A Practical Approach to DOACROSS Parallelization

Priya Unnikrishnan[1], Jun Shirako[2], Kit Barton[1], Sanjay Chatterjee[2], Raul Silvera[1], and Vivek Sarkar[2]

[1] IBM Toronto Laboratory, {priyau,kbarton,rauls}@ca.ibm.com
[2] Department of Computer Science, Rice University, {js20,cs20,vs3}@rice.edu

Abstract. Loops with cross-iteration dependences (DOACROSS loops) often contain significant amounts of parallelism that can potentially be exploited on modern manycore processors. However, most production-strength compilers focus their automatic parallelization efforts on DOALL loops, and consider DOACROSS parallelism to be impractical due to the space inefficiencies and the synchronization overheads of past approaches. This paper presents a novel and *practical* approach to automatically parallelizing DOACROSS loops for execution on manycore-SMP systems. We introduce a compiler-and-runtime optimization called *dependence folding* that bounds the number of synchronization variables allocated per worker thread (processor core) to be at most the maximum depth of a loop nest being considered for automatic parallelization. Our approach has been implemented in a development version of the IBM XL Fortran V13.1 commercial parallelizing compiler and runtime system. For four benchmarks where automatic DOALL parallelization was largely ineffective (speedups of under 2×), our implementation delivered speedups of 6.5×, 9.0×, 17.3×, and 17.5× on a 32-core IBM Power7 SMP system, thereby showing that DOACROSS parallelization can be a valuable technique to complement DOALL parallelization.

1 Introduction

As hardware processors move from multicore to manycore designs, the challenge of enabling software to exploit parallelism is gaining a heightened urgency. While a number of programming models have been proposed for explicit parallelism, manual parallelization still requires a high degree of parallel programming expertise, and is often time-consuming and error-prone. It is widely believed that *automatic parallelization* can play an important role in improving the programmability of manycore-SMP systems [4] because it requires minimal or no effort by users. Furthermore, techniques for automatic parallelization can also be used in programming tools that assist in manual parallelization.

Most compilers focus on loops with no cross-iteration dependences in which all iterations can be executed completely in parallel with each other; such loops are referred to as DOALL loops. Loops with cross-iteration dependences are referred to as DOACROSS loops, and are usually serialized or, in some cases, transformed into skewed DOALL loops when practical to do so. However, Amdahl's Law dictates that it will be increasingly important to pay attention to the sequential fraction of the program, including non-parallelized DOACROSS loops, as we move to manycore processors. Unfortunately, past approaches to DOACROSS parallelization are impractical for use in production-strength

compilers due to unrealistic assumptions about *space* (*e.g.*, allocating one synchronization variable per dynamic iteration instance) or *granularity* (*e.g.*, performing synchronization operations even when their overhead exceeds the execution time of a loop iteration).

This paper presents a novel and *practical* approach to automatically parallelizing DOACROSS loops for execution on manycore-SMP systems. We introduce a compiler-and-runtime optimization called *dependence folding* that bounds the number of synchronization variables per worker thread (processor core) to be at most the maximum depth of a loop nest being considered for automatic parallelization. We also present an effective cost analysis to determine the profitability of DOACROSS parallelization, and practical techniques to increase the granularity of computation between successive synchronization operations. Our approach has been implemented in a development version of the IBM XL Fortran V13.1 commercial parallelizing compiler and runtime system. For four benchmarks where automatic DOALL parallelization was largely ineffective (speedups of under $2\times$), our implementation delivered speedups of $6.5\times$ for LU, $9.0\times$ for Poisson, $17.3\times$ for SOR, and $17.5\times$ for Jacobi on a 32-core IBM Power7 SMP system. Thus, DOACROSS parallelization can be a valuable technique to complement DOALL parallelism in cases where DOALL parallelization results in limited benefits [13].

The rest of the paper is organized as follows. Section 2 summarizes some of the previous work in this area. Section 3 describes our approach to DOACROSS parallelization, with details on dependence folding and runtime algorithms. Section 4 describes our methods for cost analysis and optimal grain size selection. Section 5 presents experimental results to evaluate the effectiveness of our approach. Section 6 contains our conclusions, along with suggestions for future work.

2 Previous Work

Early work on DOACROSS parallelization concentrated primarily on the synchronization mechanisms used. Cytron [2] showed how to determine a DOACROSS schedule to enforce a given set of dependences, based on the delays to be introduced for different iterations of the loop in processors that execute synchronously. Padua and Midkiff [7] focused on synchronization techniques to enforce loop carried dependences in singly-nested DOACROSS loops. They use one synchronization variable per data-dependence in the loop, and do not consider multi-dimensional loops. Wolfe [12] looked at four different synchronization mechanisms such as synchronizing at every data-dependence relationship in the loop, dividing the loop into segments of statements and pipelining the execution of the segments, inserting barriers at various points in the loop, using ordered critical sections etc. Again, only singly-nested loops were considered.

Su and Yew [10] proposed several interesting data synchronization schemes. The data-oriented scheme uses a dedicated synchronization variable for each datum involved in a dependence relationship in the loop, while the statement-oriented and process-oriented schemes have one per statement and iteration, respectively. They considered multi-dimensional loops with both a single level of parallelism and nested parallelism, but did not include any experimental results. Li [5] presents algorithms to generate synchronization code based directly on array subscripts and loop bounds using an array of event variables. This technique does not require constant data dependence distances

and can target arbitrarily nested loops. Chen et al [3] proposed an algorithm for runtime parallelization of DOACROSS loops when data dependences cannot be determined at compile-time. Tang et al [8], presented synchronization schemes that can parallelize general nested loop structures with complicated cross-iteration data dependences.

Our experience in the area of automatic parallelization has led us to believe strongly that the choice of synchronization mechanism, its implementation and its tuning all have a major impact on the (im)practicality of a given approach to DOACROSS parallelization. All of the above papers [2, 3, 10, 5, 7, 8, 12] propose interesting techniques for synchronization, but lack quantitative measurements on the performance gains achieved, and the synchronization costs and memory requirements of the stated methods. Our synchronization mechanism uses a simple and intuitive "iteration vector" based scheme that can be easily applied to multi-dimensional loop nests. Our experimental results show that a single level of parallelism is sufficient in most cases to exploit the available resources.

There has also been some notable past work on optimizing synchronization operations. Krothapalli [11] targeted redundant synchronization elimination by removing redundant dependences in simple loops with constant dependences. Rajamony and Cox [9] used integer programming to determine the optimal solution to minimize the amount of synchronization in DOACROSS loops while retaining the parallelism that can be extracted from the loop. Chen [1] focused on increasing parallelism with statement reordering and reducing communication overhead by eliminating redundant synchronizations.

An optimal granularity of computation is required to offset the overhead of synchronization. Pan et al [14] used tiling to increase the parallelization granularity and propose a formulation for the optimal tile size. They conclude that static scheduling significantly outperforms dynamic self-scheduling by enhancing inter-tile locality. Lowenthal [6] presented a flexible runtime approach to determine the granularity for pipelined parallelization. Our work instead uses a cost-based combination of compile-time and runtime analyses to determine the granularity of work. Our results show that the accuracy of cost analysis can have a significant impact on parallel performance and scalability.

3 DOACROSS Parallelization Algorithm

Our approach to automatic DOACROSS parallelization is based on the assumption that there is one (logical) synchronization variable allocated per dynamic loop iteration. Thus, the sources and targets of inter-iteration synchronization operations can be denoted as *iteration vectors*. (Recall that iteration vector $\overrightarrow{I_v} = (I_1, I_2, I_3, ...I_n)$ represents a unique point in an n-dimensional iteration space.) The core idea is that a dependent iteration can examine the status of the synchronization variables of the iterations that it is waiting on to determine when it can start execution. Using the iteration vector as the synchronization variable interface has several advantages. It is very efficient to implement in terms of the memory required (as we will see below), simple to understand and implement, easily extensible to multidimensional loops, and does not constrain the inherent parallelism in the loop nest.

In our approach, synchronization is performed at the statement level of a given program representation. We assume that standard POST/WAIT operations can be performed on the iteration vector synchronization variables to enforce the data dependence rela-

tionships in the loop. A POST is inserted after the source statement of the dependence and the WAIT statement is inserted before the sink statement of the dependence:

1. $WAIT(w\overrightarrow{I_v})$: Causes execution to wait until the iteration specified by the iteration vector $w\overrightarrow{I_v}$ is completed. The iteration vector $w\overrightarrow{I_v}$ of WAIT is computed using the current iteration vector and the dependence distance vector $\overrightarrow{D} = (d_1, d_2, d_3, ...d_n)$ of the data dependence $w\overrightarrow{I_v} = (I_v - \overrightarrow{D}) = (I_1 - d_1, I_2 - d_2, ...I_n - d_n)$
2. $POST(p\overrightarrow{I_v})$: Indicates the completion of the iteration specified by the iteration vector $p\overrightarrow{I_v}$. The iteration vector $p\overrightarrow{I_v}$ of POST is the current iteration being executed. $p\overrightarrow{I_v} = \overrightarrow{I_v} = (I_1, I_2, ...I_n)$

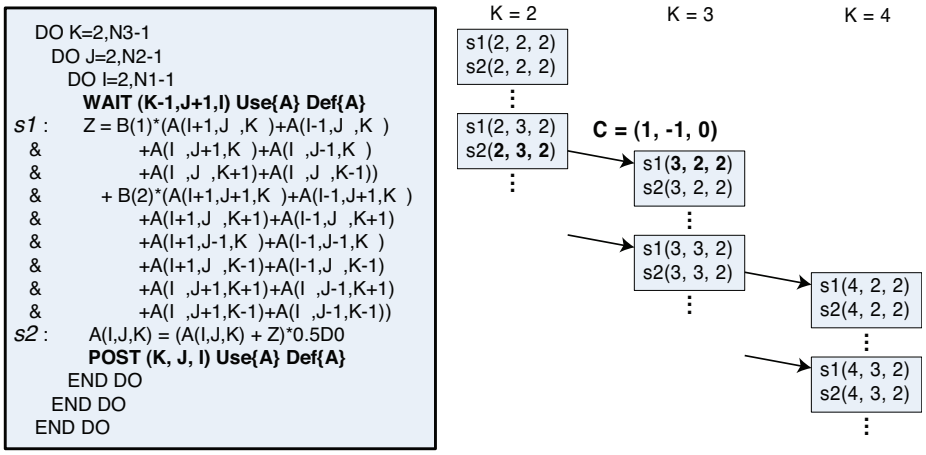
3.1 Dependence Folding

With the aim of reducing synchronization overheads so as to make DOACROSS parallelization practical, our implementation folds all the loop-carried dependences in the loop into a single, conservative dependence. This leads to the insertion of at most one pair of synchronization primitives per iteration. In our experience with current hardware, the lower synchronization cost resulting from at most one synchronization per iteration far outweighs the potential loss in parallelism due to conservative approximation.

Definition 1. *A loop-carried data dependence is composed of the source statement, sink statement and the dependence distance* $\Delta = \{S_{src}\delta^* S_{sink}, \overrightarrow{D}\}$.

Consider a perfect loop nest L with n dimensions and m statements $\{S_1..S_m\}$ and k data dependences $\Delta^i = \{S_x\delta^* S_y, \overrightarrow{D^i}\}$, $i \in \{1..k\}$, $x \in \{1..m\}$ and $y \in \{1..m\}$. Each dependence vector, $\boldsymbol{D^i}$ has the form $\boldsymbol{D^i} = (d_1^i, d_2^i, ..., d_n^i)$. The single conservative dependence is computed by considering all the data dependences $\Delta^{1..k}$ in loop nest L. The source of the conservative dependence is computed as the *Lexically Latest Source* (LLS) statement across all the data dependences in the loop nest. In control flow terms, the LLS statement can be computed as follows. First compute the least common ancestor, LCA, of all source statements in the *postdominator tree* for the loop; then, find the closest ancestor of LCA in the postdominator tree that is unconditionally executed in the loop body. This statement is the LLS. Likewise, the sink of the conservative dependence is computed as the *Lexically Earliest Sink* (LES) statement across all the data dependences in the loop nest (by using the dominator tree instead of the postdominator tree). After the source and sink statements have been identified for the conservative dependence, the next step is to identify the conservative dependence distance vector $\overrightarrow{C}$. As our mechanism applies only to a single level of parallelism in the loop, it is possible to use a trivial formulation for the conservative dependence distance shown below. Assuming the outermost dimension is parallelized without loop chunking, the first dimension of $\overrightarrow{D^i}$ denotes the stride (i.e. dependence distance) along with the inter-thread loop dependence. Therefore, the first dimension of $\overrightarrow{C}$ should correspond to the maximum value of common strides in that dimension, which is the GCD value. The remaining dimensions can be conservatively computed by using $min\_vect(\overrightarrow{V_1}, \overrightarrow{V_2}, .., \overrightarrow{V_k})$, which determines the lexicographically smallest vector of $\overrightarrow{V_1}$, $\overrightarrow{V_2}$, $..\overrightarrow{V_k}$.

$$\overrightarrow{C} = \begin{pmatrix} C[1] \\ C[2..n] \end{pmatrix} = \begin{pmatrix} gcd(d_1^1, d_1^2, ...d_1^k) \\ min\_vect(D^1[2..n], ...D^k[2..n]) \end{pmatrix}$$



(a) Kernel with POST/WAIT (b) Conservative dependence

Fig. 1: Pipelining POISSON

After insertion of POST/WAIT operations, the compiler will look for code motion opportunities to move the POST operations as early as possible in the loop, and the WAIT operations as late as possible in the loop. To ensure that such transformations do not violate any data dependences, the POST/WAIT operations are augmented with pseudo USE and DEF sets as follows:

1. Flow dependence (δ^f): A flow dependence is from a *def* of the variable to its *use*. The WAIT is inserted before the *use* and the POST is inserted after the *def*. In order to prevent the *use* of the dependence variable from moving up past the WAIT call, the variable is inserted in a pseudo DEF set for the WAIT call. Similarly, in order to prevent the *def* of the dependence variable from moving down below the POST call, the variable is inserted in a pseudo USE set for the POST call.
2. Anti dependence (δ^a): An anti dependence is from a *use* of the variable to its *def*. The WAIT is inserted before the *def* and the POST is inserted after the *use*. The dependence variable is marked as a *use* in the WAIT call to ensure that the WAIT is completed before the *def*, and as a *def* in the POST call to ensure that the POST is performed after the *use* of the variable.
3. Output dependence (δ^o): An output dependence is between two *def*'s of the same variable. The dependence variable is marked as *use* in both the WAIT and POST call to ensure that the WAIT is done before all the *defs* and POST is done after all the *defs* of that variable.

Figure 1a shows the POISSON computational kernel, which is a 3-dimensional ($400\times400\times400$) DOACROSS loop nest with the POST/WAIT synchronization primitives inserted after conservative dependence computation. In this case, there are multiple flow dependences from s2 to s1 for array A with the following dependence distances:
$\overrightarrow{D^1} = (1,0,0)$, $\overrightarrow{D^2} = (1,0,-1)$, $\overrightarrow{D^3} = (1,0,1)$, $\overrightarrow{D^4} = (1,-1,0)$, $\overrightarrow{D^5} = (1,1,0)$

and multiple anti dependences from $s1$ to $s2$ for array A with the following dependence distances:
$\overrightarrow{D^6} = (1,0,0)$, $\overrightarrow{D^7} = (1,0,1)$, $\overrightarrow{D^8} = (1,0,-1)$, $\overrightarrow{D^9} = (1,1,0)$, $\overrightarrow{D^{10}} = (1,-1,0)$.
The conservative dependence are computed as follow:

$$\begin{aligned}\overrightarrow{C} &= (gcd(1,1,1,1,1,1,1,1,1,1),\\ &\quad min\_vect((0,0),(0,-1),(0,1),(-1,0),(1,0),(0,0),(0,1),(0,-1),(1,0),(-1,0)))\\ &= (1,-1,0)\end{aligned}$$

Based on the conservative dependence, `POST(K,J,I)` is inserted after lexically last source $s2$ and `WAIT(K-1,J+1,I)` is inserted before lexically earliest sink $s1$.

3.2 Runtime Implementation

The compiler outlines and parameterizes the DOACROSS loops after the POST/WAIT synchronization calls to the runtime are inserted. The parallel runtime system is responsible for initializing data structures and scheduling the DOACROSS loops. The current implementation employs a static cyclic scheduling policy with a chunk size of one, where iterations are assigned to processors in a round-robin fashion. The static cyclic policy inherently brings good load balance and data locality in addition to low overhead due to static iteration mapping.

Runtime Data Structure: Let m denote the number of threads and n denote the dimension of the DOACROSS loop nest. The runtime allocates a 2-dimensional array, $sync\_vec[1:m][1:n]$, which is a set of $m \times n$ synchronization variables to manage the POST/WAIT synchronization on the DOACROSS loop. Given a thread with id = $thrd\_id$, the 1-dimensional sub-array $sync\_vec[thrd\_id][1:n]$ represents the last iteration instance whose completion is ensured by the POST operation. Note that the iteration space is normalized and it is guaranteed that an iteration instance $p\overrightarrow{I_v}$ passed to a POST operation monotonically increases for each thread. A WAIT operation is blocked until when $sync\_vec[thrd\_id]$ is lexicographically larger or equal to the iteration instance $w\overrightarrow{I_v}$ passed to the WAIT.

Algorithm 1: POST algorithm

Input : The iteration vector of the current iteration $pI_v = (I_1, I_2, ..., I_n)$, n = dimension of loop, m = number of threads

```
begin
    // Check for boundary conditions
    // The loops are all lower bound and bump normalized
    if within_boundary(pI_v[1...n]) then
        thrd_id = mythread()
        // Update the synchronization variable of the current thread
        sync_vec[thrd_id][1..n] = pI_v[1..n]
end
```

Algorithm 2: WAIT algorithm

Input : The iteration vector of the dependence source $wI_v = (I_1 - d_1, I_2 - d_2, ..., I_n - d_n)$, n = dimension of loop, m = number of threads, $\overrightarrow{D} = (d_1, d_2..d_n)$ is the dependence distance

begin

  if $within\_boundary(wI_v[1...n])$ **then**

    // Determine the thread executing the source iteration specified by wI_v

    // Schedule is static with chunksize=1.

    $thrd\_id = wI_v[1]\%m$

    // Block until $sync\_vec[thrd\_id]$ is lexicographically larger or equal to $wI_v[1..n]$

    while $vector\_compare(sync\_vec[thrd\_id][1..n], wI_v[1..n]) < 0$ **do**

      wait

end

POST/WAIT Algorithm: Algorithms 1 and 2 show the $POST(p\overrightarrow{I_v})$ and $WAIT(w\overrightarrow{I_v})$ algorithms, respectively. The boundary check for the iteration instance $p\overrightarrow{I_v}/w\overrightarrow{I_v}$ is performed at the beginning of the POST/WAIT algorithm. Note that all valid elements of $p\overrightarrow{I_v}/w\overrightarrow{I_v}$ are non-negative because of the loop normalization. The POST algorithm assigns $p\overrightarrow{I_v}$ to $sync\_vec[thrd\_id]$ for the current thread. In the implementation, this assignment is done in reverse order, i.e., starting with the innermost dimension and going outer along with appropriate memory barriers. This ensures that the intermediate state of $sync\_vec[thrd\_id]$ is always smaller than $p\overrightarrow{I_v}$. The WAIT algorithm computes the target (source) thread based on the first dimension of $w\overrightarrow{I_v}$, and waits until $sync\_vec[thrd\_id]$ becomes lexicographically larger or equal to $w\overrightarrow{I_v}$. This vector comparison is done starting with the outermost dimension and going inner. The order of updating and reading of the synchronization vector by the POST and WAIT calls respectively ensures that a WAIT operation will never be unblocked prematurely due to an illegal intermediate state of $sync\_vec[thrd\_id]$. The WAIT operation is relatively cheap because it only performs a read of the synchronization variable of another thread. The POST operation is very expensive because it performs a write of the synchronization variable. Because the synchronization variable in our method is an iteration vector, the number of writes is equal to the number of dimensions of the doacross loop.

4 Profitability Analysis and Grain Size Selection

Profitability and cost analysis play a major role in automatic parallelization. Excessive synchronization or insufficient granularity of computations for parallelism can result in significant performance degradation. These considerations have been studied in past work on parallelization of DOALL loops [13], and need to be extended for parallelization of DOACROSS loops. In this section, we introduce a profitability analysis to determine when it is worthwhile to parallelize a DOACROSS loop.

First, we perform a special-case check for a one-dimensional loop nest. If the POST/WAIT calls encompass the entire loop body and the conservative dependence distance equals 1, then DOACROSS parallelization cannot be profitable since the POST/WAIT calls

effectively serialize the entire loop. This check does not apply if the loop nest contains $n > 1$ loops, since there may still be useful parallelism with a conservative dependence distance of 1 at the outermost level (enabled by fine-grained synchronization calls in the inner loops).

Second, we perform loop unrolling to reduce the amortized overhead of synchronization operations by increasing the granularity of computation between POST and WAIT operations. After unrolling, the lexically last POST and earliest WAIT operations are retained, and all the intervening calls to POST/WAIT are removed so as to reduce the overall synchronization overhead. Also, the iteration instance $w\overrightarrow{I_v}$ of the lexically earliest WAIT is adjusted to match the lexically last POST according to the unrolling factor.

We assume the availability of two parameters, *MinGrainSize* and *MaxLoopBodySize*, to guide our transformations for grain size selection. *MinGrainSize* imposes a lower bound on the granularity of computation to be performed between POST and WAIT operations. To compute the heuristics for the grain-size *MinGrainSize* for DOACROSS parallelization, we start by looking at the heuristics for DOALL parallelization [13]. These values were then adjusted to take into account the communication overhead. Subsequently, experimental runs were performed to further fine tune the heuristics. Similarly to determine the code-size *MaxLoopBodySize* for DOACROSS parallelization, we rely on previously calculated heuristics for the unrolling transformation. These heuristics are adjusted to prevent excessive code growth during unrolling. For the platform studied in this paper (Power7 with an XLC runtime system), it was determined that 20,000 cycles and 320 cycles are reasonable value for *MinGrainSize* and *MaxLoopBodySize* respectively. However, our approach is applicable to any other values that may be specified for these parameters.

To select the unroll factor, *UF*, for the innermost loop, we first estimate the cost of a single iteration of the loop, *LoopBodyCost*. Then, the unroll factor selected by our approach can be specified as $UF = \min(32, \lceil MaxLoopBodySize/LoopBodyCost \rceil)$, where 32 is an upper bound that is imposed on *UF* for practical reasons. If $n = 1$, an extra constraint is imposed to ensure that *UF* is less than the conservative dependence distance for the DOACROSS loop.

Finally, we perform a special form of *chunking* of the inner loops in a DOACROSS loop nest, by estimating a chunk size that we refer to as a *Runtime Granularity Factor*, *RGF*. *RGF* specifies the number of iterations of the inner loops that should be executed sequentially. This is achieved by skipping $RGF-1$ POST operations in the inner loops, so that one POST operation is performed for every *RGF* POSTs. As described in Section 3.2, the POST$(p\overrightarrow{I_v})$ operation ensures that all WAIT operations whose iteration instance $w\overrightarrow{I_v}$ is lexicographically smaller or equal to $p\overrightarrow{I_v}$ can be unblocked. Therefore, it is safe to perform only the last POST operation after $RGF-1$ POSTs. To avoid a potential deadlock when the number of iterations is not an exact multiple of *RGF*, an additional POST operation is inserted at the end of each iteration of the outermost DOACROSS loop to signal that all iteration instances included in that iteration have been completed. Note that WAIT operations, which have much smaller synchronization cost than POST operations, are always performed.

The initial value of *RGF* is selected at compile-time by using the formula, $RGF = MinGrainSize/(UF * LoopBodyCost)$, This value is further adjusted at runtime based on the number of threads executing the DOACROSS loop. If there are more threads, a

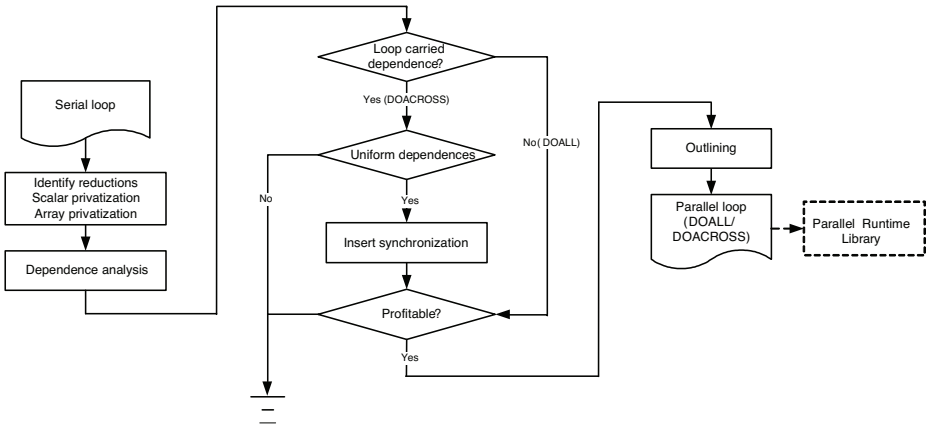


Fig. 2: Context for automatic DOACROSS parallelization in the XL Fortran and C/C++ compilers

larger value of *RGF* may reduce the amount of parallelism that can be exploited. Thus we adjust the *RGF* value selected at compile-time to a runtime value, RGF', as follows: $RGF' = 2 \times RGF/NumThreads$. Note that $RGF' = RGF$ when $NumThreads = 2$, and becomes proportionately smaller as *NumThreads* increases, thereby balancing the trade-off between overhead and parallelism.

5 Experimental Results

This section presents results from the experiments conducted to evaluate our implementation. The experiments were performed on a Power7 system with 32-core 3.55GHz processors running Red Hat Enterprise Linux release 5.4. The measurements were done using a development version of the XL Fortran 13.1 for Linux (see Figure 2). We used 4 benchmark programs for our evaluation: Poisson, 2-dimensional LU from the NAS Parallel Benchmarks Suite (Version 3.2), SOR algorithm and 2-dimensional Jacobi computation. We manually applied array privatization for some loops in blts and buts, for which the compiler failed to automatically privatize the arrays. All these benchmarks are excellent candidates for DOACROSS parallelization. All benchmarks were compiled with option "-O5" for the sequential baseline, and "-O5 -qsmp" for the automatic parallelization enabling DOACROSS parallelization. We evaluated four experimental variants: a) **only doall** represents the speedup where the automatic DOACROSS parallelization is turned off and uses only DOALL parallelism (far left), b) **doall w/ manual skew** represents the speedup with DOALL loops including DOACROSS loops which were converted to DOALL loops after manual loop skewing (second left), c) **doall + doacross (w/o cost-analysis)** is the speedup where both DOALL and DOACROSS parallelization are enabled, but with the cost analysis and granularity control turned off (second right), and d) **doall + doacross** is the speedup where both DOALL and DOACROSS parallelization with cost analysis and granularity control are enabled (far right).

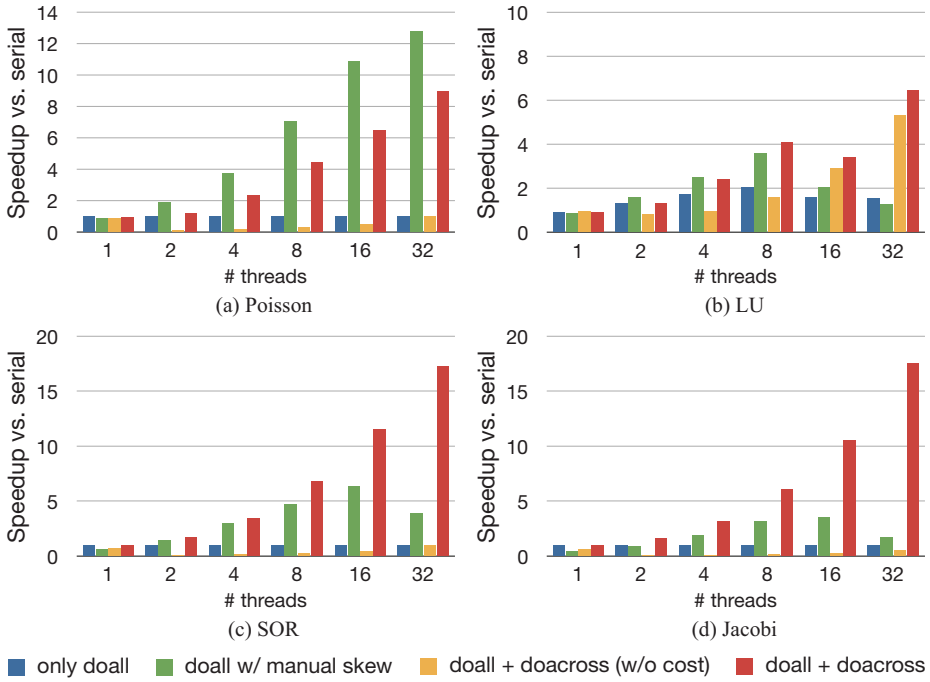


Fig. 3: Speedup related to sequential run on Power7

5.1 POISSON

We use the POISSON kernel discussed in Section 3.1. The DOACROSS loop is invoked 20 times in this experiment. Figure 3a shows the speedup of the 4 variants listed above when compared to the sequential execution.

The **only doall** case results in no speedup; **doall + doacross (w/o cost-analysis)** can results in worse performance than sequential execution because of the large synchronization overhead. **doall + doacross** delivers a speedup of up to 9.0×. Note that **doall w/ manual skew** shows better performance than **doall + doacross**. This is because the POISSON kernel is a triply nested DOACROSS loop. We manually selected the outermost and middle-nested loops for the target of loop skewing and this choice turns out to have a better granularity. On the other hand, the auto DOACROSS version inserted POST/WAIT in the innermost loop body and hence the total synchronization overhead became larger despite the granularity control. (In the future, optimizing compilers could reduce this gap by further adjusting the granularity for DOACROSS parallelization.) However, the automatic **doall + doacross** version outperforms the **doall w/ manual skew** version in the other three benchmarks whose kernel loops are doubly nested and both the manual and automatic versions use the same granularity.

5.2 LU

LU has 2 DOACROSS loops in subroutines *blts* and *buts*. Together they account for about 40% of the sequential execution time of LU. They are 2-dimensional (160×160)

DOACROSS loops that are invoked 40160 times. The conservative dependence vector is (1,0) for both *blts* and *buts*, and the corresponding WAIT/POST synchronizations are inserted. Although LU contains many DOALL loops, the best speedup with DOACROSS parallelism disabled is 2.1× using 8 cores. On the other hand, our DOACROSS parallelization brings more scalable performance, up to 6.5× speedup with 32 cores as shown in Figure 3b. The figure also shows the significant impact of the cost analysis and granularity control on performance. Furthermore, the DOACROSS version show even better scalability than the manual DOALL version that converted the DOACROSS loops in *blts* and *buts* into DOALL loops. It is well known that DOACROSS parallelization has better data locality and lower synchronization overhead than DOALL with loop skewing, even when they use same granularity for wavefront parallelism.

5.3 SOR and Jacobi

The kernel loop nest of SOR is a 2-dimensional 20000×10000 loop which is invoked 50 times. Note that the kernel loop of Jacobi has very similar structure as SOR, and the conservative dependence vector for both SOR and Jacobi kernels is (1,0). Our framework extracted DOACROSS parallelism for both cases, and achieves up to 17.3× and 17.5× speedup for SOR and Jacobi, respectively as shown in Figures 3c and 3d. On the other hand, the best speedups when manually converting DOACROSS into DOALL by loop skewing are 6.4× for SOR and 3.5× for Jacobi. The figures also show that the granularity control is essential to obtain scalable speedup using DOACROSS parallelism.

6 Conclusions and Future Work

We presented a novel and practical approach to automatically parallelizing DOACROSS loops for execution on manycore-SMP systems, based on a compiler-and-runtime optimization called *dependence folding*. The proposed framework uses a conservative dependence vector analysis to identify suitable program points where POST/WAIT synchronization operations can be inserted. A profitability analysis is used to guide unrolling and chunking transformations to select an optimized granularity of computation for DOACROSS parallelization. Further, our runtime framework includes a lightweight and space-efficient implementation of point-to-point synchronization for DOACROSS loops.

The proposed framework has been implemented in a development version of the IBM XL Fortran V13.1 commercial parallelizing compiler and runtime system. For four benchmarks where automatic DOALL parallelization was largely ineffective (speedups of under 2×), our implementation delivered speedups of 6.5×, 9.0×, 17.3×, and 17.5× on a 32-core IBM Power7 SMP system, thereby showing that DOACROSS parallelization can be a valuable technique to complement DOALL parallelization.

During the course of our work in enabling DOACROSS parallelization in the XL compilers, we encountered multiple opportunities for future work related to interactions between DOACROSS parallelization and lower-level compiler optimizations. We found cases where the DOACROSS transformation inhibited *software pipelining* (a technique for scheduling instructions to exploit instruction level parallelism in inner loops by overlapping loop iterations). In such cases, it would be desirable to extend the profitability analysis to take the impact on software pipelining into account. As another example,

predictive commoning (an optimization to reuse computations across loop iterations by detecting indexing sequences and unrolling to avoid register copies), if performed earlier, can inhibit the detection of DOACROSS loops. A detailed study of these interactions is part of our planned future work. Other opportunities for future work include deeper analyses for synchronization overhead and parallel efficiency so as to improve accuracy of profitability analysis, and performance comparison against other existing work. As shown in the paper, POST/WAIT operations are well-suited for user annotations and the technique of dependence folding can also be adapted for the explicit parallelization using such annotations. Extensions of the proposed framework to explicit parallelization is another important direction of future work.

Acknowledgements

The authors would like to thank Osamu Gohda from IBM Japan for the hand-pipelined POISSON and LU codes. This work was supported in part by an IBM CAS Fellowship in 2011 and 2012.

References

1. Chen, D.K.: Compiler optimizations for parallel loops with fine-grained synchronization. PhD Thesis (1994)
2. Cytron, R.: Doacross: Beyond vectorization for multiprocessors. Proceedings of the 1986 International Conference for Parallel Processing pp. 836–844 (August 1986)
3. Ding-Kai Chen, Josep Torrellas, P.C.Y.: An efficient algorithm for the run-time parallelization of doacross loops. Proc. Supercomputing 1994 pp. 518–527 (Nov 1994)
4. Gupta, R., Pande, S., Psarris, K., Sarkar, V.: Compilation techniques for parallel systems. Parallel Computing 25(13-14), 1741–1783 (1999)
5. Li, Z.: Compiler algorithms for event variable synchronization. Proceedings of the 5th international conference on Supercomputing, Cologne, West Germany pp. 85–95 (June 1991)
6. Lowenthal, D.K.: Accurately selecting block size at run time in pipelined parallel programs. International Journal of Parallel Programming 28(3), 245–274 (June 2000)
7. Midkiff, S.P., Padua, D.A.: Compiler algorithms for synchronization. IEEE Transactions on computers C-36, 1485–1495 (December 1987)
8. P. Tang, P. Yew, C.Z.: Compiler techniques for data synchronization in nested parallel loop. Proc. of 1990 ACM Intl. Conf. on Supercomputing, Amsterdam pp. 177–186 (June 1990)
9. R. Rajamony, A.L.C.: Optimally synchronizing doacross loops on shared memory multiprocessors. Proc. of Intl. Conf. on Parallel Architectures and Compilation Techniques (Nov 1997)
10. Su, H.M., Yew, P.C.: On data synchronization for multiprocessors. Proc. of the 16th annual international symposium on Computer architecture, Jerusalem, Israel pp. 416–423 (April 1989)
11. V.P. Krothapalli, P.S.: Removal of redundant dependences in doacross loops with constant dependences. IEEE Transactions on Parallel and Distributed Systems pp. 281–289 (Jul 1991)
12. Wolfe, M.: Multiprocessor synchronization for concurrent loops. IEEE Software v.5 n.1, 34–42 (January 1988)
13. Zhang, G., Unnikrishnan, P., Ren, J.: Experiments with auto-parallelizing SPEC2000FP benchmarks. 17th Intl Workshop on Languages and Compilers for Parallel Computing (2004)
14. Zhelong Pan, Brian Armstrong, H.B., Eigenmann, R.: On the interaction of tiling and automatic parallelization. First International Workshop on OpenMP (Wompat) (June 2005)