Expressing DOACROSS Loop Dependences in OpenMP

Jun Shirako[1], Priya Unnikrishnan[2], Sanjay Chatterjee[1], Kelvin Li[2], and Vivek Sarkar[1]

[1] Department of Computer Science, Rice University
[2] IBM Toronto Laboratory

Abstract. OpenMP is a widely used programming standard for a broad range of parallel systems. In the OpenMP programming model, synchronization points are specified by implicit or explicit barrier operations within a parallel region. However, certain classes of computations, such as stencil algorithms, can be supported with better synchronization efficiency and data locality when using *doacross parallelism* with point-to-point synchronization than *wavefront parallelism* with barrier synchronization. In this paper, we propose new synchronization constructs to enable doacross parallelism in the context of the OpenMP programming model. Experimental results on a 32-core IBM Power7 system using four benchmark programs show performance improvements of the proposed doacross approach over OpenMP barriers by factors of 1.4× to 5.2× when using all 32 cores.

1 Introduction

Multicore and manycore processors are now becoming mainstream in the computer industry. Instead of using processors with faster clock speeds, all computers— embedded, mainstream, and high-end systems — are being built using chips with an increasing number of processor cores with little or no increase in clock speed per core. This trend has forced the need for improved productivity in parallel programming models. A major obstacle to productivity lies in the programmability and performance challenges related to coordinating and synchronizing parallel tasks. Effective use of barrier and point-to-point synchronization are major sources of complexity in that regard. In the OpenMP programming model [1, 2], synchronization points are specified by implicit or explicit barrier operations, which force all parallel threads in the current parallel region to synchronize with each other[1]. However, certain classes of computations such as stencil algorithms require to specify synchronization only among particular iterations so as to support *doacross parallelism* [3] with better synchronization efficiency and data locality than *wavefront parallelism* using all-to-all barriers.

In this paper, we propose new synchronization constructs to express cross-iteration dependences of a parallelized loop and enable doacross parallelism in

[1] This paper focuses on extensions to OpenMP synchronization constructs for parallel loops rather than parallel tasks.

the context of the OpenMP programming model. Note that the proposed constructs aim to express ordering constraints among iterations and we do not distinguish among flow, anti and output dependences. Experimental results on a 32-core IBM Power7 system using numerical applications show performance improvements of the proposed doacross approach over OpenMP barriers by the factors of 1.4–5.2 when using all 32 cores.

The rest of the paper is organized as follows. Section 2 provides background on OpenMP and discusses current limitations in expressing iteration-level dependences. This section also includes examples of low-level hand-coded doacross synchronization in current OpenMP programs, thereby providing additional motivation for our proposed doacross extensions. Section 3 introduces the proposed extensions to support cross-iteration dependence in OpenMP. Section 4 discusses the interaction of the proposed doacross extensions with existing OpenMP constructs. Section 5 describes compiler optimizations to reduce synchronization overhead and runtime implementations to support efficient cross-iteration synchronizations. Section 6 presents our experimental results on a 32-core IBM Power7 platform. Related work is discussed in Section 7, and we conclude in Section 8.

2 Background

2.1 OpenMP

In this section, we give a brief summary of the OpenMP constructs [2] that are most relevant to this paper. The `parallel` construct supports the functionality to start parallel execution by creating parallel threads. The number of threads created is determined by the environment variable `OMP_NUM_THREADS`, runtime function `omp_set_num_threads` or `num_threads` clause specified on the `parallel` construct. The `barrier` construct specifies an all-to-all barrier operation among threads in the current `parallel` region[2]. Therefore, each `barrier` region must be encountered by all threads or by none at all. The loop constructs, `for` construct in C/C++ and `do` construct in Fortran, are work-sharing constructs to specify that the iterations of the loop will be executed in parallel. An implicit barrier is performed immediately after the loop region. The implicit barrier may be omitted if a `nowait` clause is specified on the loop directive. Further, a `barrier` is not allowed inside a loop region. The `collapse` clause on a loop directive collapses multiple perfectly nested rectangular loops into a singly nested loop with an equivalent size of iteration space. The `ordered` construct specifies a structured block in a loop region that will be executed in the order of the loop iterations. This sequentializes and orders the code within an `ordered` region while allowing code outside the region to run in parallel. Note that an `ordered` clause must be specified on the loop directive, and the `ordered` region must be executed only once per iteration of the loop.

[2] A region may be thought of as the dynamic or runtime extent of construct - i.e., region includes any code in called routines while a construct does not.

```
#pragma omp parallel for collapse(2) ordered
for (i = 1; i < n-1; i++) {
    for (j = 1; j < m-1; j++) {
        #pragma omp ordered
        A[i][j] = stencil(A[i][j], A[i][j-1], A[i][j+1],
                          A[i-1][j], A[i+1][j]);
} }
                  (a) Ordered construct

#pragma omp parallel private(i2)
{
    for (i2 = 2; i2 < n+m-3; i2++) {    /* Loop skewing */
        #pragma omp for
        for (j = max(1,i2-n+2);j < min(m-1,i2); j++) {
            int i = i2 - j;
            A[i][j] = stencil(A[i][j], A[i][j-1], A[i][j+1],
                              A[i-1][j], A[i+1][j]);
} } }
              (b) Doall with implicit barrier
```

Fig. 1. 2-D Stencil using existing OpenMP constructs: (a) serialized execution using ordered construct, (b) doall with all-to-all barrier after loop skewing

2.2 Expressiveness of Loop Dependences in OpenMP

As mentioned earlier, a `barrier` construct is not allowed within a loop region, and `ordered` is the only synchronization construct that expresses cross-iteration loop dependences among `for`/`do` loop iterations. However, the expressiveness of loop dependence by `ordered` construct is limited to sequential order and does not cover general loop dependence expressions such as dependence distance vectors. Figure 1a shows an example code for 2-D stencil computation using `ordered`. The `collapse` clause on the `for` directive converts the doubly nested loops into a single nest. This clause is used to ensure that the `ordered` region is executed only once per iteration of the parallel loop, as required by the specifications. Although the dependence distance vectors of the doubly nested loop are (1,0) and (0,1) and hence it has doacross parallelism, the `ordered` construct serializes the execution and no parallelism is available as shown in Figure 2a. An alternative way to exploit parallelism is to apply loop skewing so as to convert doacross parallelism into doall in wavefront fashion. Figure 1b shows the code after loop skewing and parallelizing the inner j-loop using a `for` construct, which is followed by an implicit barrier. As shown in Figure 2b, the major performance drawback of this approach is using all-to-all barrier synchronizations, which are generally more expensive than point-to-point synchronizations used for doacross. Further, this requires programmer expertise in loop restructuring techniques - i.e., selecting correct loop skewing factor and providing skewed loop boundaries.

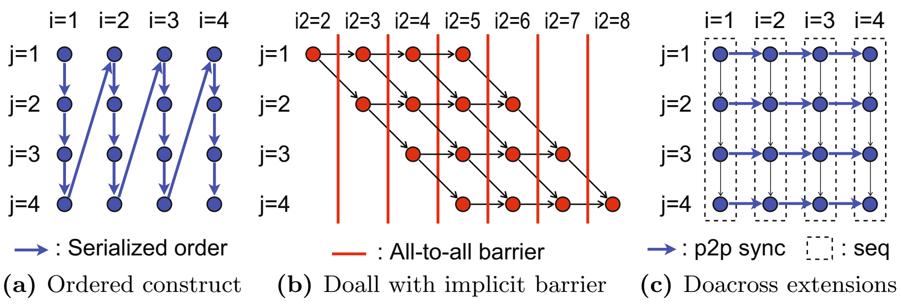

Fig. 2. Synchronization Pattern for 2-D Stencil

2.3 Examples of Hand-Coded Doacross Synchronization in Current OpenMP Programs

Some expert users provide customized barriers/point-to-point synchronizations based on busy-wait local spinning implementations [4] using additional volatile variables for handling synchronization. Although such customized implementations can bring fully optimized performance, they require solid knowledge of parallel programming and the underlying system and are easy to introduce error and/or potential deadlock.

Examples can be found in the OpenMP version of the NAS Parallel Benchmark [5, 6], which is a widely used HPC benchmark suite since 1992. E.g., in NPB3.3.1/NPB3.3-OMP/LU, the pipelining of the SSOR algorithm is achieved by point-to-point synchronizations through extra synchronization variables, busy-waiting, and `flush` directive for memory consistency. The code utilizes OpenMP library functions, `threadprivate` directive in addition to the loop construct with `schedule(static)` and `nowait` clauses. The end result is code that is non-intuitive and unduly complicated.

Further, the LU implementation in NPB reveals a non-compliant usage of `nowait`. Although it makes an assumption that a `nowait` is always enforced, the OpenMP standard states that *"... If a* `nowait` *clause is present, an implementation* **may** *omit the barrier at the end of the worksharing region."*. Because of "may", implementations are allowed to ignore the `nowait` and introduce a barrier. However, if this happens, then the LU implementation will deadlock.

Our proposal aims to address all those issues regarding complexity and deadlock avoidance while keeping the synchronization efficiency via point-to-point synchronizations.

3 New Pragmas for Doacross Parallelization

This section introduces our proposed OpenMP extensions to express general cross-iteration loop dependences and thereby support doacross parallelization. Due to space limitations, we focus on the pragma syntax for C/C++ in this paper, although our proposal is naturally applicable to both C/C++ and Fortran.

The proposed doacross annotations consist of `nest` clause to specify target loops of doacross parallelization and `post`/`await` constructs to express source/ sink of cross-iteration dependences. Ideally, an `await` construct works as a blocking operation that waits for `post` constructs in specific loop iterations. The `post` construct serves as the corresponding unblocking operation. According to the OpenMP terminology, loops that are affected by a loop directive are called *associated loops*. For ease of presentation, we will call the associated loops that are the target of doacross parallelization as a *doacross loop nest*. According to the OpenMP specification [2], all the associated loops must be perfectly nested and have canonical form.

- **nest clause:**
  The "`nest`(n)" clause, which appears on a loop directive, specifies the nest-level of a doacross loop nest. Although `nest` clause defines associated loops as with `collapse` clause, there are two semantical differences from `collapse` clause: 1) `nest` clause is an informational clause and does not necessarily imply any loop restructuring, and 2) `nest` clause permits triangular/trapezoidal loops, whereas `collapse` clause is restricted to rectangular loops.
- **await construct:**
  The "`await depend`(*vect*)[[,] `depend`(*vect*)...]" construct specifies the source iteration vectors of the cross-iteration dependences. There must be at least one `depend` clause on an `await` directive. The current iteration is blocked until all the source iterations specified by *vect* of `depend` clauses[3] execute their `post` constructs. Based on OpenMP's default sequential semantics, the loop dependence vector defined by `depend` clause must be lexicographically positive, and we also require that the dependence vector is constant at compile-time so as to simplify the legality check. Therefore, we restrict the form of *vect* to $(x_1 - d_1, x_2 - d_2, ..., x_n - d_n)$, where n is the dimension specified by the `nest` clause, x_i denotes the loop index of i-th nested loop, and d_i is a constant integer for all $1 \leq i \leq n$ [4]. The dependence distance vector is simply defined as $(d_1, d_2, ..., d_n)$. If the *vect* indicates an invalid iteration (i.e., if vector $(d_1, \ldots, d_n)$ is lexicographically non-positive) then the `depend` clause is ignored implying that there is no real cross-iteration dependence. The `await` is a stand-alone directive without associated executable user codes and designates the location where the blocking operation is invoked. Note that at most one `await` construct can exist in the lexical scope[5] of the loop body of a doacross loop nest.
- **post construct:**
  The "`post`" construct indicates the termination of the computation that causes loop dependences from the current iteration. This stand-alone

[3] The `depend` clause is also under consideration for expressing "inter-task" dependences for `task` construct in OpenMP 4.0.

[4] It is a simple matter to also permit + operators in this syntax since $x_i + d$ is the same as $x_i - (-d)$.

[5] This means `await` and `post` cannot be dynamically nested inside a function invoked from the loop body because they are closely associated with the induction variables of the doacross loop nest.

```
 1 #pragma omp parallel for nest(2)
 2 for (i = 1; i < n-1; i++) {
 3     for (j = 1; j < m-1; j++) {
 4         #pragma omp await depend(i-1,j) depend(i,j-1)
 5         A[i][j] = stencil(A[i][j], A[i][j-1], A[i][j+1],
 6                           A[i-1][j], A[i+1][j]);
 7 }   }
 8   (a) Iteration-level dependences: explicit await at top and
        implicit post at bottom
 9
10 #pragma omp parallel for nest(2)
11 for (i = 1; i < n-1; i++) {
12     for (j = 1; j < m-1; j++) {
13         int tmp = foo(A[i][j]);
14         #pragma omp await depend(i-1,j) depend(i,j-1)
15         A[i][j] = stencil(tmp, A[i][j-1], A[i][j+1],
16                           A[i-1][j], A[i+1][j]);
17         #pragma omp post
18         B[i][j] = bar(A[i][j]);
19 }   }
20   (b) Statement-level dependences: explicit await before line
        15 and explicit post after line 16
```

Fig. 3. 2-D Stencil with the doacross extensions

directive designates the location to invoke the unblocking operation. Analogous to `await` construct, the location must be in the loop body of the doacross loop nest. The difference from `await` construct is that there is an implicit `post` at the end of the loop body. Note that the parameter in the `nest` clause determines the location of the implicit `post`. Due to the presence of the implicit `post`, it is legal to have no `post` constructs inserted by users while the explicit `post` is allowed at most once. The implicit `post` becomes no-op when the invocation of the explicit `post` per loop body is detected at runtime. Finally, it is possible for an explicit `post` to be invoked before an `await` in the loop body.

Figure 3 contains two example codes for the 2-D stencil with the doacross extensions that specify cross-iteration dependences $(1, 0)$ and $(0, 1)$. As shown in Figure 3a, programmers can specify iteration-level dependences very simply by placing an `await` construct at the start of the loop body and relying on the implicit `post` construct at the end of the loop body. On the other hand, Figure 3b shows a case in which `post` and `await` are optimally placed around lines 15 and 16 to optimize statement-level dependences and minimize the critical path length of the doacross loop nest. Note that functions foo and bar at lines 13 and 18 do not contribute to the cross-iteration dependences; foo can be executed before the `await` and bar can execute after the `post`. The `post`/`await` constructs semantically specify the source/sink of cross-iteration dependences and allow flexibility on how to parallelize the doacross loop nest.

4 Interaction with Other OpenMP Constructs

This section discusses the interaction of the proposed doacross extensions with the existing OpenMP constructs. We classify the existing constructs into three categories: 1) constructs that cannot be legally used in conjunction with the doacross extensions, 2) constructs that can be safely used with the doacross extensions, and 3) constructs that require careful consideration when used with the doacross extensions.

4.1 Illegal Usage with Doacross

The `nest`, `await` and `post` constructs make sense only in the context of loops. So the usage of these constructs along with the `sections`, `single` and `master` constructs are illegal. Similarly using them with a `parallel` region without an associate loop construct is illegal, e.g., `#pragma omp parallel for nest()` is legitimate but `#pragma omp parallel nest()` is not.

4.2 Safe Usage with Doacross

The `nest`, `await` and `post` constructs are to be used in conjunction with loops only. The doacross extension can be safely used with the following clauses.

– **private/firstprivate/lastprivate clauses:**
  These are data handling clauses that appear on a loop construct. Because they are only concerned with the data and have not affect on loop scheduling nor synchronization, it is always safe to use with the doacross extensions.
– **reduction clause:**
  This clause appears on a loop directive and specifies a reduction operator, e.g., `+` and `*`, and target variable(s). Analogous to `private` constructs, `reduction` clause can be safely combined with the doacross constructs.
– **ordered construct:**
  The `ordered` clause to appear on a loop directive serializes loop iterations as demonstrated in Section 2. The `ordered` construct specifies a statement or structured block to be serialized in the loop body. Because loop dependences specified by `await` constructs are lexicographically positive, any doacross dependence does not go against with the sequential order by the `ordered` construct, and hence the combination of `ordered` and `await` constructs creates no theoretical deadlock cycle.
– **collapse clause:**
  The `collapse` clause attached on a loop directive is to specify how many loops are associated with the loop construct, and the iterations of all associated loops are collapsed into one iteration space with equivalent size. We allow the combination of `collapse` clause and `nest` clause in the following manner. When `collapse`(m) and `nest`(n) clauses are specified on a loop nest whose nest-level is l, the loop transformation due to `collapse` clause is first processed and the original l-level loop nest is converted into

a $(l - m + 1)$-level loop nest. Then, the `nest(`n`)` clause and corresponding `post`/`await` constructs are applied to the resulting loop nest. Note that $l \geq m + n - 1$, otherwise it results in a compile-timer error.

– **schedule clause:**
The `schedule` clause attached on a loop directive is to specify how the parallel loop iterations are divided into chunks, which are contiguous nonempty subsets, and how these chunks are distributed among threads. The `schedule` clause supports several loop scheduling kinds: `static`, `dynamic`, `guided`, `auto` and `runtime`, and allows users to specify the chunk size. As discussed in Section 5, any scheduling kind and chunk size are safe to use with the doacross extensions.
– **task construct:**
The `task` construct to define an explicit task is available within a loop region. Analogous to `ordered` construct, we prohibit a `post` construct from being used within a `task` region since such a `post` will have a race condition with the implicit `post` at the loop body end.
– **atomic/critical constructs:**
The `atomic` and `critical` constructs to support atomicity and mutual exclusion respectively can be used within a loop region. In order to avoid deadlock, we disallow a `critical` region to contain an `await` construct as with `ordered` construct.
– **simd construct:**
The `simd` construct, which will be introduced in the OpenMP 4.0, can be applied to a loop to indicate that the loop can be transformed into a SIMD loop (that is, multiple iterations of the loop can be executed concurrently using SIMD instructions). We disallow `simd` clause and `nest` clause from appearing on the same loop construct due to conflict in semantics, i.e., `nest` clause implies loop dependence while `simd` mentions SIMD parallelism. Instead, we allow SIMD loop(s) to be nested inside a doacross loop nest.

4.3 Constructs Requiring Careful Consideration

The lock routines supported in the OpenMP library can cause a deadlock when interacted with the `await` construct, especially under the following situation: 1) an `await` construct is located between lock and unlock operations and 2) a `post` construct is located after the lock operation. It is user's responsibility to avoid such deadlock situations due to the interaction.

5 Implementation

This section describes a simple and efficient implementation approach for the proposed doacross extensions to OpenMP; however, other implementation approaches are possible as well. Our approach only parallelizes the outermost loop of the doacross loop nest and keeps inner loops as sequential to be processed by a single thread. The loop dependences that cross the parallelized iterations are enforced via runtime point-to-point synchronizations. Figure 2c shows the synchronization pattern of this approach for the doacross loop in Figure 3, where the outer i-loop is parallelized and its cross-iteration dependence, $(1, 0)$, is preserved by the point-to-point synchronizations. In order to avoid deadlock due to the point-to-point synchronization, the runtime loop scheduling must satisfy a condition that an earlier iteration of the parallel loop is scheduled to a thread before a later iteration. According to the OpenMP Specification (lines 19–21 in page 49 for 4.0 RC2) [2], this condition is satisfied by any OpenMP loop scheduling policy. Note that the same condition is necessary for `ordered` clause to avoid deadlock. Further, any chunk size can be used without causing deadlock. However, chunk sizes greater than the dependence distance of the parallel loop significantly reduce doacross parallelism. Therefore, we assume that the default chunk size for doacross loops is 1. Further, the default loop schedule is `static` so as to enable the lightweight version of synchronization runtime as discussed in Section 5.2.

5.1 Compiler Supports for Doacross Extension

The major task for compilers is to check the legality of the annotated information via the `nest` and `post/await` constructs, and convert the information into runtime calls to POST/WAIT operations. Further, we introduce a compile-time optimization called *dependence folding* [7], which integrates the specified cross-iteration dependences into a conservative dependence vector so as to reduce the runtime synchronizations [7].

Legality Check and Parsing for Doacross Annotations: As with `collapse` clause, the loop nest specified by `nest` clause must be perfectly nested and have canonical loop form [2]. To verify `nest` clauses, we can reuse the same check as `collapse` clause. The legality check for `await`, `depend` and `post` constructs ensure 1) at most one `post/await` directive exists at the nest-level specified by the `nest` clause, 2) the dependence vector of a `depend` clause is lexicographically positive and constant, and 3) the dimension of the dependence vector is same as the parameter of the `nest` clause.

After all checks are passed and the optimization described in next paragraph is applied, the doacross information is converted into the POST and WAIT runtime calls. The locations for these calls are same as the `post/await` constructs. The argument for the POST call is the current iteration vector $(x_1, x_2, ..., x_n)$, while the argument for the WAIT call is defined as $(x_1 - c_1, x_2 - c_2, ..., x_n - c_n)$ by using the conservative dependence vector discussed below.

Dependence Folding: In order to reduce the runtime synchronization overhead, we employ dependence folding [7] that integrates the multiple cross-iteration dependences specified by the `await` construct into a single conservative dependence. First, we ignore dependence vectors whose first dimension - i.e., the dependence distance of the outermost loop - is zero because such dependences are always preserved by the single thread execution. The following discussion assumes that any dependence vector has a positive value in the first dimension in addition to the guarantee of constant dependence vectors.

For an n-th nested doacross loop with k dependence distance vectors, let $\boldsymbol{D^i} = (d_1^i, d_2^i, ..., d_n^i)$ denote the i-th dependence vector ($1 \leq i \leq k$). We define the conservative dependence vector $\boldsymbol{C} = (c_1, c_2, ..., c_n)$ of all k dependences as follow.

$$\boldsymbol{C} = \begin{pmatrix} C[1] : (c_1) \\ C[2..n] : (c_2, c_3, ..., c_n) \end{pmatrix} = \begin{pmatrix} gcd(d_1^1, d_1^2, ..., d_1^k) \\ min\_vect(D^1[2..n], D^2[2..n], ..., D^k[2..n]) \end{pmatrix}$$

Because the outermost loop is parallelized, the first dimension of $\boldsymbol{D^i}$, d_1^i, denotes the stride of dependence across parallel iterations. Therefore, the first dimension of $\boldsymbol{C}$ should correspond to the GCD value of d_1^1, d_1^2, ..., d_1^k. The remaining dimensions, $\boldsymbol{C}[2..n]$, can be computed as the lexicographical minimum vector of $D^1[2..n]$, $D^2[2..n]$, ..., $D^k[2..n]$ because such a minimum vector and the sequential execution of inner loops should preserve all other dependence vectors. After dependence folding, the conservative dependence $\boldsymbol{C}$ is used for the POST call as described in the previous paragraph.

5.2 Runtime Supports for POST/WAIT Synchronizations

This section briefly introduces the runtime algorithms of the POST/WAIT operations. When the loop schedule kind is specified as `static`, which is the default for doacross loops, we employ the algorithms introduced in our previous work [7]. For other schedule kinds such as `dynamic`, `guided`, `auto`, and `runtime`, we use the following simple extensions. Figure 4 shows the pseudo codes for the extended POST/WAIT operations. To trace the POST operations, we provides a 2-dimensional synchronization field $sync\_vec[lw : up][1 : n]$, where lw/up is the lower/upper bound of the outermost loop and n is the nest-level of the doacross loop nest. Because the iteration space of the outermost loop is parallelized and scheduled to arbitrary threads at runtime, we need to trace all the parallel iterations separately while the status of an iteration i ($lw \leq i \leq up$) is represented by $sync\_vec[i][1 : n]$. During the execution of inner loops by a single thread, the thread keeps updating the $sync\_vec[i]$ via the POST operation with the current iteration vector $pvec$, while the WAIT operation is implemented as a local-spinning until the POST operation corresponding to the current WAIT is done - i.e., $sync\_vec[i]$ becomes greater or equal to the dependence source iteration vector $wvec$. As shown at Line 9 of Figure 4, the WAIT operation becomes no-op if the $wvec$ is outside the legal loop boundaries.

6 Experimental Results

In this section, we present the experimental results for the proposed doacross extensions in OpenMP. The experiments were performed on a Power7 system with 32-core 3.55GHz processors running Red Hat Enterprise Linux release 5.4.

```
volatile int sync_vec[lw:up][1:n];

void post(int pvec[1:n]) {
    int i = pvec[1];            /* Outermost loop index value */
    for (int j = n; j > 0; j--) sync_vec[i][j] = pvec[j];
}

void wait(int wvec[1:n]) {
    if (outside_loop_bounds(wvec)) return; /* invalid await */
    int i = wvec[1];            /* Outermost loop index value */
    while (vector_compare(sync_vec[i], wvec]) < 0) sleep();
}
```

Fig. 4. Pseudo codes for POST and WAIT operations

The measurements were done using a development version of the XL Fortran 13.1 for Linux, which supports automatic doacross loop parallelization in addition to doall parallelization. Although we use the Fortran compiler and benchmarks for our experiments, essential functionalities to support the doacross parallelization are also common for any C compilers. We used 4 benchmark programs for our evaluation: SOR and Jacobi, which are variants of the 2-dimensional stencil computation, Poisson computation, and 2-dimensional LU from the NAS Parallel Benchmarks Suite (Version 3.2). 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 parallel executions. a) omp doacross is the speedup where the doacross parallelism is enabled by the proposed doacross extensions (right), b) omp existing is the speedup where the same doacross parallelism is converted into doall parallelism via manual loop skewing and parallelized by the OpenMP loop construct (center), and c) auto par represents the speedup where the automatic parallelization for doall and doacross loops by the XL compiler is enabled (left). As shown below, auto par does not always find the same doacross parallelism as omp doacross. We used default schedule kind and chunk size, i.e., `static` with chunk size = 1 for omp doacross and auto par, and `static` with no chunk size specified for omp existing.

6.1 SOR and Jacobi

Figure 5 shows the kernel computation of SOR, which repeats `mjmax`×`mimax` 2-D stencil by `nstep` times. We selected `nstep` = 10000, `mjmax` = 10000 and `mimax` = 100 so as to highlight the existing OpenMP performance with loop skewing. Jacobi has a very similar computation to SOR and both have the same pattern of cross-iteration dependences. For the proposed doacross extensions, we specified the outermost `l`-loop and middle `j`-loop as doubly nested doacross loops with cross-iteration dependences (1,-1) and (0,1). For the existing OpenMP parallelization, we converted the same doacross parallelism into doall via loop skewing.

```
!$omp parallel do nest(2)
      do 10 l = 1, nstep
       do 10 j = 2, mjmax
!$omp await(l-1,j+1) await(l,j-1)
        do 10 i = 2, mimax
          p(i,j)=(p(i,j)+p(i+1,j)+p(i-1,j)+p(i,j+1)+p(i,j-1))/5
 10   continue
!$omp end parallel do
```

Fig. 5. SOR Kernel

Figure 6 shows the speedups of the three versions listed above when compared to the sequential execution. As shown in the Figure 6a and 6b, omp doacross has better scalability than omp existing for both SOR and Jacobi despite of the same degree of parallelism. This is mainly because the doacross version enables point-to-point synchronizations between neighboring threads, which is more efficient than the all-to-all barrier operations by the existing approach. The version of auto par applied doacross parallelization to the middle `j`-loop and innermost `i`-loop; the scalability is worse than the manual approaches due to the finer-grained parallelism. Note that the outermost `l`-loop is time dimension and difficult for compilers to automatically compute dependence distance vectors. The enhanced dependence analysis in the XL compiler should be addressed in future work

6.2 Poisson

The kernel loop of Poisson is also a triply nested doacross loop with size of 400×400×400. As with SOR and Jacobi, omp doacross and omp existing use the doubly nested doacross parallelism of the outermost and middle loops, and the innermost loop is executed without any synchronization. Figure 6c shows the doacross version has better scalability due to the point-to-point synchronizations. On the other hand, although auto par exactly detected all the dependence distance vectors and applied doacross parallelization at the outermost loop level, it parallelized the loop nest as a triply nested doacross and inserted the POST/WAIT synchronizations at the innermost loop body. Note that auto par applies compile-time and runtime granularity controls (loop unrolling and POST canceling, respectively) based on the cost estimation [7]. However, selecting the doacross nest-level as 2 brought more efficiency for the manual versions as shown in Figure 6c. The automatic selection of doacross nest-level is another important future work for the doacross parallelization by the XL compiler.

6.3 LU

LU has 2 doacross loop nests in subroutines `blts` and `buts`, which are 160×160 doubly nested doacross loops and account for about 40% of the sequential execution time. As observed for other benchmarks, omp doacross has better scalability than omp existing due to the synchronization efficiency. For the case of LU, both omp doacross and auto par use the same doacross parallelism. A difference is that

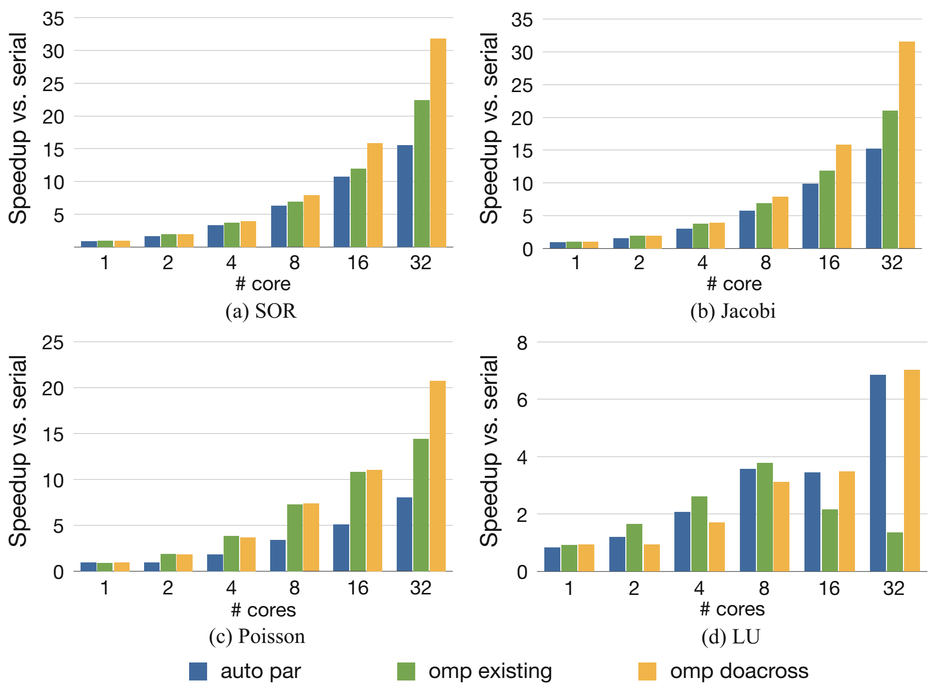

Fig. 6. Speedup related to sequential run on Power7

the granularity control is disabled for `omp doacross` since we should not assume such optimizations in the language specification. However, the execution cost for the doacross loop bodies in `blts` and `buts` is not small; disabling granularity control did not result in large performance degradation with up to 8 cores, and even better/same performance was shown with 32/16 cores because increasing granularity can also affect the amount of parallelism.

7 Related Work

There is an extensive body of literature on doacross parallelization and point-to-point synchronization. In this section, we focus on past contributions that are most closely related to this paper.

Some of the seminal work in synchronization mechanism was done by Padua and Midkiff [8, 9], where they focused on synchronization techniques for single-nested doacross loops using synchronization variable per loop dependence. MPI supports several functions for point-to-point synchronization and communication among threads, such as `MPI_send` and `MPI_recv`. As a recent research outcome, phasers in the Habanero project [10] and `java.util.concurrent.Phaser` from Java 7, which was influenced by Habanero phasers [11], support point-to-point synchronizations among dynamically created tasks. Further, a large amount of existing work on handling non-uniform cross-iteration dependences at runtime [12–15] have been proposed.

There is also a long history on doacross loop scheduling and granularity control [3, 16–18] and compile-time/runtime optimizations for synchronizations [19–21]. These techniques are applicable to the proposed doacross extensions.

8 Conclusions

This paper proposed new synchronization constructs to express cross-iteration dependences of a parallelized loop and enable doacross parallelism in the context of OpenMP programming model. We introduced the proposed API designs and detailed semantics, and discussed the interaction with the existing OpenMP constructs. Further, we described the fundamental implementations for compilers and runtime libraries to support the proposed doacross extensions. Experimental results on a 32-core IBM Power7 system using numerical applications show performance improvements of the proposed doacross approach over existing OpenMP approach with additional loop restructuring by factors of 1.4–5.2 when using all 32 cores. Opportunities for future research include performance experiments with different program sizes and platforms, explorations for the combination with other OpenMP features, e.g., `simd` and `task` constructs, and generalization of point-to-point synchronization aiming for the support of task dependence in OpenMP 4.0.

References

1. Dagum, L., Menon, R.: OpenMP: An industry standard API for shared memory programming. IEEE Computational Science & Engineering (1998)
2. OpenMP specifications, `http://openmp.org/wp/openmp-specifications`
3. Cytron, R.: Doacross: Beyond vectorization for multiprocessors. In: Proceedings of the 1986 International Conference for Parallel Processing, pp. 836–844 (1986)
4. Mellor-Crummey, J., Scott, M.: Algorithms for Scalable Synchronization on Shared Memory Multiprocessors. ACM Transactions on Computer Systems 9(1), 21–65 (1991)
5. N. A. S. Division, NAS Parallel Benchmarks Changes, `http://www.nas.nasa.gov/publications/npb_changes.html#url`
6. Jin, H., Frumkin, M., Yan, J.: The openmp implementation of nas parallel benchmarks and its performance. Tech. Rep. (1999)
7. Unnikrishnan, P., Shirako, J., Barton, K., Chatterjee, S., Silvera, R., Sarkar, V.: A practical approach to doacross parallelization. In: International European Conference on Parallel and Distributed Computing, Euro-Par (2012)
8. Padua, D.A.: Multiprocessors: Discussion of sometheoretical and practical problems. PhD thesis, Department of Computer Science, University of Illinois, Urbana, Illinois (October 1979)
9. Midkiff, S.P., Padua, D.A.: Compiler algorithms for synchronization. IEEE Transactions on Computers C-36, 1485–1495 (1987)
10. Shirako, J., et al.: Phasers: a unified deadlock-free construct for collective and point-to-point synchronization. In: ICS 2008: Proceedings of the 22nd Annual International Conference on Supercomputing, pp. 277–288. ACM, New York (2008)

11. Miller, A.: Set your Java 7 Phasers to stun (2008), `http://tech.puredanger.com/2008/07/08/java7-phasers/`
12. Su, H.-M., Yew, P.-C.: On data synchronization for multiprocessors. In: Proc. of the 16th Annual International Symposium on Computer Architecture, Jerusalem, Israel, pp. 416–423 (April 1989)
13. Tang, C.Z.P., Yew, P.: Compiler techniques for data synchronization in nested parallel loop. In: Proc. of 1990 ACM Intl. Conf. on Supercomputing, Amsterdam, Amsterdam, pp. 177–186 (June 1990)
14. Li, Z.: Compiler algorithms for event variable synchronization. In: Proceedings of the 5th International Conference on Supercomputing, Cologne, West, Germany, pp. 85–95 (June 1991)
15. Ding-Kai Chen, P.-C.Y., Torrellas, J.: An efficient algorithm for the run-time parallelization of doacross loops. In: Proc. Supercomputing 1994, pp. 518–527 (1994)
16. Lowenthal, D.K.: Accurately selecting block size at run time in pipelined parallel programs. International Journal of Parallel Programming 28(3), 245–274 (2000)
17. Manjikian, N., Abdelrahman, T.S.: Exploiting wavefront parallelism on large-scale shared-memory multiprocessors. IEEE Transactions on Parallel and Distributed Systems 12(3), 259–271 (2001)
18. Pan, Z., Armstrong, B., Bae, H., Eigenmann, R.: On the interaction of tiling and automatic parallelization. In: Mueller, M.S., Chapman, B.M., de Supinski, B.R., Malony, A.D., Voss, M. (eds.) IWOMP 2005 and IWOMP 2006. LNCS, vol. 4315, pp. 24–35. Springer, Heidelberg (2008)
19. Krothapalli, P.S.V.P.: Removal of redundant dependences in doacross loops with constant dependences. IEEE Transactions on Parallel and Distributed Systems, 281–289 (July 1991)
20. Chen, D.-K.: Compiler optimizations for parallel loops with fine-grained synchronization. PhD Thesis (1994)
21. Rajamony, A.L.C.R.: Optimally synchronizing doacross loops on shared memory multiprocessors. In: Proc. of Intl. Conf. on Parallel Architectures and Compilation Techniques (November 1997)