Sixth Annual Workshop on Languages and Compilers for Parallel Computing, Vol 768 of Lecture Notes on Computer Science, pp 169-183, Portland, OR, Aug 1993, Springer Verlag

Do&Merge: Integrating Parallel Loops and Reductions

Bwolen Yang, Jon Webb, James M. Stichnoth,
David R. O'Hallaron, and Thomas Gross

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213-3891

Abstract

Many computations perform operations that match this pattern: first, a loop iterates over an input array, producing an array of (partial) results. The loop iterations are independent of each other and can be done in parallel. Second, a reduction operation combines the elements of the partial result array to produce the single final result. We call these two steps a Do&Merge computation. The most common way to effectively parallelize such a computation is for the programmer to apply a DOALL operation across the input array, and then to apply a reduction operator to the partial results. We show that combining the Do phase and the Merge phase into a single Do&Merge computation can lead to improved execution time and memory usage. In this paper we describe a simple and efficient construct (called the *Pdo* loop) that is included in an experimental HPF-like compiler for private-memory parallel systems.

1 Introduction

Suppose we wish to compute the histogram of a two-dimensional image. To perform this computation, we can independently compute a partial histogram from each row, and then apply the summation operator across the partial histograms, yielding the complete histogram. In this example, we are applying a *conversion function* independently across an input array, and then applying an associative *merge operator* to the results, yielding a single global result. This kind of computation, which we refer to as a *Do&Merge*, arises frequently. For example, in the domain of image processing, such applications as edge detection

Supported in part by the Advanced Research Projects Agency, Information Science and Technology Office, under the title "Research on Parallel Computing," ARPA Order No. 7330. Work furnished in connection with this research is provided under prime contract MDA972–90–C–0035 issued by ARPA/CMO to Carnegie Mellon University, and in part by the Air Force Office of Scientific Research under Contract F49620–92–J–0131. This material is based in part upon work supported under a National Science Foundation Graduate Research Fellowship.

and connected components can be solved using this paradigm. In the signal processing domain, we see this pattern whenever narrow-band data needs to be combined to yield broad-band data.

In most parallel programming languages, the two component operations are explicitly separate: the programmer first specifies the conversion function to be applied across the input in parallel, and then specifies the merge operator. We point out the benefit of not separating the two operations; often the programmer can combine them into a single function that is more efficient to execute.

In the remainder of this paper, we describe the semantics of the Do&Merge computation and we describe a construct, called the *Pdo* loop, for expressing the Do&Merge. Section 2 describes the basic Do&Merge semantics and describes two existing constructs for expressing the Do&Merge operation. Section 3 describes the *Pdo* loop, which allows the programmer to combine the Do&Merge operators in a more efficient manner, resulting in improved execution time and memory usage. In Section 4 we describe how a compiler can simply and efficiently compile a *Pdo* loop. Finally, Section 5 formulates several problems as Do&Merge computations and compares the *Pdo* performance with that of existing methods. We use simple example problems to help get the basic idea across. However, for many important problems (e.g., connected components) the conversion function and merge operator are complex user-defined functions.

2 Do&Merge operation

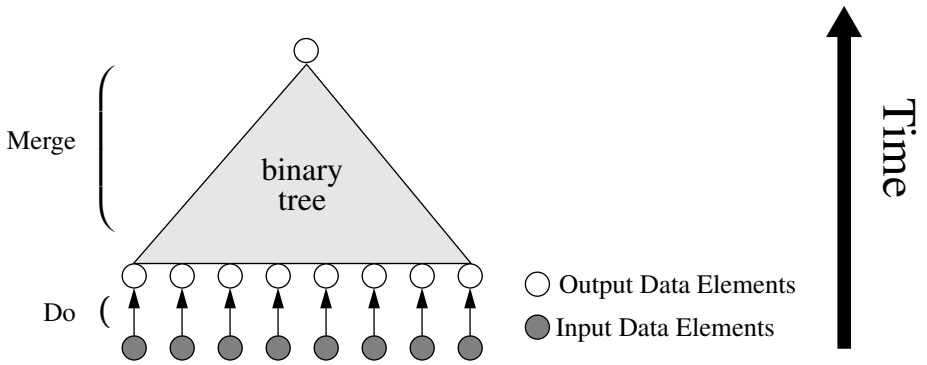

Figure 1: The Do&Merge operation.

The Do&Merge operation is illustrated in Figure 1. Conceptually, a conversion function F is applied in parallel to each of the N input elements, to form a set of partial results. Then an associative merge operator C is applied $N-1$ times to produce the final result. The order of application of C implicitly forms a binary tree; any binary tree produces a valid result. The C operator is associative, but not necessarily commutative, with a left identity I. Formally, we have the following:

- $F : \beta \rightarrow \alpha$
- $C : \alpha \times \alpha \rightarrow \alpha$
- $C(I, \alpha) = \alpha$

There are many ways in which the Do&Merge can be implemented. Two such methods available in HPF [3] are the following:

`result` $= I$
`DO` $k = 1, N$
  `result` $= C($`result`$, F($`input`$[k]))$
`END DO`
	`FORALL` $(k = 1 : N)$
	  `partial`$[k] = F($`input`$[k])$
	`END FORALL`
	`result` $=$ `REDUCE`$(C,$`partial`$)$

Do implementation	*Forall* implementation

These two implementations of the Do&Merge differ only in the structure of the merge tree that is created. The merge tree structures are pictured in Figure 2 for $P = 2$ processors. The thick arrows represent interprocessor communication. Notice that the *Do* implementation is forced to run sequentially due to data dependences; these data dependences are mostly artificial, due to the associativity of C, although in practice, this may be difficult to determine for a compiler.

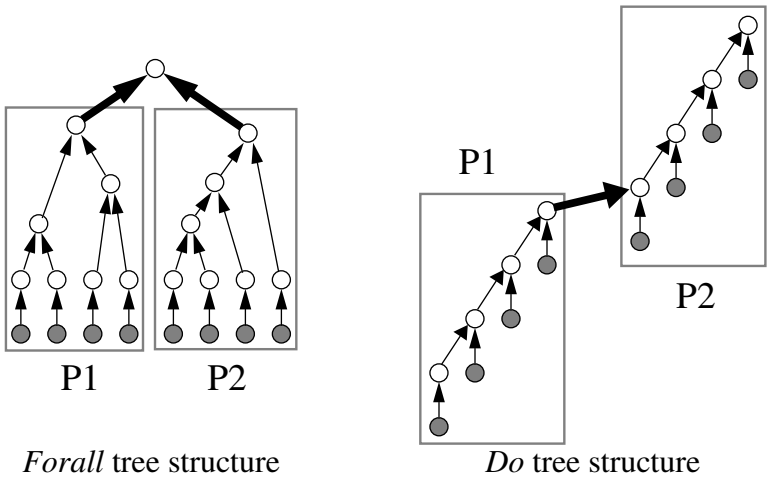

Figure 2: Tree structures of two possible Do&Merge implementations.

We can analyze the running time of each of these implementations. Let $T(F)$ be the time required to execute F, $T(C)$ the time required to execute C, P the number of processors, and $N \gg P$. Also assume that the data is distributed across the processors in *block* fashion (i.e., a single, disjoint, contiguous block of the input data resides on each processor). The total running time for the *Do* implementation is $N \times (T(F) + T(C))$. Notice that the running time is the same regardless of P, due to the sequential nature of the *Do* loop.

The *Forall* version, on the other hand, is more clever. Since N/P input elements reside on each processor, the FORALL body executes in time $(N/P)(T(F))$. The optimal REDUCE function will apply C locally on each processor to yield one partial result per processor, in time $(N/P)(T(C))$, and then merge the partial results across the processors in time $T(C)\log P$, for a total execution time of:

$$(N/P)(T(F)+T(C))+T(C)\log P. \tag{1}$$

Another possible Do&Merge implementation is the DOACROSS [2] loop, which allows F to proceed in parallel but still sequentializes the application of C. Yet another possibility is the DOANY [7] construct, which requires C to be both associative and commutative, allowing the loop to be sequentialized in any way. However, neither construct fully integrates the Do and the non-commutative Merge to allow maximum efficiency.

When performing the merge, many different legitimate merge tree structures are possible, due to the associativity of C. In Figure 2, processors P1 and P2 evaluate their local results using differently shaped merge trees. Both trees yield correct results. However, when generating the merge tree within one processor, most implementations typically use a single method to generate the tree, namely sweeping across from left to right, much as the way P2 executes in Figure 2. In the next section, we describe an implementation in which we combine the Do and Merge operators in such a way that left-to-right execution is required if that operator is to be used. This implementation proves to be more efficient than the *Forall* implementation in most cases.

3 The *Pdo* construct

In many cases the programmer has additional knowledge about F and C that may be difficult or impossible for the compiler to derive. This information can be encapsulated into a new operator $C' : \alpha \times \beta \rightarrow \alpha$, where $C'(\alpha,\beta) \equiv C(\alpha, F(\beta))$. The C' operator takes a partial result corresponding to inputs $i..j$ and the input $j+1$, and produces a new partial result corresponding to inputs $i..(j+1)$.

We have implemented the Do&Merge using a construct called the *Pdo*, in which the programmer specifies the implementation of I, C', and C. Note that defining C' and I implicitly gives a definition of F, since $C'(I,\beta) = C(I, F(\beta)) = F(\beta)$. Also note that, given implementations of F and C, a naive implementation of C' is trivial, simply by the definition of C'. The *Pdo* construct imposes a particular order on the merge tree to yield maximum parallel performance. The basic *Pdo* loop is specified in the following manner:

```
PDO i = ⟨loop bounds⟩
  FIRST: ⟨implementation of I⟩
  NEXT:  ⟨implementation of C'⟩
  MERGE: ⟨implementation of C⟩
END PDO
```

Each processor begins by executing I. Then the processor iterates from left to right through the array elements that reside on that processor, executing the

C' operator, thus yielding one partial result per processor. This step requires $(N/P)(T(C'))$ time to execute. (Note that we ignore the time for executing the `FIRST` section, which should be relatively small since N/P is assumed to be large.) Once each processor has completed this work, the partial results are merged between processors using the C operator, yielding a single result. As in the *Forall* version, this merge step requires $T(C)\log P$ time to execute, for a total execution time of:

$$(N/P)(T(C')) + T(C)\log P. \tag{2}$$

By the definition of C', it is trivial to implement C' such that $T(C') = T(F) + T(C)$. Comparing equations (1) and (2), we find that the *Pdo* version outperforms the *Forall* version whenever:

$$T(C') < T(F) + T(C). \tag{3}$$

In Section 5 we show several examples of applications where (3) holds.

The *Pdo* representation of the Do&Merge has an additional benefit: it often gives the programmer a more intuitive way of expressing the parallelism. This intuition comes from the fact that the C' operator is inherently sequential, which reflects the way that a programmer typically approaches an algorithm. The *Pdo* construct is directly based upon the parallel looping construct in the Adapt [6] programming language, which has proven to be a useful construct suitable for efficient implementation.

The Do&Merge in general, and the *Pdo* in particular, can also be used to implement a general FORALL loop, or a general reduction operator. The FORALL can be specified simply by omitting the merge operator, in which case the compiler can trivially remove the merge phase. In a general reduction, the F function is a no-op; hence C and C' are identical.

4 Compilation of the *Pdo* loop

In addition to being simple for the programmer to specify, the *Pdo* loop is simple for the compiler writer to implement efficiently if the compiler provides a framework for executing array assignment statements.

The first step for the compiler is to partition the variables referenced within the loop into 3 sets: *induction variables*, *conversion variables*, and *merge variables*. Merge variables are variables written by the C operator, and conversion variables are variables accessed by the conversion function F that are not induction variables or merge variables. We allow F to have localized side effects, so long as no two iterations try to write to the same element of a conversion variable. To determine this partitioning, the compiler can use analysis of the access patterns, or it can rely on hints from the programmer.

To simplify the discussion of the compilation of the *Pdo*, we assume for now that we are parallelizing over a single dimension. In Section 4.4.3, we discuss relaxing this restriction. Our implementation then consists of two phases: processing the Do section, and processing the Merge section.

4.1 Do phase

To process the Do section, we use copy-in/copy-out semantics. Conceptually, the entire global data space is copied into the local data space for each iteration, with the exception of the merge variables. Then each iteration runs independently and in parallel. Finally, any conversion variables that were modified are copied out into the global data space after all iterations have completed.

Clearly, it is infeasible to copy the entire data space for each iteration. Instead, we determine which section of an array is needed for each iteration, and we redistribute the array so that each iteration has all needed data local to the processor on which that iteration is executed. This redistribution is performed by creating a temporary array with the necessary distribution, and executing an array assignment statement to insert the data into the temporary. Techniques for efficiently compiling such assignment statements are described in detail in [1, 4]. The remainder of this subsection concerns how to determine the parameters of the redistribution.

We assume that array distributions are specified using the TEMPLATE, ALIGN, and DISTRIBUTE directives from HPF. The iteration space of the *Pdo* corresponds directly to the template. Thus we will declare a single template whose size is equal to the upper bound of the *Pdo* loop index.

The distribution of *Pdo* iterations to processors corresponds directly to the distribution of the template. Thus we can use any distribution of loop iterations that the compiler can also use for distributing templates across the processors.

The access pattern of a conversion variable corresponds to the way that the variable's temporary should be aligned with the template. For example, if iteration i accesses `A`$(:, i)$ (i.e., column i of array `A`), then we generate the alignment statement: "`ALIGN A`$(:, i)$ `WITH T`(i)," where `T` is the generated template. As another example, if iteration i accesses `A`$(2i + 1, :)$, we generate the alignment statement: "`ALIGN A`$(2i + 1, :)$ `WITH T`(i)." In fact, any alignment function that can be handled within `ALIGN` statements corresponds to data access patterns that can be dealt with in the Do section.

After the input arrays are redistributed into temporaries, each processor must generate a loop that iterates through the *Pdo* iterations mapped to that processor. When the Do phase completes, we perform the copy-out of conversion variables that were modified (or we can wait until after the Merge phase to do the copy-out).

4.2 Merge phase

To process the Merge section, we set up a merge tree of height $\log P$ where P is the number of processors. This merge tree is a binary tree such that the merging begins at the leaves and propagates up toward the root. For each merge step, the right child processor sends its partial result to the left child, which assumes the role of the parent node for the merge step. The left child merges the results and continues, while the right child remains idle for the remainder of the $\log P$ merge steps. Finally, when the result propagates up to the root, the root processor

broadcasts its result.

For the merge, each processor must allocate space for three copies of the merge variable. The three copies correspond to a node in the merge tree and its two children. The partial result is initially stored in the copy corresponding to the node. When a partial result is sent to another processor for merging, it is stored on the receiving processor in the copy corresponding to the right child, while the receiving processor copies its partial result into the left child copy, and then merges the two partial results and stores the new partial result in the node copy. In Section 4.3.3 we describe how to reduce the amount of extra storage needed.

4.3 Optimizations

Several simple optimizations can dramatically improve performance. The first optimization concerns eliminating the redistribution of conversion variables into temporary arrays. The second optimization precomputes local memory indices at compile time. The third optimization reduces the amount of memory usage (and the associated execution time for managing that memory) in the Merge phase.

4.3.1 Copy-in/copy-out optimization

Recall from Section 4.1 that for each conversion variable, we create a temporary array which is aligned in the appropriate dimension with the *Pdo* template. We redistribute the conversion variable into this temporary using an array assignment statement, and then use the temporary inside the *Pdo* body in place of the original variable. However, it is clear that this redistribution is unnecessary if the alignment and distribution of the original array already matches that of the proposed temporary. In particular, if the following conditions hold, then there is no need for the temporary:

- The template to which the conversion variable is aligned matches the *Pdo* template.
- The alignment of the conversion variable to its template matches the proposed alignment of the temporary to the *Pdo* template.
- The distribution of the template to which the conversion variable is aligned matches the distribution of the *Pdo* template.

Although the compiler has little choice in the *Pdo* template or in the alignment of the temporaries to the template, it can choose an arbitrary distribution of the template (and thus an arbitrary distribution of loop indices to processors). Thus to minimize the amount of redistribution that takes place, the compiler should choose a distribution to maximize the number of conversion variables for which all three conditions above hold.

For the Do phase, the overhead of computing the distributed loop bounds is essentially identical regardless of the distribution. Thus, provided the loop iterations are distributed evenly across the processors (providing a load balance), the only difference in execution time will come from array redistributions. However, not all array redistributions are equally costly; e.g., a nearest-neighbor communication should be cheaper than an all-to-all communication. Furthermore, as we see in Section 4.4.1, the distribution choice can have a large impact on the cost of the Merge phase. These issues combine to form an interesting optimization problem.

4.3.2 Local index computation

Figure 3 shows how global indices of an array are stored locally on a processor. The shaded boxes represent indices of an array `A` with a block-cyclic distribution that reside on a particular processor. These array elements are compactly stored in `LM_A` , as shown in Figure 3. When we access a portion of the array within the Do phase, we must translate the global index into a local index. This translation can be expensive, involving multiplication, division, and modulo operators. It is especially expensive if the translation is performed within an external procedure, as the procedure call breaks up basic blocks, thus limiting the effectiveness of the optimizer.

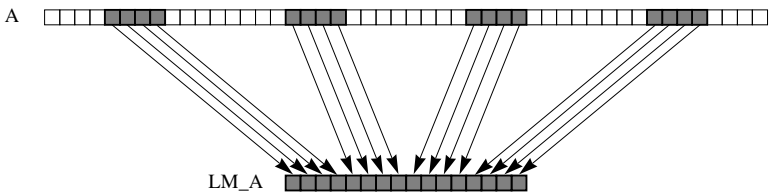

Figure 3: Array compaction.

However, a simple optimization is to compute the local memory index once per loop iteration, and avoid the repeated evaluations of the same expression. Better yet, when the stride of the loop bounds is 1, we notice that the stride of the local memory index is also 1. Thus we can compute the local memory index for only the first iteration, and simply increment it at the end of each loop iteration.

4.3.3 Merge variable memory optimization

As described in Section 4.2, the Merge phase implementation triples the amount of memory usage, to hold the left and right partial results as well as the new partial result. Since the processor corresponding to the left child performs the merge computation for itself and the right child, we clearly need to allocate memory to receive the right child's partial result. However, in many cases, we can map the left child's partial result and the new partial result to the same memory,

eliminating the need for special memory for the left child and correspondingly the need to perform the copy. Within a single merge step, if we ever read the left child's partial result after we have written to the new partial result, then we cannot share the memory for the two partial results. This situation can generally be detected using simple data dependence testing techniques.

4.4 Advanced issues

Until now, we have implicitly made many assumptions regarding the specification of the Do&Merge in general, and the *Pdo* in particular. These issues are addressed and resolved here.

4.4.1 Non-block data distribution

Our analysis for merging so far has assumed that the input array is distributed across the processors in block fashion. In particular, we have assumed that each processor iterates over a contiguous section of the global iteration space. However, if we distribute the iteration space in cyclic or block-cyclic fashion, this condition no longer holds.

Due to the non-commutativity of the merge operator, we can only apply C across contiguous blocks of the global index space. Furthermore, we can also only apply C' across contiguous blocks. This presents a problem when several contiguous blocks reside on a single processor. In this case, the processor generates several partial results to be globally reduced, one per contiguous block.

The compiler can generate code to perform the correct merge in the following manner. Recall that we previously specified that the processor would begin by initializing its partial result r to the identity specified by I, then sequentially apply C' across the input, and finally perform a global merge of each processor's r value using the C operator. Now, we generalize by making r a vector of length B, where B is the number of contiguous blocks on the processor. For each block i, we initialize $r[i]$ to the identity I, and then sequentially apply C' across the block. When performing the global merge, we apply C individually to each $r[i]$ on the left and $r[i]$ on the right, yielding the new value of $r[i]$; hence two vectors of length B are point-wise merged using the C operator to yield a new vector of length B. At the end, there will be a final vector of length B on a single processor. At this point, we simply apply C across this vector, yielding a single global result.

Note that our simplified timing analysis breaks down in the general block-cyclic case. For the purpose of simplicity, we have assumed that the time required to execute I once is trivial compared to the N/P times that we execute C'. Whereas C' is still executed N/P times per processor, I will now be executed B times per processor, and the time spent doing the global merge is increased by a factor of B (in terms of both computation and amount of communication). This effect is particularly noticeable in a cyclic distribution, where B is maximized at $B = N/P$.

4.4.2 Commutative merge operator

In many cases, the C operator is commutative. Although compiler analysis might be able to determine this, the programmer might also be able to explicitly provide this information to the compiler. When C is commutative, and the input distribution is block-cyclic, a non-block data distribution no longer presents the problems described in Section 4.4.1. Since C is commutative, we can conceptually rearrange the input to be block distributed. This rearrangement could be done explicitly with a redistribution in the form of an array assignment, although this operation would most likely involve communication. A better approach is to simply apply C' across the input array, regardless of contiguity. This approach involves minimal overhead and no extra communication.

4.4.3 Nested *Pdo* loops

Our analysis to this point has assumed that in a loop nest, we are only parallelizing a single loop. This is adequate if we are certain that only that particular dimension of the input array is distributed. However, when the input array is distributed over more than one dimension, we may wish to parallelize the outermost loops.

For the remainder of this discussion, we will assume that the *Pdo* loops are perfectly nested; hence the syntax is specified in the following manner:

$$\texttt{PDO}\ \ (i = 1 : m, j = 1 : n)\ \ \ldots$$

Distributing the index space across the processors is simple: just calculate the distribution independently for each index, and take the Cartesian product of the results.

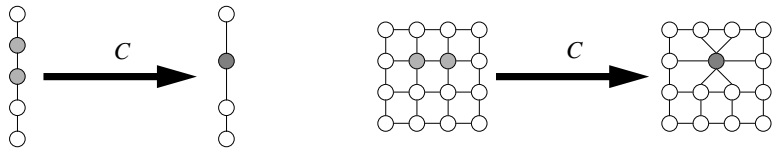

Figure 4: Merging in one and two dimensions.

The nontrivial issue involves the merge step. When the iteration space is k-dimensional, we can envision iterating over a k-dimensional mesh, with the rule that we can only apply C to two adjacent nodes in the mesh. When C is applied to the two adjacent nodes, the edge between them is contracted. One such contraction is illustrated in Figure 4 for both a 1-dimensional and a 2-dimensional index space. These contractions continue until there is only one node remaining in the graph.

However, this model of arbitrary edge contraction may in some cases place an undue burden on the programmer. For example, consider an image connected components algorithm. The F function converts a section of input to

an equivalence table. However, the F function must also save the border of the image section, which is needed by the C operator for merging two equivalence tables. The C operator must create a new equivalence table, plus the boundary of the merged region. If we allow arbitrary edge contraction, the programmer must be prepared to handle arbitrary shaped boundaries. For this reason, we might choose to guarantee the programmer that edges will be contracted in some regular order, such as in rectangular order.

5 Performance results

In this section, we consider two examples of computation that benefit from using the Do&Merge construct (and the time to perform the Do&Merge loop is less than the time it takes to perform a parallel loop and a reduction separately) : a Histogram example from the domain of image processing, and a Matvec (Matrix-vector multiplication) example from the domain of regular matrix computations. For each example, we give both *Forall* and *Pdo* versions in an HPF-like notation, and we compare their performance when compiled by a parallelizing Fortran compiler on the iWarp system [5]. In each case, the *Pdo* version is significantly faster, because it allows more overlapped execution.

5.1 Histogram

The first example is a 2-dimensional histogram. The input is an $n \times n$ *image* of grayscale pixel values, where $1 \leq image(i,j) \leq G$, and the output is a vector *hist* of length G, where $hist(i)$ is the number of pixels in *image* with a value of i. We assume that the image is block-distributed by rows.

The *Pdo* version of Histogram is implemented with the following parallel loop:

```
PDO i=1,n
INPUT image(i,:)
OUTPUT result(:)
FIRST
   result = 0
NEXT
   DO j = 1,n
      result(image(i,j)) = result(image(i,j)) + 1
   END DO
MERGE
   result = LEFT(result) + RIGHT(result)
END PDO
```

The INPUT directive aligns the ith row of the input image to the ith loop iteration. The OUTPUT directive identifies the result vector that will be merged across processors. Note that in general the information in the INPUT and OUTPUT directives could be determined automatically by compiler analysis.

We have included these directives merely to simplify the implementation of our compiler.

Recall from Section 3 that the FIRST section defines the I operator, the NEXT section defines the C' operator, and the MERGE section defines the C operator. The LEFT and RIGHT intrinsics identify the intermediate merge results from the left and right subtrees, respectively, of the merge tree.

The corresponding *Forall* version of Histogram consists of a separate FORALL, which computes the histogram per row, followed by a call to a reduction intrinsic.

```
local = 0
FORALL i=1,n
   DO j = 1,n
      local(i,image(i,j)) = local(i,image(i,j)) + 1
   END DO
END FORALL
result = REDUCE(SUM,0,local)
```

Notice that the body of the loop defines the F operator. Notice also that in each version, processors communicate only when results are merged.

For the Histogram example, we see that $T(F) + T(C) = O(G + n)$, while $T(C') = O(n)$. In practice, the savings is substantial, especially as G increases. This savings can be seen quite clearly in Figure 5.

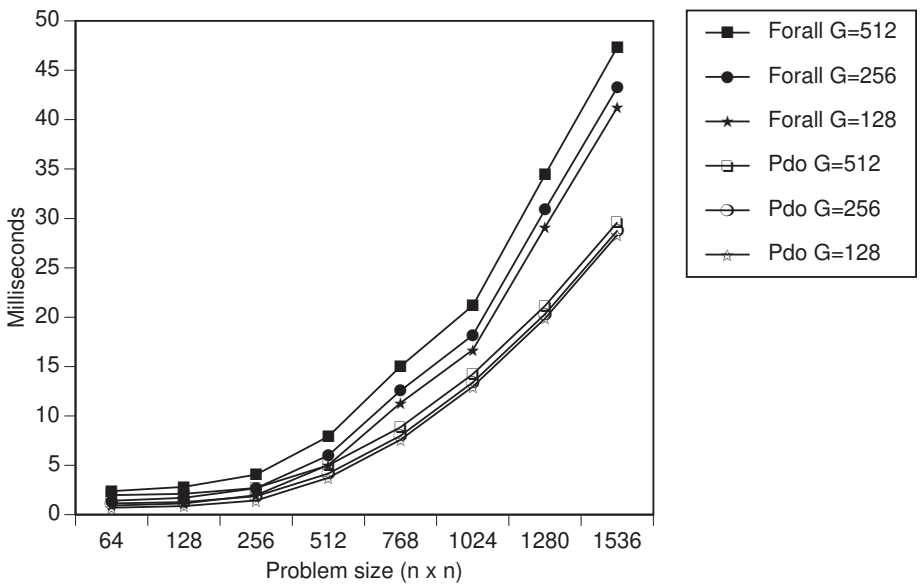

Figure 5: Performance of Histogram on iWarp.

5.2 Matvec

The second example is a Matvec operation, $y = Ax$, where A is a dense $m \times n$ matrix, x is an n-vector, and y is an m-vector. We assume that A is block-distributed by columns, x is block-distributed, and y is replicated.

The *Pdo* version of Matvec is implemented by applying a SAXPY operator on each column of A, and then summing the resulting vectors to form y.

```
PDO j=1,n
INPUT a(:,j),x(j)
OUTPUT y
FIRST
   y = 0.0
NEXT
   y = y + a(:,j)*x(j)
MERGE
   y = LEFT(y) + RIGHT(y)
ENDPDO
```

The corresponding *Forall* version of Matvec is implemented with a FORALL loop, followed by a call to a reduction intrinsic that merges the intermediate results:

```
FORALL j=1,n
   b(:,j) = a(:,j)*x(j)
END FORALL
y = REDUCE(SUM,0,b)
```

Again, note that processors communicate only when results are merged.

For the Matvec example, we see that while $T(F)+T(C) = O(m)$ and $T(C') = O(m)$, the C' operator performs $3m + 1$ memory accesses and the F and C operators perform a total of $5m + 1$ memory accesses. Thus we would expect the *Forall* version to run roughly a factor of 5/3 slower than the *Pdo* version, if we neglect the global merge time, which is relatively small as n increases. The measured results in Figure 6 confirm this expectation, especially for larger values of n and m.

6 Concluding remarks

We have identified a common operation, called the Do&Merge, that can be significantly optimized by integrating two previously separated operations into one common operation. This optimization of the Do&Merge, which we call the *Pdo*, is beneficial for a number of reasons: the programmer can often give the compiler a more efficient implementation of the C' operator than the compiler can derive by itself; and the C' operator is often more natural for the programmer to express than the F and C operators separately. Further, the *Pdo* can be implemented quite easily in any parallelizing compiler that supports array assignment statements.

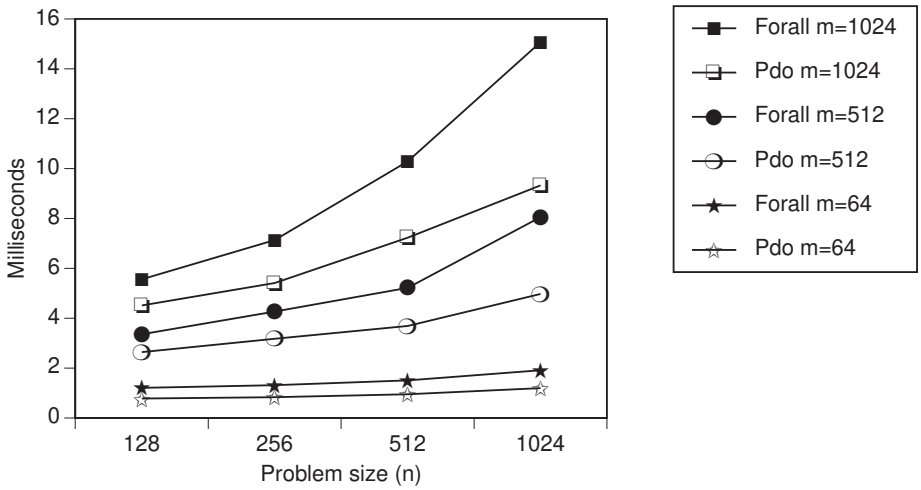

Figure 6: Performance of Matvec on iWarp.

We are currently evaluating the *Pdo* implementation within the framework of a parallelizing Fortran compiler similar to HPF. We have presented some preliminary results of that evaluation, where we found performance improvements of up to 50% for Histogram and Matvec example programs.

References

[1] S. Chatterjee, J. Gilbert, F. Long, R. Schreiber, and S. Teng. Generating local addresses and communication sets for data-parallel programs. In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 149–158, San Diego, CA, May 1993.

[2] R. Cytron. Doacross: Beyond vectorization for multiprocessors. In *Proceedings 1986 International Conference on Parallel Processing*, pages 836–844, St. Charles, Illinois, 1986.

[3] High Performance Fortran Forum. High Performance Fortran language specification version 1.0, May 1993.

[4] J. Stichnoth. Efficient compilation of array statements for private memory multicomputers. Technical Report CMU-CS-93-109, School of Computer Science, Carnegie Mellon University, February 1993.

[5] J. Subhlok, J. Stichnoth, D. O'Hallaron, and T. Gross. Exploiting task and data parallelism on a multicomputer. In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 13–22, San Diego, CA, May 1993.

[6] J. A. Webb. Steps toward architecture independent image processing. *IEEE Computer*, 25(2):21–31, February 1992.

[7] M. Wolfe. Doany: Not just another parallel loop. In *Conference Record, Fifth Workshop on Languages and Compilers for Parallel Computing*, pages 1–12, Yale University, August 1992.