1990

Array Reshaping -- A Mechanism for Optimizing Array Storage on Parallel Architecture

Ko-Yang Wang

Report Number:
90-1042

ARRAY RESHAPING - A MECHANISM FOR OPTIMIZING ARRAY STORAGE ON PARALLEL ARCHITECTURES

Ko-Yang Wang

CSD-TR-1042
November 1990

Array Reshaping - A Mechanism for Optimizing Array Storage On Parallel Architectures.

Ko-Yang Wang

Computing About Physical Objects
Department of Computer Sciences
Purdue University, West Lafayette, IN 47907

ABSTRACT

In this paper, we discuss a program transformation technique called *array reshaping*. Array reshaping is an aggressive program transformation technique that modifies the *shape* of the array storage at compile-time for efficient execution on parallel architectures. It can be used to minimize the local storage for local copies of data or change the patterns that arrays are stored and referenced. For shared-memory architectures, it can be used to copy subsets of arrays to local memories in blocks to speed up the data references. For distributed-memory architectures, this technique can be used to minimize communication between processors and optimize the data storage. Array reshaping also eliminates some of the needs to hand optimize the storage space and thus allows the user to specify algorithms in a more intuitive way. Array reshaping functions and some examples and heuristics for utilizing the array reshaping are discussed.

December 12, 1990

Array Reshaping - A Mechanism for Optimizing Array Storage On Parallel Architectures.

Ko-Yang Wang

Computing About Physical Objects
Department of Computer Sciences
Purdue University, West Lafayette, IN 47907

1. Introduction

In this paper, we introduce a new program transformation technique called *array reshaping*. Array reshaping is a program transformation technique to modify the storage pattern of an array (called the *shape* of the array). The shape of an array is the way the elements of the array are physically stored and is defined by the declaration of the array. This transformation is not to be confused with some existing transformations such as index shifting, loop skewing, linearization, that change the *view* of an array. A *view* of an array is an index set of the elements of the array and is a function defined by subscripts of an array reference. An array can have many views but only one shape. When an array reference appears inside loops, the subscripts of the array reference are functions of indices of the loops. Program transformations can change the view of an array by changing array subscripts or loop indices. For example, an n by n tri-diagonal matrix with only three non-zero diagonal elements has a square shape of size n by n. By altering the subscripts of the array reference in loops, the view of the array can be changed and only the elements that are on the tri-digonal need to be referenced. However, the change of the views does not affect the physical pattern of the array storage. If the array reshaping (as shown in the example in the next section) is applied on the array, the shape of the array may be changed into a filled array with only non-zero elements of the original array, i.e. the array has a rectangular shape of size n by 3.

Definition. The *shape* of an array a can be characterized by bounds of the array and can be defined as:

$$shape\,(a) = [l_1 .. u_1] \times [l_2 .. u_2] \times \ldots \times [l_L .. u_L].$$

where l_k and u_k are the lower and upper bounds of the k-th subscripts of the array. The *size* of the array is then defined to be:

$$Size(a) = \prod_{i=1}^{L} (u_i - l_i).$$

By nature, the array reshaping is a global program transformation that can be applied only after global analysis of the program to determine which shape and storage pattern of the array can yield the best program performance or minimal program storage. It also needs to be checked that whether any elements that are thrown out by the reshaped array are used anywhere in the original program. These can be analyzed based on the define-use chain of the program dependence graph [FeOtWa83] and a program performance predication model that we proposed in [Wang90a]. Another important usage of the array reshaping is to identify the portions of arrays that are used by certain part of the program and make local copy of these portions. This means that instead of replacing the original array, the reshaped array is placed in the local memory of a processor and the code is generated to move the data from the original array to the new array.

2. Array Reshaping Functions

Array reshaping represents a group of one-to-one functions that operate on the shapes of arrays. A reshaping function defines how the storage of an array is to be changed as well as the relation between the elements of the original and the generated array. There are two kinds of array reshaping that we are particular interested in: truncation/extension and linear transformation. The truncation changes the lower and upper bounds of the subscripts by removing or adding spaces to the arrays. The extension extends the boundaries of an array beyond its original bounds by enlarging the bounds of its subscripts. The linear transformation applies linear functions to the subscripts of the array and changes the shape of the array. The difference between these two types of reshaping functions is that for the linear transformation, the same linear function is applied to both subscripts of the elements and bounds of the array, while for the truncation or extension, only the bounds of the array are changed (an identity function is applied to the subscripts of the elements). A truncation or extension is often accompanied by a linear transformation function.

A truncating or extending array-reshaping function (represented as $[l..u]_i$) that changes the bounds of the i-th subscript of an array a into [l..u], changes the shape of the array a from $[l_1 .. u_1]$ x ... x $[l_i .. u_i]$ x ... x $[l_L .. u_L]$ into $[l_1 .. u_1]$ x ... x $[l .. u]$ x ... x $[l_L .. u_L]$.

A linear reshaping function $(f_1(I), f_2(I), \cdots, f_L(I))$ maps the element $a(i_1, i_2, \cdots, i_L)$ in the array a into $a^r(f_1(i_1,i_2,\cdots,i_L), f_2(i_1,i_2,\cdots,i_L), ..., f_L(i_1,i_2,\cdots,i_L))$, where a^r denotes the new array. And the shape of the array a is mapped from

$$[l_1 .. u_1] \text{ x } [l_2 .. u_2] \text{ x ... x } [l_L .. u_L]$$

into

$$[f_1(\overline{l_1}) .. f_1(\overline{u_1})] \text{ x } [f_2(\overline{l_2}) .. f_2(\overline{u_2})] \text{ x ... x } [f_L(\overline{l_L}) .. f_L(\overline{u_L})]$$

where $\overline{x_i}$ denotes a vector whose entries are all zero except the i-th entry, which has the value x. For convenience, the linear array reshaping function is denoted as:

$$I \text{ --> } (f_1(I), f_2(I), \cdots, f_L(I)).$$

3. Variations of Array Reshaping

Although the array reshaping can be used to change the shape of array into many different shapes, the primary purpose is to reduce the size of the array. Only elements whose images of the reshaping function fall into the new shape of the array have slots in the new array. Some basic reshaping functions are listed as follows:

- *Projection: (I, j) --> (I) or (i, I) --> (I).*

  This function can be used when the interval corresponding to the dropped subscript in the shape of the array is trivial -- has only one constant in the interval. If one subscript of an array a in a for loop is loop invariant, then the array can be projected into a lower dimensional array inside the loop. For two dimensional arrays, the projection (i,j) --> (i) maps the original array into the i-th row and the projection (i,j) --> (j) maps the original array into the j-th column. The projection can be viewed as a special case of truncation; when the range of a subscript of the array is truncated into only one integer, the corresponding dimension of the array can be dropped in the

task.

- *Transporting: (i,j) --> (j,i).*

  Two subscripts of the array are exchanged. In this case, an array of shape [1..N]x[1..M] will be changed into an array of shape [1..M]x[1..N].

- *Compaction: A linear function that can compact the bounds of the array subscript is called compacting function. For example, (i) -> (i/2) reduces the size of the array in half.*

- *Expansion: A linear function that expand an array into a larger array is called an expansion function. For example, (i) --> (2*i) doubles the size of the array. The expansion is usually applied on arrays that were compacted to map them back to the original arrays.*

  We denote the compaction and expansion as:

$$a^l \leftarrow ((i,j) \longrightarrow (f_1(i,j), f_2(i,j)))(a)$$

$$\text{and } ((i,j) \longrightarrow (f_1{}^{-1}(i,j), f_2{}^{-1}(i,j)))\ (a^l) \leftarrow a$$

  respectively. Where f_i^{-1} denotes the inverse function of f_i.

  To reduce the data transmitting cost in distributed systems, for a compacting reshape, the compacting is usually done before the data is sent; and for expanding reshape, the expansion is usually done at the receiving processor.

Integer linear functions A integer linear function is a linear function whose cofficients are all integers. A general form for the linear functions is $\sum_{k=1}^{n} a_k * i_k$, where a_ks are integer cofficients and i_ks are the indices of the array subscripts.

4. Opportunities For Applying Array Reshaping

"How can array reshaping be used in program restructuring to improve the parallelism of a user program on a target architecture?" Array reshaping can be used to copy the data into local memory of a processor or change the storage pattern of an array to reduce space or reference time. The latter includes simplifying array subscript-calculation, improving cache-hit ratio, changing array strides without interchanging the loops and

reducing unnecessary traffic on the network. We list a few cases here and study these cases through examples.

Case 1. Minimizing the array storage.

For example, given a band matrix a of size n by n with width w declared as:

a: array [1 .. n, 1 .. n] of real;

The matrix uses n^2 spaces. By reshaping the band matrix into a rectangular array of size n by $2*w+1$, the storage requirement is decrease to $n*(2*w+1)$. This situation can be recognized by observing that the second subscript j of the array reference $a(i,j)$ is always bounded by $i-w$ and $i+w$. By transferring the array based on the function (i,j) --> $(i, j-i)$, the array a is mapped into a new array a' with the elements being repositioned. And the declaration of the array becomes:

a: array [1 .. n-1, -w .. w] of real;

Case 2. Changing the physical reference order of the array to improve performance.

For example, transporting a vectorizable array that has a long array stride but happen to be in a pair of non-interchangeable loops may yield a stride-1 vector. Even if the loop is interchangeable, we still have case where in a doubly-nested loop we have two references to two arrays for which one has stride-n reference and another has stride-1 reference. By interchanging the loops, we would change one references into stride-1 and another into stride-m.

The reshaping function can also be compounded with other reshaping functions to change both the view and shape of the array.

For example, for the example in case 1, if the array is better transported (for it to be vectorized or other purposes) for other parts of the program, the transporting function (i,j) --> (j,i) can be applied. By applying (i,j) --> (j,i) to the above example we obtain a combined reshaping function (i,j) --> $(j-i,i)$, and the array declaration becomes

a: array [-w .. w, 1 .. n-1] of real;

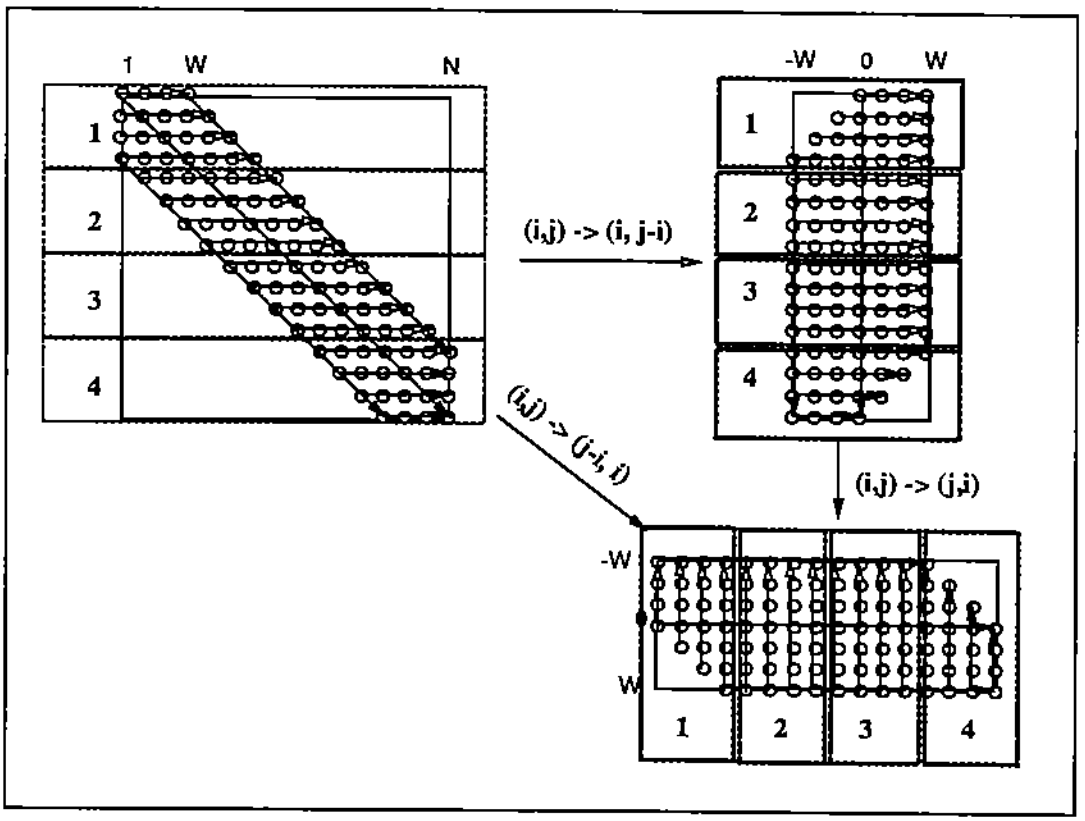

Figure 1. Reshaping a band matrix.

Case 3. Minimizing messages in distributed-memory systems.

The data communication between processors of distributed-memory systems can be significantly reduced when the sub-array that is used is compacted and copied over. In case the sub-array is modified, the part of the array that are modified can be stored back to the original processor by transmitting the results back and expanded into the original array.

For example, suppose array *a* is stored in process *i* but used by processor *j* in the following loop:

```
for i := 0 .. P-1 do (* parallelized loop *)
    for k := 0 .. M-1 do
        b[i, k] := a[i-1, 2*k];
        ....
    end for
end for
```

In a distributed-memory system, the reference to array a will be replaced by a copy of the array a as shown below:

```
forall i := 0 .. P-1 do
    local aloc: array [ 0 .. M-1 ] of real;
    if (i != 0) aloc[0..M-1] <- a[i-1, 0..M-1];
    for k := 0 .. M-1 do
        b[i, k] := aloc[2*k];
    end for
    ....
end forall
```

Then array a has to be sent from processor i to processor j. By reshaping a', the copy of the array a in the processor j, with the reshaping function $i \longrightarrow (i/2)$ on the local array *aloc*, the program becomes:

```
forall i := 0 .. P-1 do
    local aloc: array [ 0 .. M/2 ] of real;
    if (i != 0) aloc[0..M/2] <- (a[i+1, 2*j], j=0..M/2);
    for k := 0 .. M-1 do
        b[i, k] := aloc[k];
    end for
end forall
```

As can be seen in the example, the conversion is done in the sender and the size of the array that is sent through the network is halved and the array subscript computation is also simplified.

Note that this transformation is based on the specific shape and usage of the array, numerical analysts have been doing this by hand by decades. However, this hand optimization usually obscures the clarity of the algorithm and sometimes it also creates difficulties for the compiler in optimizing the program. By having the compiler perform this kinds of optimizations automatically, it not only eases the burden on the programmers but also makes the optimization easier on the compilers. This point is made even more clear in the following example:

Case 4. Avoiding obscured algorithms due to optimization.

Excessive program optimization often leads to obscured algorithms. For example, the band matrix factorization uses Gaussian elimination to factor the array. The following program is a direct coding of the Gaussian elimination with the additional knowledge of knowing that array *a* is a band matrix.

```
a: array [1..n, 1..n] of real;
for i in 1 .. n-w do
    for j in i+1 .. i+w do
        a[j,i] := - a[j,i] / a[i,i];
        for k := i+1 to i+w do
            a[j,k] := a[j,k] + a[j,i] * a[i,k];
        end for
    end for
end for
```

For large *n*, most spaces in array *a* are unused and wasted. In order to save spaces, a "competent" programmer would implement the above algorithm into the following form:

```
a: array [1..n, -w..w] of real;
for i in 1 .. n-w do
    for j in 1 .. w do
        a[j+i,-j] := - a[j+i,-j] / a[i,0];
        for k := 1 to w do
            a[j+i,k-j] := a[j+i,k-j] + a[j+i,-j] * a[i,k];
        end for
    end for
end for
```

Unfortunately, this program is so obscured that most people have to spend quite a bit of time to comprehend the meaning of the subscribes and the program. By examining the above example more carefully, we will find that the second program can be obtained by shifting the loop indices of loop j and k by i and then applying the reshaping function $(m,o) \dashrightarrow (m,o-m)$ to the first program. This optimization can be obtained by a simple

heuristic encoded as rules in the rule base. By letting the compiler optimize the storage usage, the program can be implemented as close to the algorithm as possible.

Case 5. Array Copying for the Functional Semantic of Forall Loops.

For languages whose *forall* loops have functional semantics, copy-arrays may need to be created in each loop instance that uses or updates the array to preserve the copy-in and copy-out semantic [Wang86]. The array reshaping can be used to reshape the local copy-array of the original array for each loop instance. Since memory accesses inside for loops are usually very regular, this means that the array reshaping can reduce the size of the copy-array to the minimum. This will guarantee that only the necessary data is copied into the forall loops. On distributed systems, only the remote array elements that are used by the local processors need to be copied. This minimizes the cost of copying arrays in the implementation of functional FORALL loops and makes it practical.

For example, consider the Jacobi iteration shown in the next program fragment:

```
var
    A, New_A: array[0..N, 0..N] of real;
    pid: integer;

forall i in 1 .. N-1, j in 1 .. N-1 do
    New_A[i,j] := 0.25 * (A[i-1,j] + A[i+1,j] + A[i,j-1] + A[i,j+1]);
end forall;
```

Note that this forall loop is usually surrounded by an outer loop that iterates until the solution converges. At the end of the iteration there is code to copy contents of the array New_A into A (or an optimization-minded user might interchange the array New_A with array A to avoid the copying). For shared memory architecture, the array A and New_A can be in global memory, but it is beneficial to create local copy of the array. The local copies of the array Old_A are block-transferred to the processors before the first iteration and the results are copied back after the last iteration. This model is similar to the distributed-memory model. To execute this program on a distributed-memory machine, one simple decomposition is to partition the arrays into P^2 blocks of size MxM. Where $M = (N + 2) / P$. This yields the following program:

```
forall k in 0 .. P-1, l in 0 .. P-1 do
    const
        low1 = k*M; low2 = l*M;
        high1 = min(low1+M-1, N+1); high2 = min(low2+M-1, N+1);
    var
        Old_A, New_A: array [low1-1..high1+1, low2-1..high2+1] of real;
    DATAMOVEMENT();
    for i in 1 .. M, j in 1..M do
        New_A[i,j] := 0.25 * (Old_A[i-1,j] + Old_A[i+1,j] +
            Old_A[i,j-1] + Old_A[i,j+1]);
    end for
end forall
```

In the above program, the array Old_A is the reshaped array of the original array A that the iteration uses for each process. There is an overlapping area of the array with processors that compute the adjacent blocks. If we only look at the forall loop, the array New_A should be declared as follow:

```
New_A: array [low1..high1, low2..high2] of real;
```

However, since the array Old_A is copied into (or switched with) array New_A, after the forall loop, the boundaries of the array New_A is extended (array reshaping replaced the above declaration by the extended array). Based on the dependence analysis of the original program, the statement DATAMOVEMENT() represents the code to move data into the local processor at the first iteration of the outer loop of the forall loop and moving data from adjacent processors and copying inside the processor in the subsequent iterations. This simple example actually represents a very sophisticated process of program restructuring.

5. Heuristics for Applying Array Reshaping

How does a compiler recognize opportunities for applying the array reshaping? When is array reshaping beneficial? Since the array reshaping changes the declaration of the arrays, it is only applicable when all the expressions involved are resolvable at compile

time. In the last section, we listed some opportunities for applying the array reshaping. Here we list some simple heuristics that a parallel compiler can use to decide when to apply the array reshaping.

1. If a subscript of an array is a constant for all references of that array inside a task, then the projection can be applied to the array to map the array to a new array with the subscript dropped. Inside loops, the precondition means that the said subscript of all references to the array is loop invariant.

2. If the range of one subscript in all references of the array is a subset of the bounds of that subscript then the shape of the array can be shrunk by truncation to truncate the bounds of that subscript into the actual range.

3. If a subscript of the array is a multiple of a loop index by an integer, such as a*i, then the array can be compacted by the array reshaping function i/a in that subscript. This heuristic was used in the last example.

4. If the expression of a subscript appears in another subscript of the array, this expression can be eliminated by array reshaping. For example, for the program in the example in case 4 in the last section, the index of loop i appears in the loop bounds of loops j and k. Consequently, array references in the loop that use indices j and k are determined by the value of i implicitly. Just to see the relations more explicitly, we apply the transformation index shifting to the original band matrix factorization program shown in the above example and we get:

```
for i in 1 .. n-w do
    for j in 1 .. w do
        a[j+i,i] := - a[j+i,i] / a[i,i];
        for k := 1 to w do
            a[j+i,k+i] := a[j+i,k+i] + a[j+i,i] * a[i,k+i];
        end for
    end for
end for
```

From the program it is clear that the index *i* appears in every subscript of every reference. For each loop instance of loop *i*, *i* is a constant for the loop instance.

Therefore, the shape of the array a in the loop instance is:

$$[i+1..i+w]\ x\ [i] \cup [i]\ x\ [i] \cup [i+1..i+w]\ x\ [i+1..i+w] \cup [i]\ x\ [i+1..i+w]$$

$$= [i+1..i+w]\ x\ [i+1..i+w].$$

The shape of all references for a in the loop is [i+1..i+w]x[i+1..i+w] where $1 \le i \le n$. This implies that the array references of a in the loop all fall into a band of width w. Our heuristic says that by applying the function (i,j) --> $(i,j-i)$ to the subscripts of the array reference, the shape becomes [i+1..i+w]x[-w..w]. And for loop i, the shape of the array becomes [1..n]x[w..-w], achieving a reduction of (n-2*w-1)*n in space. The second subscript in the new array represents the distance of the element to the diagonal in the original array.

6. Closing Remarks

The methodologies that we described in this paper can serve as a starting point for studying the array reshaping for program optimization. Array reshaping is a powerful but complicated program transformation technique. It is particularly useful to minimize data communication cost for architectures that has non-trivial data transmission cost and programs that only utilize parts of the arrays. It can also be used to minimize data storage for machines that have limited memory available (such as the hypercube computers). Since array reshaping has a significant effects on the data references and communication cost, it usage should be carefully planned to avoid disastrous counter-effects. The effects of the array reshaping on the performance of the program depends on the archtecture features of the target machine [Wang89] and can be estimated by a performance prediction model [Wang90a]. This warrants a more thoroughly study about the potential benefit of array reshaping in optimizing programs for parallel machines - especially when combined with other program transformation techniques.

In [Wang90b] we studied a program transformation technique call message consolation. Array reshaping can be combined with message consolation to minimize communication cost on distribute-memory architectures. Both techniques are utilized in an intelligent, optimizing parallel compiler [WaGa89, Wang90c] for different parallel computers. The selection of the shapes of an array is guided by some heuristics and a performance

prediction model that bases on machine features. We are currently conducting some experiments with different heuristics to test the effects of the technique on distributed-memory parallel architectures and will report our further findings when we revised this paper later.

7. REFERENCES

[BuCy86] M. Burke, R. Cytron, "Interprocedural Dependence Analysis and Parallelization," *SIGPLAN symposium on Compiler Construction,* 1986, 613-641.

[FeOtWa83] J. Ferrante, K. Ottenstein, J. Warren, "The Program Dependence Graph and Its Uses in Optimization," IBM Technical Report RC 10543, Aug. 1983.

[Wang86] K. Wang, "Array Proection and Copying Issues in the Blaze and E-Blaze Implementation," Internal memo, the RP3 group, IBM, Aug. 1986.

[Wang89] K. Wang, "Machine Knowledge Representation and Manipulation For Parallel Compilers," Technical Report CSD-TR-843, Department of Computer Sciences, Purdue University, Jan. 1989.

[WaGa89] K. Wang and D. Gannon, "Applying AI Techniques to Program Optimizations For Parallel Computers," in K. Hwang and D. DeGroot, editors, *Parallel Processing for Supercomputers and Artificial Intelligence,* McGraw-Hill, 1989, pp. 441-485.

[Wang90a] K. Wang, "A Performance Prediction Model For Parallel Compilers," Tech. Report, CSD-TR-1041, CER-90-43, Department of Computer Science, Purdue University, Nov. 1990.

[Wang90b] K. Wang, "Managing Data Synchronization Automatically For Distributed-Memory Architectures," Tech. Report, CSD-TR-1043, CER-90-45, Department of Computer Science, Purdue University, Nov. 1990.

[Wang90c] K. Wang, "A framework For Intelligent Parallel Compilers," Tech. Report, CSD-TR-1044, Department of Computer Science, Purdue University, Nov. 1990.