness of affinity regions we will use the following program fragment:

```
PARALLEL LOOP J=1,N
   PARALLEL LOOP I=1,N
      A(I,J)=0
      LOOP K=1,N
         A(I,J) = A(I,J) + B(K,I)*C(K,J)
      END LOOP
   END LOOP
ENDLOOP
PARALLEL LOOP I = 1,N
   LOOP K=1,N
      SUM(I) = SUM(I) + B(K,I)
   END LOOP
END LOOP
```

Program 1: Parallel Loop Nests

In this example we can fuse the two outer loop nests after the first loop has been distributed (to remove the initialization of `A`), and the resulting loop nest without the initialization has been interchanged. However, fusion would create more problems than it solves. Although fusion can obtain locality for `B`, the `A` initialization had to be distributed, so locality for `A` has been lost.

Alternatively, the two loop nests in Program 1 can be placed in a `PARALLEL REGION`, and an explicit explicit mapping of iterations to processors used:

```
PARALLEL LOOP I = 1, N
   PARALLEL LOOP J = 1, N
      A(J,I) = 0.0
   END LOOP
END LOOP
CALL PTHREAD_PROCSETINFO(NUMPROCS, IPROC_IDS, 1, ISTAT)
NUMTHREADS = NUMPROCS
PARALLEL REGION(NUMTHREADS)
MYNUM = IPR_MID()+1
LOOP I = MYNUM, N, NUMPROCS
   LOOP J = 1, N
      LOOP K = 1, N
         A(I,J) = A(I,J) + B(K,I)*C(K,J)
      ENDLOOP
   ENDLOOP
```

```
ENDLOOP
LOOP I = MYNUM, N, NUMPROCS
    LOOP K = 1, N
       SUM(I) = SUM(I) + B(K,I)
    ENDLOOP
ENDLOOP
END PARALLEL REGION
```

Program 2: Parallel Regions

To summarize, the disadvantages of using either fusion or parallel regions to obtain locality are:

fusion is possible only if the loops are adjacent (no intervening serial code that cannot be moved) and fusion of the loops will not result in the introduction of loop-carried dependences in the resulting fused loop nest.

parallel regions use explicit processor or thread to iteration mappings, which are machine dependent, difficult to maintain, and error-prone. If threads (logical processors) are used, then the program must still ensure that the threads do not migrate to a different processor within the region.

Using the affinity region construct of the KSR-1 together with PTILEs (parallel loops), we can rewrite Program 1 in KSR Fortran[1] as:

```
        DO 10 J = 1, N
           DO 20 I = 1, N
              A(I,J) = 0
    20     CONTINUE
    10  CONTINUE
C*KSR* AFFINITY REGION(I:1,N, K:1,N)
        DO 30 J = 1, N
C*KSR* PTILE(I,K,TILESIZE=(I:1,K:N),AFFMEMBER=1)
           DO 40 I = 1, N
              DO 50 K = 1, N
                 A(I,J) = A(I,J) + B(K,I)*C(K,J)
    50        CONTINUE
    40     CONTINUE
    30  CONTINUE
C*KSR* PTILE(I,K,TILESIZE=(I:1,K:N),AFFMEMBER=1)
        DO 60 I = 1, N
           DO 70 K = 1, N
              SUM(I) = SUM(I) + B(K,I)
    70     CONTINUE
    60  CONTINUE
C*KSR* END AFFINITY REGION
```

Program 3: Affinity Regions (KSR Fortran)

KSR Parallel Fortran specifies parallel loop nests using the `PTILE` directive, whose arguments include:

- The parallel loop indices (`I` and `K` in both loop nests above).
- The size and shape of tiles allocated to threads (`TILESIZE=(I:1,K:N)` in the first loop nest above).
- If the loop nest is a member of an enclosing affinity region (`AFFMEMBER=1` or true).

In order to use affinity regions effectively, we need to provide answers to several questions.

Feasibility When can two loop nests be placed in an affinity region, with some gain in locality?

Profitability How can the performance gains from using affinity regions be quantified?

The answer to this question allows us to compare the performance of affinity regions to that of other transformations such as fusion.

Strategy How do affinity regions interact with other program transformations, especially intraloop transformations such as interchange and skewing?

Answers to the first two questions were provided in [4]. In summary, there needs to be a reaching dependence (including input dependences) between the two loop nests, of an array variable `A`, whose index is dependent upon a parallel loop index in both loops. In addition, there must be a 1-1 mapping between a subset of the ranges of the parallel loops in the two loop nests ($\mathcal{L}_{MAP} : \mathcal{L}^1_{Par-index} \mapsto \mathcal{L}^2_{Par-index}$) such that under the mapping identical array elements are accessed (if s^1 is the subscript expression vector of `A` in the first loop, and s^2 is the subscript expression vector of `A` in the second loop nest, then for each subscript dimension j of A: either $s^2_j \equiv \mathcal{L}_{MAP}(s^1_j)$, or s^1_j does not contain any subscripts in $Domain(\mathcal{L}_{MAP})$ and s^2_j does not contain any subscripts in $Range(\mathcal{L}_{MAP})$).

Affinity regions lead to definite performance improvements. The following table shows the relative execution time of Program 3 with and without affinity regions. The program was run on a KSR-1 with 32 nodes.

Size `N`	Without Affinity Regions	Affinity Regions
15	78.8	21.4
20	100.1	24.1
30	132.7	38.4
90	364.7	167.2

Profitability can be measured by the number of (possibly) non-local memory references converted to local memory references by the use of an affinity region. A complete analysis of the profitability of affinity regions requires a machine model. For example, on the KSR-1 prefetching and post-store can be used to overlap computations with data movement in order to reduce the timing overhead involved in accessing data from non-local memory to further improve locality.

This paper addresses the last question: how does the use of affinity regions interact with other transformations for parallelism?

At first sight, the question seems fairly trivial, as affinity regions can be inserted only when loop nests have already been parallelized. Thus, the insertion of affinity regions might appear to be carried out in a simple post-pass after parallelization. However, the following analysis shows that the interactions are not as simple as they appear.

The analysis below assumes a general model for affinity regions[4], which is not constrained by the limitations of the current implementations imposed by the KSR-1.

2 Transformations for Affinity

There may be more than one possible affinity region to encompass two loop nests. This in turn implies that the affinity regions may differ in the common subscripted variable being used. It would be profitable if affinity regions can be merged in some way. The merging of affinity regions can be made possible by the use of transformations in [4]. The transformations that we are discussing in this paper are chiefly *tiling*, an optimization for scheduling iterations, *alignment* (modifying the subscript expressions), and *loop distribution* and *loop skewing* (iteration space transformations).

2.1 Tiling

Iteration space tiling is a program transformation that tiles the iteration space in order to achieve better data reuse and reduce data movement. Consider the following code fragment:

```
LOOP J = 1, N
   LOOP I = 1, N
      B(I,J) = ...
   END LOOP
END LOOP

LOOP J = 1, N
   LOOP I = 1, N
       X(I,J) = B(I+1,J) + B(I,J)
   END LOOP
END LOOP
```

Program 4: Tiling

If the two loop nests, $\mathcal{L}^1$ and $\mathcal{L}^2$, are scheduled independent of each other, self-scheduling and tiling might be used for $\mathcal{L}^1$ and $\mathcal{L}^2$ respectively. Although tiling for $\mathcal{L}^2$ improves data locality, the execution of the two loop nests using different scheduling strategies results in a large amount of data movement. However, the two loop nests meet the criteria necessary to be put into an affinity region[4]. The affinity region construct insists that the same scheduling strategy be used in both loop nests. Since $\mathcal{L}^2$ requires tiling for better data locality, the same strategy will be used in $\mathcal{L}^1$ if they are both in the affinity region. Insertion of affinity regions therefore resulted in converting the self-scheduling strategy of $\mathcal{L}^1$ into one of tiling to preserve data locality between loop nests. Thus, the scheduling strategy used for an affinity region needs to be chosen based upon all loops in the nest, even if that means reducing the number of levels of parallelism.

2.2 Alignment

We initially assumed that the affinity regions were inserted after all the transformations for maximal parallelism were done.

```
LOOP J = 1, N
   LOOP I = 1, N
      A(I,J) = C(I,J)
      C(I-1,J)  = B(I,J)
   END LOOP
END LOOP
LOOP J = 1, N
   LOOP I = 1, N
      A(I,J) = ....
      B(I,J) = ....
   END LOOP
END LOOP
```

Program 5: Alignment

Consider the scenario illustrated by the above code fragment. If $\mathcal{L}^1$ is transformed for maximal parallelism, then the access to array `C` in the second statement of $\mathcal{L}^1$ will be aligned with the first statement resulting in the change in the subscript expression of `B` as shown below (ignoring peeled iterations).

```
C Peeled iterations
           LOOP J = 1, N
              LOOP I = 2, N-1
                 A(I,J) = C(I,J)
                 C(I,J) = B(I+1,J)
              END LOOP
           END LOOP
C Peeled iterations
```

Program 6: Aligned Loop nest 1 for Parallelism

The loop nests $\mathcal{L}^1$ and $\mathcal{L}^2$ are also *affinity adjacent* (sharing a variable with a reaching dependence [4]) based on their access to `A`. However, there are potentially two possible affinity regions for the above loop nests - one based on the accesses to A and the other based on the accesses to the subscripted variable B. Applying either one of these two possible affinity regions results in the movement of data of the other array resulting in a loss of data locality between the two loop nests.

To maximize the data locality between the two loop nests, we need to merge the two possible affinity regions. In order to do the merge, we align the accesses to the array B in $\mathcal{L}^2$ with the accesses to the same array in $\mathcal{L}^1$. The transformed code is shown below (ignoring peeled iterations again).

```
LOOP J = 1, N
   LOOP I = 2, N-1
      A(I,J) = C(I,J)
      C(I,J)  = B(I+1,J)
   END LOOP
END LOOP
LOOP J = 1, N
   LOOP I = 2, N-1
      A(I,J) = ....
      B(I+1,J) = ....
   END LOOP
END LOOP
```

Program 7: Aligned Loop Nest 2 for Data Locality

The alignment of accesses to B is made possible due to the absence of dependences that may have prevented maximal parallelism in $\mathcal{L}^2$. We may run into a situation where the alignment along a certain dimension for improved data locality between two loop nests reduces the degree of parallelism or more specifically, the loop nest will not be maximally parallel but can be run in parallel with synchronization points in the loop. These synchronization points in the resulting code fragment also cause messages to go form one processor to the other. The overhead incurred because of the communication to synchronize iterations can be equated to the overhead due to data movement when the loop nests are maximally parallel. We are looking into a more sophisticated performance model to make the decision in favor of one or the other.

2.3 Linear Transformation of Subscripts

It may not always be possible to use alignment for transforming the subscript expressions so that the accesses to the subscripted variables preserve data locality. For instance, consider the program fragment shown in program 8:

```
LOOP J = 1, N
   LOOP I = 1, N
      A(I,J) = ....
      B(I,J)  = ....
   END LOOP
END LOOP
```

```
LOOP J = 1, N
   LOOP I = 1, N
      A(I,J) = ....
      B(I, N-J) = FUNC(I,J)
   END LOOP
END LOOP
```

Program 8: Reverse Access

The access pattern of B in the two loop nests indicate that the data locality will be lost if the two loop nests are surrounded by an affinity region based on the accesses to the array A. Also, tiling and alignment are not appropriate transformations to preserve preserve data locality when B is a member of an affinity region based on accesses to A. There is no 1 to 1 mapping of iterations in the first loop nest to iterations in the second loop nest that will preserve locality for both A and B, although locality can be preserved for either array individually. However, the use of linear transformation on the subscript expression of B in the second loop nest, as shown in Program 9, ensures that data locality for both A and B is maintained without any loss of correctness of the program semantics.

```
LOOP J = 1, N
   LOOP I = 1, N
      A(I,J) = ....
      B(I,J)  = ....
   END LOOP
END LOOP
LOOP J = 1, N
   LOOP I = 1, N
      A(I,J) = ....
      B(I,J) = FUNC(I,N-J)
   END LOOP
END LOOP
```

Program 9: Linear Transformation for Affinity

3 Transformations not affecting Subscript Expressions

We have already stated that transformations play a major role in the insertion of affinity regions only when there is more than one potentially applicable affinity region. The transformations that have been discussed so far modify the subscript expressions of the subscripted variables so that the access pattern to the subscripted variables under a specified iteration space access pattern is altered to reflect the enhancement in the data locality for the variable. In this section, we conjecture that the transformations that do not transform the subscript expressions but transform the loops alone do not improve data locality or, in other words, do not improve the possibilities of merging the potentially applicable affinity regions. Our conjecture is based on the following scenario:

- There are several potential affinity regions.
- If a potential affinity region has to be merged then the access pattern has to be transformed
- The iteration space distribution or any other transformation of the iteration space can aid in achieving the desired data locality for both the subscripted variables `A` and `B` if and only if the affinity regions can be merged with a different mapping between the iteration space of the two loop nests.

We know that loop distribution can be used to improve parallelism. Also, it has been shown that loop fusion results in better data locality in the presence of memory hierarchy[5, 6]. For example:

```
LOOP I = 1, N
   A(I) = ....
   B(I) = ....
END LOOP
LOOP J = 1, N
   A(I) = ....
   B(I+STEP) = A(I)
END LOOP
```

Program 10: Fragment for Loop Distribution

In the example program 10, there are two possible affinity regions based on the accesses to the variables `A` and `B` respectively. But the affinity regions cannot be merged by applying any of the aforementioned subscript transformations because they do not suffice to both improve locality as well as ensure correctness. Instead, we can distribute the second loop nest so that the affinity region based on `A` does not affect the accesses to `B` as shown in program 11.

```
LOOP I = 1, N
   A(I) = ....
   B(I)  = ....
END LOOP
LOOP J = 1, N
   A(I) = ....
END LOOP
LOOP J = 1, N
   B(I+STEP) = A(I)
END LOOP
```

Program 11: Distributed Loop nests

However, when we have to execute the second part of the distributed loop nest containing the accesses to `B`, the data locality for `A` will lost anyway since it also has accesses to `A`.

In general, the alignment cannot be applied to the loop nests to improve affinity because there is a cyclic dependence, either direct or indirect, in at least one of the statements in the loop nest. If the iteration space of the loop is transformed, the resultant loop nests may have more parallelism. But the distribution has not resulted in any relative change in the access pattern among the subscripted variables involved in the cycle. According to our conjecture, if there is no change in the relative data access pattern then, invariably, there is no improvement in the data locality.

4 Global Optimization for Parallelism and Locality

Finding affinity regions is based upon being able to find a 1-1 mapping between a subset of the ranges of the parallel loops in the two loop nests ($\mathcal{L}_{MAP} : \mathcal{L}^1_{Par-index} \mapsto \mathcal{L}^2_{Par-index}$) which preserves locality for a particular array accessed in both loops. This is related to the approach of Anderson and Lam[3], who use an explicit iteration to processor mapping *vecc* and an explicit array to processor mapping **d**. Composing these mappings yields an iteration to iteration mapping which can be equivalent to $\mathcal{L}_{MAP}$. However, there are some significant differences in the two approaches:

1. Introducing an explicit mapping for data **d** is not necessary, as the distribution of data can be deduced. In addition, it is difficult to reconcile with data migration, replication, and renaming.
2. The mappings *vecc* and **d** are restricted to affine expressions (which can be represented by matrices).
3. Their approach is based upon putting loop nests into maximally parallel form without performing any statement transformations. A general approach must incorporate statement transformations such as those discussed in the paper. In addition, putting a loop nest into maximally parallel form with associated overheads such as skewing introduces is not always necessary as parallelism may only be exploited in a subset of the dimensions when locality is considered.

5 Conclusion

Even if a parallel programming language does not support affinity regions directly, they are still a highly desirable high-level approach to improving locality. Affinity regions are generally applied as a "post-pass" after parallelization. However, our analysis shows that they do interact subtly with other transformations for parallelism. We are currently implementing the algorithm for feasibility detection of affinity regions, and heuristics for combining them with other program transformations. Also, the mapping used in affinity regions offers promise as a new approach to global optimization for locality and parallelism.

References

1. *KSR Fortran Programming.* Kendall Square Research, Waltham, Massachusetts, 1991.

2. *KSR Parallel Programming.* Kendall Square Research, Waltham, Massachusetts, 1991.
3. Anderson, J., and Lam, M. S. Global optimzations for parallelism and locality on scalable parallel machines. In *SIGPLAN Programming Language Design and Implementation* (1993), pp. 112–125.
4. Appelbe, W. F., and Lakshmanan, B. Optimizing parallel programs using affinity regions. In *International Conference on Parallel Processing* (August 1993).
5. Wolfe, M. *Optimizing Supercompilers for Supercomputers.* The MIT Press, Cambridge, Massachusetts, 1989.
6. Zima, H., and Chapman, B. *Supercompilers for Parallel and Vector Computers.* ACM Press, New York, New York, 1990.