A New Algorithm for Global Optimization for Parallelism and Locality

Bill Appelbe, Srinivas Doddapaneni and Charles Hardnett

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

Abstract. Converting sequential programs to execute on parallel computers is difficult because of the need to globally optimize for both parallelism and data locality. The choice of which loop nests to parallelize, and how, drastically affects data locality. Similarly, data distribution directives, such as `DISTRIBUTE` in High Performance Fortran (HPF), affects available parallelism and locality. What is needed is a systematic approach to converting programs to parallel form, based upon analysis that identifies opportunities for both parallelism and locality in one representation.

This paper presents a global framework for optimizing parallelism and locality, based upon constraint solving for locality between potentially parallel loop nests. We outline the theory behind the framework, and provide a global algorithm for parallelizing programs while optimizing for locality. We also give results from applying the algorithm to parallelizing the Perfect benchmarks, targeted at the KSR-1, and analyze the results. Unlike other approaches, we do not assume an explicit distribution of data to processors. The distribution is inferred from locality constraints and available parallelism. This approach works well for machines such as the KSR-1, where there is no explicit distribution of data. However, our approach could be used to generate code for distributed memory processors (such as generating HPF) with explicit data distribution.

1 Introduction

In order to effectively parallelize programs for distributed memory computers such as the Paragon or CM-5 or "shared memory" architectures such as the KSR-1 it is necessary to optimize for locality of reference. On distributed memory systems such as the Paragon, data must be explicitly passed between processors, so the communication costs are obvious, at least at the object program level. On shared memory machines, such as the KSR-1, the communication costs are "hidden" as cycle stalls due to fetches from non-local memory. Nevertheless, the communication costs are a significant overhead in practice, as the results in section 4 show. Shared memory machines introduce further problems due to false sharing.

Most previous work has focused upon subproblems such as optimizing locality for a single loop nest [12, 9, 10]. Other authors have focused upon the problem of scheduling loop iterations and transforming loops for maximum parallelism[5, 11] or optimizing data distribution[6].

All these approaches do not generalize easily to the problem of globally optimizing for both parallelism and locality. The fundamental insight to overcoming these limitations is that parallelism and locality cannot be optimized independently. What is needed is a program representation and associated analysis which captures global opportunities for both parallelism and locality, without restrictive assumptions. The model needs to be able to handle locality at both the processor and cache levels. Once the representation has been constructed, we can then systematically select sequences of program transformations and data distributions to parallelize the program and distribute data (using a machine dependent cost model when necessary).

Also, we do not want to preclude any parallelization transformations a priori, or apply any transformations before applying the global framework analysis, or restrict ourselves to any particular subset of transformations or data distributions initially.

This paper presents just such a general framework for Parallelism and Locality, PAL. In this paper we first define the PAL framework and its assumptions. We then show how it is applied to some "hard to parallelize" program fragments. We give a global algorithm for parallelization and data distribution using this model. Results of applying the algorithm to benchmarks are given, and the results discussed. Finally, we contrast our approach with other approaches to global parallelization strategies.

2 PAL Framework

Our approach is based on several underlying assumptions.

1. Available parallelism is at the loop-level. We want to find loops that can be executed in parallel, with different iterations mapped to different processors. Thus the our approach is not intended for fine-grain, instruction-level parallelism or for large-grain data-flow.
2. Dependence distances or directions are known. Of course, if a few dependences are not known then loop distribution can sometimes be used and loops containing only known dependences parallelized. For such loops it may not be possible to optimize at compile-time for locality of references. If not, the program may be amenable to parallelization by the inspector/executor model of runtime dependence testing.
3. The program is sequential, and no data distribution has been performed. This is hardly a restriction, as HPF permits data distributions to be ignored, and the model could either ignore parallel loop directives or constrain the solution space by these.

The input to our algorithm is a source program written in an imperative programming language such as Fortran, and we assume that syntactic and semantic analysis have been done. Thus, we are ignoring issues such as interprocedural data-flow analysis and so on.

Our approach can be summarized as a four stage process:

1. Determining available parallelism.
2. Determine the *locality constraints* upon the available parallelism.
3. Global optimization for locality.
   Construct the PAL graph, which represents all locality constraints and available parallelism. Then try to satisfy these constraints while still retaining as much as possible of the available parallelism. When all constraints cannot be met, break locality constraints that have the lowest cost (communication overhead).
4. Code Generation.
   Choose parallelization strategies that meet the remaining constraints and provide low overhead, minimize false sharing, and good load balancing. We use tiling, with tile sizes based upon the number of processors and problem size. Optimize communication, by overlapping it with computation using asynchronous messages (*prefetch* and *poststore* operations on the KSR).

Each of these steps are discussed in the following sections

2.1 Determining Available Parallelism

The goal of determining available parallelism is to classify each loop in each loop nest in the program as:

parallelizable no loop carried dependences, or parallelizable by statement transformations, or
skew-parallelizable two or more perfectly nested loops with recurrences, or
serial a loop with recurrences, not nested with other loops with recurrences.

For a loopnest $\mathcal{L}$, of depth d, the iteration space is denoted by $\mathcal{I}$, and consists of a set of iterations. Each iteration is identified by its index vector $\mathbf{i} = (i_1, i_2, i_3, ...i_d)$. The set of loop indices is denoted by $\mathcal{L}_{index}$, and the subset of these which are not serial is called the *parallel index* set, and denoted by $\mathcal{L}_{Par-index}$.

Thus, available parallelism simply means loops which are not serial. Loop carried dependences can be systematically removed by statement transformations (*alignment*, *replication*, *substitution*) [4] and by standard transformations for privatization of local variables. If there are no recurrences (more precisely, no cycles in the dependence graph for that loop level with all dependence directions positive), then all dependences can be removed, otherwise recurrences can be isolated in serial loops using loop distribution. These transformations can be applied to each level of a loop nest, including loops containing conditional statements[2].

An alternative approach is to remove dependences by skewing and other unimodular transformations (transforming the iteration space rather than statements). In an N-level loop this can result in N-1 levels of parallelism[11]. However, skewing introduces a communication overhead (which can later be minimized by block skewing).

Iteration space skewing introduces parallelism in nested loops with dependence cycles that cannot be broken by replication, such as the following stencil fragment:

```
FOR I = 2 TO N DO
    FOR J = 2 TO N DO
        A(I,J) = f(A(I-1, J-1), A(I, J-1), A(I-1, J))
```

Example 2: Stencil Program Fragment

Since there are cycles in both dimensions, statement transformations cannot be used to introduce parallelism. Instead, the iterations must be skewed, after blocking for locality. The iteration space after block skewing can be pictured as shown in figure 1:

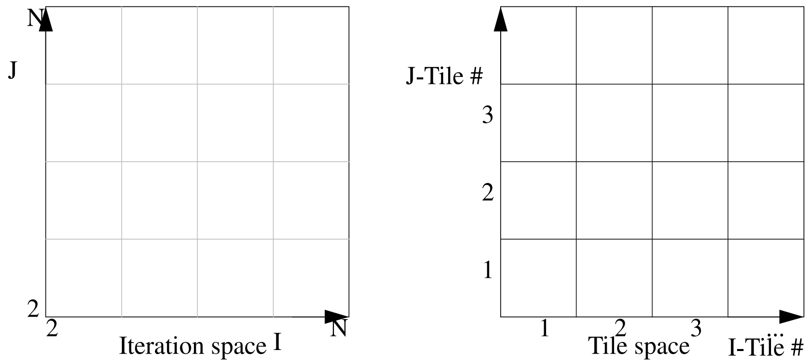

Figure 1 Tiling Iteration Spaces

Any tile (i, j) cannot be executed until tiles $(i-1, j)$ and $(i, j-1)$ have completed, and so on. However, tiles can be scheduled so that all tiles in a given row or column can be executed by the same processor without any degradation over a dynamic scheduling strategy. Once a processor completes execution of a tile (i, j), it can be allocated to execute either tile $(i+1, j)$, or $(i, j+1)$, or a new row or column of tiles. By allocating the same processor to a row or column of tiles, locality is improved, as data must always be transferred across tile boundaries in all tile dimensions. The choice of row or column parallelism is dictated either by compatibility for locality with other loop nests (explained below), or by cache considerations (e.g., row or column major array allocation).

Effectively, if a loop can be parallelized only by skewing the iteration space in k dimensions, then the same processor can be allocated to any $k-1$ of these dimensions or a finer granularity of parallelism can be obtained by parallelizing in up to $k-1$ dimensions.

In general, statement alignment *always* improves locality, and is preferable for removing dependences.[1] However, replication (for removing anti-dependences)

[1] More accurately, statement alignment can never degrade locality, as alignment shifts all accesses of an array variable to the same iteration, and hence the same processor.

has a computational overhead (making copies), although strip mining can be used to reduce copying[2]. Similarly, distribution introduces an overhead although it may improve parallelism in a given dimension. Hence, in our model we *identify* available parallelism rather than actually performing the transformations before analyzing for locality. Hence, in Example 1 above, we would identify the loop as parallelizable in the { `I,J` } dimensions by statement alignment in both dimensions.

3 Determining Locality Constraints

Locality occurs when the data accessed by a processor is resident in that processor's memory. We will now give a general definition of locality under the assumption that a data element is resident on only one processor (later this will be relaxed).

When a parallel loop nest $\mathcal{L}$ is executed, each iteration will be executed by some processor. We assume that there is an unbounded set or space of logical processors $\mathcal{P}$. The iteration space or set of iterations $\mathcal{I}$ of a parallel loop nest is *partitioned* into an ordered set of iterations executed by each processor:

$P(\mathcal{I}) = \{\mathcal{I}_1, ..., \mathcal{I}_p, ...\}, p \in \mathcal{P}$

The partition is constrained only by the requirement that there should be no loop carried dependences between two iterations in different elements of the partition. The finest partition, corresponding to the finest granularity of parallelism, is one in which each iteration in the iteration subspace spanned by the parallel index set $\mathcal{L}_{Par-index}$ is executed by a different processor. [2]

For an access of an array `A`, $A(\mathbf{f}(\mathbf{i}))$, within a parallel loop nest, the partition induces a set of elements of `A` accessed by a processor p: $A^{'}_p = \{A(\mathbf{f}(\mathbf{i})) : \mathbf{i} \in \mathcal{I}_p\}$. Thus, any partition P of the iteration space $\mathcal{I}$ induces a *distribution* D of the data for the access $A^{'}$ to each processor p.

$D(A(\mathbf{f}(\mathbf{i})), P(\mathcal{I})) = \{A^{'}_0, ...A^{'}_p ...\}$

This must be a partition if there are any loop-carried dependences on `A` in this loop, as there cannot be dependences between partition elements.

For another access to the array `A`, $A^{''} = A(\mathbf{g}(\mathbf{j}))$, in this or another parallel loop nest $\mathcal{L}^{''}$, there will be a corresponding distribution of the elements of `A`, based upon the iteration space partition used in that loop and the array subscript expression.

If there is a reaching dependence (either true or input) from $A(\mathbf{f}(\mathbf{i}))$ to $A(\mathbf{g}(\mathbf{j}))$ then we need to ensure locality (different processors do not access the same elements of `A`). This occurs if the two iteration space partitions $P^{'}$ and $P^{''}$ satisfy the following locality constraint for the data distributions they induce on the accesses $A^{'}$ and $A^{''}$:

Even if the alignment is in a dimension that is not parallelized, some improvement in locality is likely due to cache level locality improvement.

[2] Strictly speaking, if levels are skew-parallelizable, then there there are dependences between elements in the finest partition. However, these can be ignored as they will be preserved by code generation later.

$$p, q \in \mathcal{P} : (A_p^{'} \cap A_q^{''}) \neq \emptyset \;\; implies \;\; that \;\; p = q \tag{1}$$

That is, there is no intersection between the elements of `A` accessed by different processors p and q in the set of processors $\mathcal{P}$.

This implies that there must be a 1-1 mapping between the subsets of the two partitions which access the same elements of `A` if they satisfy the locality constraint.

This mapping can be determined from the array subscript expressions, by equating them to determine the constraints on iterations which access the same array elements. If there is a reaching dependence from $A(\mathbf{f}(\mathbf{i}))$ in loopnest $\mathcal{L}^{'}$, to $A(\mathbf{g}(\mathbf{j}))$ in loop nest $\mathcal{L}^{''}$, and

$\mathbf{f}(\mathbf{i}) = (f_1(\mathbf{i}), f_2(\mathbf{i}), ...f_m(\mathbf{i})), \quad \mathbf{g}(\mathbf{j}) = (g_1(\mathbf{j}), g_2(\mathbf{j}), ...g_m(\mathbf{j}))$

Then, the *index constraints* for locality are:

$$\text{for each subscript } k \text{ in the range } 1..m, \; f_k(\mathbf{i}) = g_k(\mathbf{j}) \tag{2}$$

To satisfy locality between loop nest $\mathcal{L}^{'}$, with iteration space $\mathcal{I}^{'}$, and loop nest $\mathcal{L}^{''}$ with iteration space $\mathcal{I}^{''}$, the criteria is:

$$\mathbf{i} \in \mathcal{I}_p^{'} \text{ and } \mathbf{j} \in \mathcal{I}_q^{''} \text{ and } g_k(\mathbf{j}) = f_k(\mathbf{i}) \text{ for all } k \;\; implies \;\; that \;\; p = q \tag{3}$$

That is, if the same array element is accessed in the iteration $\mathbf{i}$ in $\mathcal{L}^{'}$ and $\mathbf{j}$ in $\mathcal{L}^{''}$, then these must be iterations executed by the same processor.

In general, the constraints cannot be solved unless we restrict the subscript expressions to simple forms. If we restrict the subscript expressions to be affine expressions of the loop indices, then any index constraint will be of the form:

$$\mathbf{F}.\mathbf{i} + F_0 = \mathbf{G}.\mathbf{j} + G_0 \tag{4}$$

where each of the F_i and G_i are integer constants.

If `A` is a one dimensional array, then $m = 1$ and there is just one index constraint. $\mathbf{F}.\mathbf{i} + F_0$ and $\mathbf{G}.\mathbf{j} + G_0$ each define a family of hyperplanes in the two iteration spaces.

One processor must execute all iterations in a hyperplane for locality. In effect, parallelism is in the direction normal to the hyperplane. This can be done in the source program by skewing the iteration space. For reasonable locality on shared memory machines like the KSR-1 it is necessary to tile (use a block distribution) at least in the column direction.

If any of the indices is a serial index then generally the constraint cannot be satisfied, as each processor will access a range of values even in the finest partition, and these ranges will overlap. For example,

```
A(I1) = ... /* in loop nest L': parallel index set = { I1 }
...
A(J1 + J2) = ... /* in loop nest L": parallel index set = { J1 }
```

If `J2` takes on a range of values, then no partition of the iteration spaces will satisfy the locality constraint (assuming that the `I1` and `J1` loops have stride less than the range of `J2`).

3.1 Satisfying Multiple Constraints

Each index constraint potentially induces a partition of two iteration spaces at the source and sink of the dependence. As shown above, any single affine index constraint (of indices in the feasible parallel index set) can be satisfied if none of the indices in the constraint are scalar. In general, several affine index constraints can be satisfied provided that they are over disjoint sets of indices in the parallel index sets. Each constraint specifies that all points (iteration space vectors) in a hyperplance be executed by one processor. However, as noted below, hyperplanes which differ only by a constant value (alignment) can be partially satisfied by using a tiling strategy (communication occurs only at the boundary of tiles). In addition, by tiling the iteration space using several hyperplanes, communication costs can be reduced.

3.2 An Algorithm to Find Partitions

Finding partitions that preserve locality is trivial: merely have all iterations together in one partition element (serialize the program). However, we want to find the finest grain of partition. By aggregating elements of a finer partition we obtain coarser partitions corresponding to coarser parallelism. Of course, if a coarser partition is used in one loop nest, then a corresponding coarser partition is usually needed in other loop nests to preserve locality.

Ignoring locality, the finest partitions can be created by allocating one processor per iteration in all loops that are parallelizable. The finest partition is unique (under the assumption that no communication is allowed between processors). Given the finest partition of each loop nest (by identifying all loop that are parallelizable by statement transformations), the finest partition can be constructed by using the locality constraint as follows. Given the two subscript expressions for the array accesses $A(\mathbf{f}(\mathbf{i}))$ and $A(\mathbf{g}(\mathbf{j}))$, if $\mathbf{g}(\mathbf{j}) = \mathbf{f}(\mathbf{i})$ then $\mathbf{i}$ and $\mathbf{j}$ must be in the same partition. A partition of both loops which satisfies this constraint is the finest granularity of parallelism which satisfies locality.

For example:

```
FOR ALL I1 = 1 TO N DO
    FOR ALL I2 = 1 TO N DO
        Y(I1,N-I2) += X(I1,I2)
FOR ALL J1 = 1 TO N DO
    FOR J2 = 1 TO N DO
        Z(J1,J2) = Z(J1,J2-1) + Y(J2,J1-1)
```

Example 3: Adjacent Loop Nests

In this example, the first loop nest is parallel in both `I1` and `I2`, whereas the second loop is parallel in `J1`. These give the initial partitions. Locality can be achieved for `Y` only if the two constraints `I1 = J2` and `N-I2 = J1-1` are satisfied. Since the second loop nest is serial in `J2`, the first constraint implies that the first loop nest must be serial in `I1`. The second constraint implies a mapping `I2` $\mapsto$ `N-J1+1`.

Consider another example:

```
FOR I1 = 2 TO N DO
    FOR I2 = 2 TO N DO
        A(I1,I2) = f(A(I1-1, I2-1), A(I1, I2-1), A(I1-1, I2))
FOR J1 = 2 TO N DO
    FOR J2 = 2 TO N DO
        A(J1,J2) = g(A(J1, J2-1))
```

Example 4: Adjacent Loop Nests Parallelizable by Skewing

In this example, the first loop nest is parallelizable only by skewing, that is the dimensions `I1` and `I2` are pseudo-parallel. Since there are four accesses to `A` in the first loop, there are four constraints with the access in the second loop, which is parallel in the `J1` dimension only. If we solve the constraints for the two assignments, it gives the solution that the first loop should be parallel in the `I1` dimension, and the mapping between iterations should be `I1` $\mapsto$ `J1`. Solving the second constraint gives the same parallel loops, but a different mapping `I1` $\mapsto$ `J1+1`. Obviously, these two mappings cannot be simultaneously used. However, these mappings differ only by a constant: one is of the form $m(\mathbf{i})$ and the other is of the form $m(\mathbf{i}) + k$. Then partial locality can be obtained by block decomposition of the iteration space. Partitioning the iteration spaces into blocks implies that the intersection between sets of array elements accessed by different processors is relatively small.

4 Global Optimization for Locality

Previous sections outlined how to solve constraints for a single dependence to obtain locality for that access. The key question is how to solve sets of constraints for the entire program, how to compromise when all constraints cannot be satisfied, and how to optimize communication.

Prior to global partitioning, we assume that a number of standard optimizations have been performed, and that symbolic analysis[8] has been done to simplify the program and determine the number of loop iterations. In practice, for the benchmarks below, we have used runtime traces to determine loop iterations. Also, we have used loop distribution to obtain perfect loop nests when possible and when inner loops were candidates for parallel execution.

The global algorithm for partitioning the iteration and data spaces, code generation, and communication optimization is as follows:

1. For each loopnest, $\mathcal{L}$, determine the candidate parallel index set $\mathcal{L}_{Par-index}$.
2. Determine the candidate *distributed array references*: any array access within a loopnest with a non-null candidate parallel index set, subscripted by an expression containing candidate parallel indices.
3. Determine the reaching dependences between all these distributed array references.
4. For each of these reaching dependences, determine the locality constraints as follows:

(a) Initialize copies of the parallel index sets for the source and sink of the dependence:
$\mathcal{L}'_{Dep-Par-index} = \mathcal{L}'_{Par-index}$ and $\mathcal{L}''_{Dep-Par-index} = \mathcal{L}''_{Par-index}$.

(b) Equate each of the subscript expressions, to obtain a set of constraints (equalities between subscript expressions), $C = \{c_1, c_2, ...c_{nc}\}$, where the initial number of constraints nc equals the number of dimensions of the array.

   i. remove any duplicate constraints and constraints with no indices in $\mathcal{L}'_{Dep-Par-index}$ and $\mathcal{L}''_{Dep-Par-index}$

   ii. if any remaining constraints contain indices not in either $\mathcal{L}'_{Dep-Par-index}$ or $\mathcal{L}''_{Dep-Par-index}$ (serial indices) remove this constraint and all indices in it from $\mathcal{L}'_{Dep-Par-index}$ and $\mathcal{L}''_{Dep-Par-index}$. Repeat until no further constraints removed.

(c) Annotate the reaching dependence with C.
Each subscript constraint c_k implies one or more mappings between parallel index sets and corresponding planes to parallelize in.

The end result of this stage is represented by the *PAL Graph*, which shows all parallelizable loop nests in the program, the constraints between them and the corresponding mappings between iteration spaces.

5. For each pair of loop nests $\mathcal{L}'$ and $\mathcal{L}''$ with locality constraints between them, determine the intersection of all these constraints. The intersection of two constraint sets $C' = \{c'_1, c'_2, ...c'_{nc'}\}$ and $C'' = \{c''_1, c''_2, ...c''_{nc''}\}$ is simply the set of identical constraints (or constraints which differ by a constant for partial locality using blocking).
If the intersection is empty, then locality cannot be obtained for all data. If constraints differ by a constant then partial locality can be obtained by block scheduling. Otherwise, scheduling can either be block or cyclic (though cache effects on shared memory multiprocessors generally favor block scheduling and allocation). If there is more than one constraint, locality can be obtained in more than one dimension provided that the constraints are over disjoint sets of indices.
If there is a constraint between $\mathcal{L}_1$ and $\mathcal{L}_2$, and another between $\mathcal{L}_2$ and $\mathcal{L}_3$ then this implies a constraint between $\mathcal{L}_1$ and $\mathcal{L}_3$ (which must be intersected with any direct constraint between $\mathcal{L}_1$ and $\mathcal{L}_3$).
6. Apply loop distribution when feasible and when it will improve locality: two conflicting constraints are from dependences between loopnests, whose sources (or sinks) within a loop nest have no loop carried dependences, or path of loop carried dependences between them.
7. Determine communication costs for breaking one or more locality constraints. Choose the lowest communication cost constraint to break. On the KSR, communication costs are simply the number of non-local memory references required to transfer data. This can be determined statically, although the expressions are a function of the loop bounds. In general, choose to break communication costs for read-only data first as this can be replicated.

5 Code Generation

After deciding which locality constraints to drop the final step is code generation. Code generation is highly dependent upon the target architecture, and our code generation algorithm is targeted at shared memory multiprocessors such as the KSR-1. There are three key steps in our code generation algorithm:

1. Blocking data structures to cache sizes and coalescing arrays
2. Scheduling loop iterations (mapping iterations to processors)
3. Generating parallel regions

We have applied the above algorithm to a range of benchmarks, including Perfect Benchmarks such as TRFD and FLO52 and NAS benchmarks including Scalar Pentadiagonal. The algorithm is broadly applicable to these benchmarks, excluding benchmarks such as SPICE which use sparse matrices. The discussion below focuses upon TRFD.

5.1 Blocking Data Structures

Blocking array row sizes to a multiple of the cache size is critical to performance on shared memory multiprocessors to avoid false sharing. In general, with large data structures and tiles, false sharing is a relatively small proportion of the overhead. In the TRFD benchmark, the overhead of false sharing was about 6% for the smaller iterations of the outermost scalar loop. This was removed by block aligning all arrays so that their column size was a multiple of the block size ([3] gives a global algorithm and feasibility criteria for global allocation of arrays on block boundaries).

On the NAS benchmark, blocking reduces execution time for a timestep from 2.54sec. to 2.23sec. for 30 processors. This can be done either at this stage or before analysis for locality, together will *array coalescing*. Coalescing takes `k` arrays with the same dimensions `name(N, M, ...)`, accessed using the same subscript expressions in parallel loops, and replaces them by one array `NAME(K, N, M, ... )`. This increases cache reuse and reduces the minimum tile size in the next dimension.

5.2 Scheduling Loop Iterations

In general, the strategy we used for loops with balanced loads was to tile in one or more dimensions (always including the row dimension to avoid false sharing) dividing the dimension(s) equally among the processors, one tile per processor (sized as a multiple of the 128 byte cache line). This calculation is done at runtime. However, in cases of triangular loops load balancing is needed.

Optimizations for Load Balancing Loop nest L4 and L5 in TRFD (Figure 3) present an opportunity for locality optimization and load balancing combined. The access patterns of L4 require a schedule where rows of `X4` are assigned to individual processors. Ideally we want L5 to use the same schedule as L4. The algorithm finds the following constraints in L5:

```
I5 = J5', J5 = I5'
```

These constraints convey that the accesses occur in the upper and lower triangle of the matrix. It is only necessary to schedule the lower or upper triangle. Since we are scheduling a triangular access pattern it is important to consider load balancing. If we follow the schedule of L4 we would not have an even distribution of work when executing L5. Assuming there are P processors, and the the matrix is NxN, each processor would work on W tiles such that lower bound(P/2) $\leq$ W $\leq$ upper bound(P/2). Since each processor is assigned 1 tile on the main diagonal, W tiles are assigned to each processor from the lower triangle. Indirectly, W tiles are assigned from the upper triangle causing each processor to complete 2W + 1 tiles(see figure 4). This schedule guarantees that each processor will work on upper bound(P+1) and lower bound(P-1) tiles. Such optimizations are typically quite important. It often is not sufficient to obtain even locality and parallelism; load balancing must be considered too.

5.3 Generating Parallel Regions

As noted earlier, Barriers need to be inserted between any parallel loops which have a reaching dependence with an unsatisfied locality constraint after parallelization. Algorithms for minimizing the number of barriers have been developed by other authors (e.g., the Kennedy/Callahan algorithm[13]). The Kennedy/Callahan algorithm needs to be modified for our application, as it results in a barrier between any two parallel loops with a reaching dependence (as it assumes no processor to iteration mapping). We need to insert a barrier only if the resulting locality constraint is not satisfied.

The Kennedy/Callahan algorithm uses loop fusion of parallel loops to reduce the number of barriers needed. In our environment such fusion is unnecessary (although it is a minor optimization that reduces loop overhead and may improve cache locality).

The simple algorithm above for barrier insertion does not work when generating one large parallel region for the entire program (with serial sections executed by one processor). The problem is that execution of parallel loops may be control dependent on serial loops. This situation occurs several times in the Perfect benchmarks such as FLO52 (parallel loops within serial loops or branches from serial sections around parallel loops). The approach we have taken is to propagate the branch result outside the serial section. The alternative, small parallel regions rather than one large region, has a significant performance degradation on the KSR-1.

In our application the serial code is always executed by the same processor (processor 0).

The KSR-1 provides prefetch and poststore, which bring issue a non-blocking read/write of a cache line. These both have broadcast capability. The simplest strategy is to insert prefetches several iterations prior to the data being needed (the time for prefetch is about 150 cycles). On the NAS benchmark prefetches improved the performance of a timestep from 2.14sec. to 1.89sec. (about a 10% improvement). The KSR has several limitations on prefetch which prevent much better improvement.

6 Experimental Results

The program TRFD is about 500 lines of Fortran, and has an outer loop which iterates on the problem size which ranges from 10 to 40 in steps of 5. In TRFD there are 4 global shared arrays and seven loop nests within the outer loop that are potentially parallel (ignoring I/O and timing loops). The first step is to determine the available parallelism for each of these loop nests. Each loop nest is parallel in one or more dimensions. Some transformations were done to the program at this stage to increase available parallelism, such as: distribution to make loop nests perfectly nested, eliminating exits from loops, scalar expansion (privatization), eliminating auxiliary induction variables.

The next stage is constructing the PAL graph (figure 3, at the end of the paper).

All of the locality constraints cannot be satisfied, the graph indicates which constraints were selected to be satisfied to minimize communication.

For example, in loop nest L3 to loop L4, the candidate parallel index sets are $\{\mathtt{I}_3\}$ and $\{\mathtt{I}_4, \mathtt{J}_4\}$ respectively

The locality constraints between L3 and L4 are:

$\mathtt{X3}$: $\mathtt{X3}(\mathtt{J}_3, \mathtt{I}_3) \rightarrow \mathtt{X3}(\mathtt{K}_4, \mathtt{I}_4)$; $\mathtt{X3}(\mathtt{K}_3, \mathtt{I}_3) \rightarrow \mathtt{X3}(\mathtt{K}_4, \mathtt{I}_4)$

$\mathtt{X4}$: $\mathtt{X4}(\mathtt{J}_3 \times (\mathtt{J}_3 - 1) + \mathtt{K}_3, \mathtt{MRS}) \rightarrow \mathtt{X4}(\mathtt{I}_4 \times (\mathtt{I}_4 - 1) + \mathtt{J}_4, \mathtt{MRS})$

Equating subscript expressions for the constraints on $\mathtt{X3}$:
$\mathtt{J}_3 = \mathtt{K}_4 \;\; \mathtt{I}_3 = \mathtt{I}_4$ and $\mathtt{K}_3 = \mathtt{K}_4 \;\; \mathtt{I}_3 = \mathtt{I}_4$
leads to satisfying the locality constraint with $\mathtt{I}_3$ and $\mathtt{I}_4$ parallel, and the mapping $\mathtt{I}_3 \rightarrow \mathtt{I}_4$.

The non-linear constraint on $\mathtt{X4}$ cannot be satisfied.

The locality constraints between loop nest L4 and L6 and L1, for $\mathtt{X1}$ and $\mathtt{X3}$, leads to $\mathtt{J}_4$ being dropped from the candidate parallel index set for L4. Parallelizing, and hence distributing data, on $\mathtt{I}_4$, leads to locality for $\mathtt{X3}$, $\mathtt{X2}$, and $\mathtt{X1}$.

6.1 Performance

The table below summarizes the results obtained for a 10 processor KSR-1 for the last iteration of the outer scalar loop (largest problem size: N = 40). The result scale for more or fewer processors. All times are in secs., measured using KSR *pmon* timing routines on dedicated processors. The times given are for each of the 7 parallel loop nests and the entire program.

Loop Nest	Serial Time	Tiled Time	Parallel Time	(Min Pi time)
		10 proc.	10 proc.	10 proc.
L1	0.01	0.03	0.00	0.00
L2	2.15	2.83	0.37	0.30
L3	75.50	47.80	7.47	7.43
L4	42.06	6.66	7.87	0.72
L5	1.41	0.46	0.26	0.20
L6	28.50	7.80	4.29	1.54
L7	21.17	15.40	2.92	0.56
Total	170.80	81.00	23.20	

The "tiled time" is the execution time using KSR `TILE` directives to parallelize each loop nest individually. This generally gives poor performance for across multiple loop nests because their is no control of the allocation of processors to tiles (hence locality is lost). KSR has a directive to allow sharing of schedules across loop nests (`AFFINITY REGIONS`), but we have found it of limited generality, and not as effective as explicit `PARALLEL REGIONS` and allocation of tiles to processors based upon processor ids.

For comparison, we parallelized the program using KSR KAP preprocessor/compiler. KAP can only generate `TILE` directives, and does no optimization for locality across loops. In addition, KAP uses very simple strategy to decide tile sizes and which loop nest to parallelize. On a KSR-2, TRFD took 69 sec. on 1 processor; when parallelized by KAP it took 102.5 sec. on 5 processors and 102 sec. on 10 processors.

Overall, our speedup for the parallel version with 10 processors was about 7.5. Some loops have poor load balancing, as illustrated by the column "Min Pi time", which gives the minimum execution time of the loop for each processor. We use static tiling of the iteration space, although our approach does not require static distribution of data. However, for several loops, particularly L4 and L7, there is an uneven workload distribution. The reason for this is triangular iteration space.

We also applied the algorithm to the NAS SP (Scalar Pentadiagonal) benchmark, which is a somewhat simpler benchmark with three parallel loop nests with communication between. In this case the scalar execution time for one time step was 35sec. With 31 processors, after communication optimizations described below, the time for one time step was 1.14 sec. (very close to linear speedup).

For the KSR-1, there is no need for explicit replication. In the benchmark example, the array `X1` is implicitly replicated. Loop distribution and fusion are complementary. Loop fusion can be done as a pre or post pass. Loop distribution only improves potential locality when there are no dependences between the two loops that result from distribution (otherwise a locality constraint on the loop prior to distribution will be transformed into a locality constraint between the two loops after distribution).

7 Comparison with Other Work

Traditionally the problem of transforming a program to obtain parallelism and locality of reference is viewed as two independent subproblems. First, transform the program to expose the parallelism in the program. Next, analyze the array accesses in the parallel loops to determine the data and computation distributions that optimize the locality of reference (PARADIGM[7], and ADG[6]). But the optimality can only be obtained if subproblems are treated together[1]. Our approach brings both program transformations for parallel and determining distributions of data and computation to processors into a single framework.

The closest related work is that of Anderson and Lam[1]. They use explicit data distributions and iteration to processor mappings represented as linear transformation matrices. Unlike our approach, they rely upon iteration space transformations rather than initially using statement transformations. Our representation of partitions is more precise than theirs (which relies on basis vectors). However, in many cases both approaches would derive the same mappings. In addition, they use a communication graph corresponding to the PAL graph above. However, their communication graph is less precise (does not include dependences or constraints).

Other related work includes PARADIGM[7]. Their model is based upon explicit data distribution, and mesh decomposition. They use constraint solution, and do not have an explicit phase for identifying available parallelism. Other authors use graph representations for finding data distributions that optimize communication for a parallel program. The Alignment-Distribution Graph (ADG[6]) is similar in spirit to our PAL Graph. ADG is based on dataflow graph of a single assignment form of a given program, where as PAL graph is constructed by representing each loop nest as a node and dependences between the loop nests as edges labeled by constraints.

8 Conclusion & Future Work

We have presented a framework for analyzing for both parallelism and locality for distributed memory systems which is independent of any particular architecture. It is able to represent both locality and parallelism without relying on explicit data distributions or already parallelized programs. We have found that the PAL graph captures very succinctly both the available parallelism and locality. By itself, it is a useful program representation for programmers as well as program analysis.

Constraints on locality can be combined to determine global locality. We have applied the model to a range of benchmarks successfully by hand, are investigating implementing the algorithms as part of PAT, a tool for interactive parallelization. Current work is focusing upon a performance estimation model for the KSR to guide transformations and tiling strategies, and applying the algorithm to further benchmarks.

References

1. 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.
2. Appelbe, B., Doddapaneni, S., Hardnett, C., and Smith, K. Determining transformation sequences for loop parallelization. Tech. Rep. GIT-ICS-92/59, Georgia Institute of Technology, Nov. 1992.
3. Appelbe, B., Hardnett, C., and Doddapaneni, S. Aligning data structures for parallelism and locality. Tech. Rep. GIT-CC-94-20, Georgia Institute of Technology, Feb. 1994.
4. Appelbe, B., and Smith, K. Determining transformation sequences for loop parallelization. In *Fifth Workshop on Languages and Compilers for Parallel Computing* (July 1993).
5. Callahan, D. *A Global Approach to Detection of Parallelism.* PhD thesis, Rice University, 1987. Rice Tech Report, COMP TR87-50.
6. Chatterjee, S., Gilbert, J., and Schreiber, R. The alignment-distribution graph. In *Sixth Workshop on Languages and Compilers for Parallel Computing* (July 1993), pp. 234–252.
7. Gupta, M., and Banerjee, P. Paradigm: A compiler for automatic data distribution on multicomputers. In *International Conference on Supercomputing* (June 1993), pp. 87–96.
8. Haghighat, M., and Polychronopoulos, C. Symbolic analysis for parallelizing compilers. Tech. rep., University of Illinois, 1994.
9. Kennedy, K., and McKinley, K. Optimizing for parallelism and data locality. In *International Conference on Supercomputing* (July 1992), pp. 323–334.
10. Ramanujam, J., and Sadayappan, P. Tiling multidimensional iteration spaces for nonshared memory machines. In *Supecomputing '91* (Nov. 1991), pp. 111–121.
11. Wolf, M. E., and Lam, M. S. A data locality optimizing algorithm. In *SIGPLAN Programming Language Design and Implementation* (1991), pp. 30–44.
12. Wolf, M. E., and Lam, M. S. A loop transformation theory and an algorithm to maximize parallelism. *IEEE Transactions on Parallel and Distributed Systems* *2*, 4 (October 1991), 452–482.
13. Zima, H., and Chapman, B. *Supercompilers for Parallel and Vector Computers.* ACM Press, New York, New York, 1990.

This article was processed using the LaTeX macro package with LLNCS style

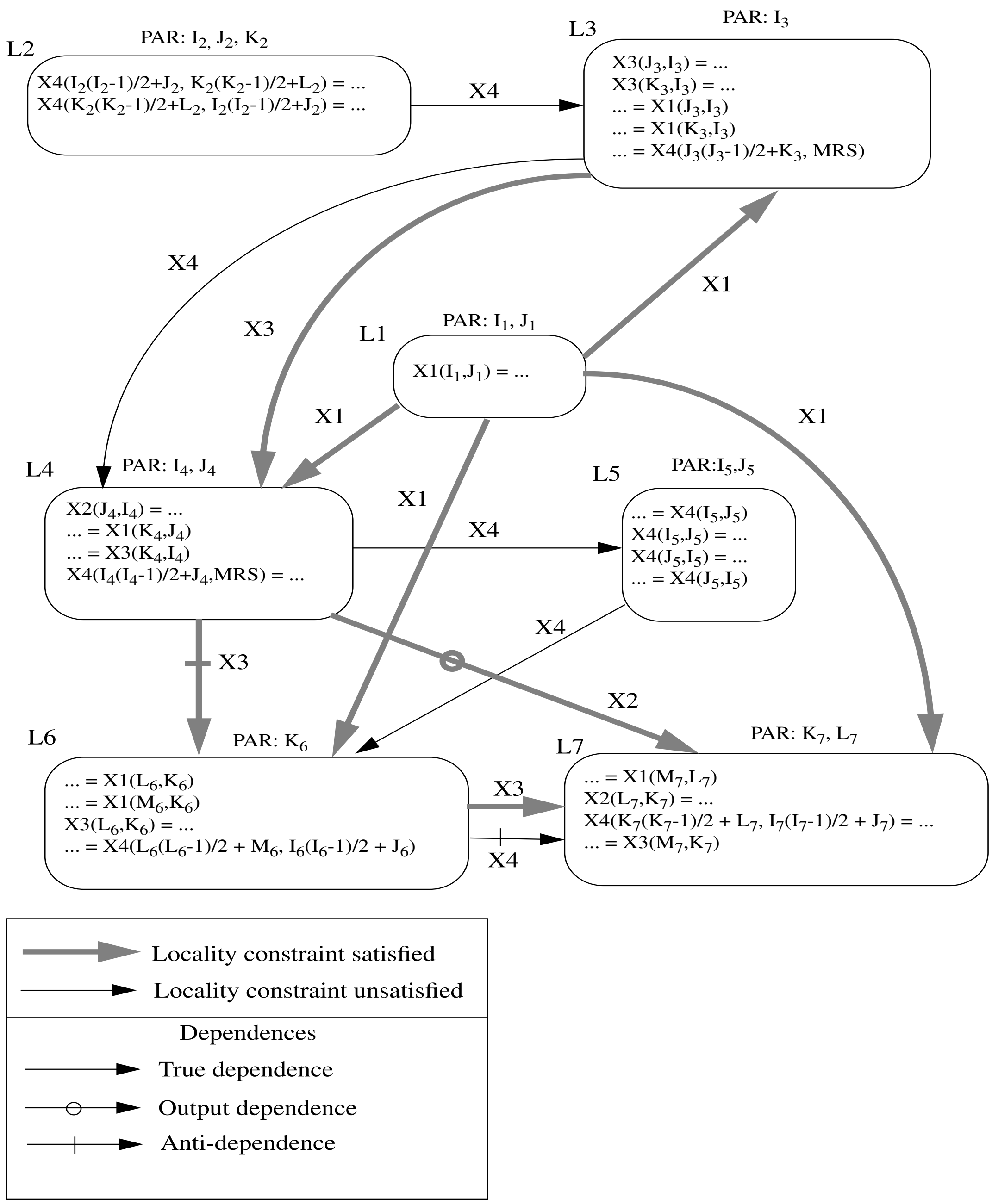

Figure 3: PAL Graph for TRFD Perfect Benchmark