On Automatic Parallelization of Irregular Reductions on Scalable Shared Memory Systems[⋆]

E. Gutiérrez, O. Plata, and E.L. Zapata

Department of Computer Architecture, University of Málaga,
P.O. Box 4114, E-29080 Málaga, Spain
{eladio,oscar,ezapata}@ac.uma.es

Abstract. This paper presents a new parallelization method for reductions of arrays with subscripted subscripts on scalable shared-memory multiprocessors. The mapping of computations is based on the conflict-free write distribution of the reduction vector across the processors. The proposed method is general, scalable, and easy to implement on a compiler. A performance evaluation and comparison with other existing techniques is presented. From the experimental results, the proposed method is a clear alternative to the array expansion and privatized buffer methods, usual on state-of-the-art parallelizing compilers, like Polaris or SUIF.

1 Introduction and Background

Irregular reduction operations are frequently found in the core of many large scientific and engineering applications. Figure 1 shows simple examples of reduction loops (*histogram* reduction [10]), with a single reduction vector, A(), updated through single or multiple subscript arrays, f1(), f2(). Due to the loop-variant nature of the subscript array/s, loop-carried dependences may be present. It is usual that this reduction loop is executed many times, in an iterative process. The subscript array/s may be static (unmodified) during all the computation, or may change, usually slowly, through the iterative process.

A general approach to parallelize irregular codes, including reductions, is based on the *inspector-executor model* [9]. However, this strategy is usually highly inefficient as introduces significant overheads due to its generality. A more specific and efficient method may be developed if data affinity is exploited, as in the (LocalWrite) technique [5] (although it is not reported as a clear good alternative to parallelize irregular reductions).

In a shared-memory context, academic parallelizers like Polaris [2] and SUIF [4], recognize and parallelize irregular reductions. A number of techniques are available [7, 1]: critical sections, data affinity, privatized buffer (SUIF), array expansion (Polaris) and reduction table. The most efficient techniques, privatized

[⋆] This work was supported by the Ministry of Education and Science (CICYT) of Spain (TIC96-1125-C03)

```
real A(1:ADim)
integer f(1:fDim)

do i = 1, fDim
    r = function(i,f(i))
    A(f(i)) = A(f(i)) + r
end do
```

(a)

```
real A(1:ADim)
integer f1(1:fDim), f2(1:fDim)

do i = 1, fDim
    r = function(i,f1(i),f2(i))
    A(f1(i)) = A(f1(i)) + r
    A(f2(i)) = A(f2(i)) - r
end do
```

(b)

Fig. 1. A single (a) and a multiple (b) irregular histogram reduction

buffer and array expansion, however have scalability problems due to the high memory overhead they exhibit.

Contemporary commercial compilers do not recognize irregular reductions, although, in some cases, they allow to parallelize them, for instance, using data affinity (as in SGI Fortran90 compiler [11]). A similar approach may also be exploited into HPF [6] (for example, using ON HOME) or OpenMP [8]. However, when multiple reduction arrays occur, conditional sentences usually appear inside the reduction loop in order to fulfill the owner compute rule (which may introduce also computation replication). These implementations reduce drastically the performance of the parallel code and compromise its scalability.

We present here a method to parallelize irregular reductions on scalable shared-memory machines (although may be adapted to a message-passing machine), whose efficiency clearly overcomes that of all the previously mentioned techniques. The mapping of computations is based on the conflict-free write distribution of the reduction vector across the processors. The proposed method is general, scalable, and easy to implement on a compiler.

2 Data Write Affinity with Loop-Index Prefetching

2.1 Single Reductions

Array expansion is based on the domain decomposition of the histogram reduction loop (that is, the [1:fDim] domain). This way, and due to the irregular data access pattern to the reduction vector through f() (see Fig. 2 (a)), private copies of such vector are needed (high memory overhead for large domains, and cache locality problem). Such private buffers can be avoided if the domain decomposition of the loop is substituted for a data decomposition of the reduction vector. The reduction vector may be, for instance, block distributed across the processors. Afterwards, the computations of the histogram loop are arranged in such a way that each processor only computes those iterations that update owned reduction vector entries. Data distribution of the reduction vector may be carried out at compile time (using some compiler or language directive), or at runtime, as a consequence of the arrangement of the loop iterations.

A simple form to implement this computation arrangement is called *data affiliated loop* in [7]. Each processor traverses all the iterations in the reduction

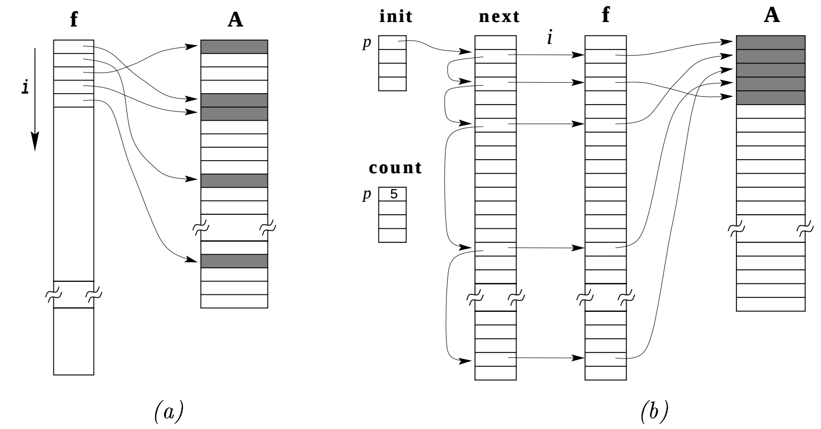


Fig. 2. Graphical depiction for irregular data accesses to the reduction vector (a), and the write data affinity-based access to it by using a loop-index prefetching array (b)

loop and checks whether it owns the reduction vector element referenced in the current iteration. If such case, the iteration is executed; otherwise, the iteration is skipped. The above implementation is not efficient for large iteration domains. A better approach consists in building a loop-index prefetching array, that contains, for each processor, the set of iterations that writes those reduction vector elements assigned to it. In the code in Fig. 3 (a), the loop-index prefetching array is implemented using three arrays, init(), count() and next(), that actually represents a linked list that reorders the subscript array f() (see Fig. 2 (b)). Each processor (thread) has an associated entry in both arrays, init() and count(). The entry in init() points to the first entry in next() owned by that processor (that is, whose index corresponds to a reduction loop iteration that writes in an owned reduction vector element). That entry in next() contains a pointer to the next element that is also owned by the same processor. The process is repeated the number of times stored in count().

In Fig. 3 (b) a simple code that implements the loop-index prefetching array is presented. The second loop in this code contains a histogram reduction on the count() array. As the reduction vector has a size given by the number of threads computing the code, it may be parallelized using array expansion without a significant memory overhead.

In a large class of scientific/engineering problems, the subscript array f() is static, that is, it is not modified during the whole execution of the program. In such codes, the loop-index prefetching array is computed only once and reused without modification in all the reduction loops of the program. Some other problems, on the other hand, have a dynamic nature, which is showed in the periodic updating of f(). The prefetching array has to be recalculated periodically, at least, partially. However, as it is usual that the dynamic nature of the problem changes slowly, the overhead of recalculating that array is partially compensated for its reuse in a number of executions of the reduction loop.

```
real A(1:ADim)
integer f(1:fDim)
integer init(1:NumThreads)
integer count(1:NumThreads)
integer next(1:fDim)

doall p = 1, NumThreads
    i = init(p)
    cnt = count(p)
    do k = 1, cnt
        r = function(i,f(i))
        A(f(i)) = A(f(i)) + r
        i = next(i)
    end do
end doall
```

(a)

```
integer prev(1:NumThreads)

BlockSz = floor(ADim/NumThreads) + 1
do p = 1, NumThreads
    count(p) = 0
end do
do i = 1, fDim
    block = (f(i)-1)/BlockSz + 1
    if ( count(block)  .eq. 0 ) then
        init(block) = i
    else
        next(prev(block)) = i
    end if
    prev(block) = i
    count(block) = count(block) + 1
end do
```

(b)

Fig. 3. Parallel single histogram reduction using data write affinity on the reduction vector with loop-index prefetching (a), and the sequential computation of the prefetching array (b)

2.2 Multiple Reductions

Many real codes include irregular reduction loops containing multiple subscript arrays indexing the same reduction vector, as shown in Fig. 1 (b). Our approach can also be applied to this case, but with some modifications. As the reduction vector A() is block distributed among the processors, there can be some iterations in the reduction loop that evaluate the subscript arrays f1() and f2() pointing to two different partitions. As each processor only writes on its partition, then such iterations have to be replicated on both processors.

To solve this problem, we have split the set of iterations of the reduction loop into two subsets. The first subset contains all iterations (local iterations) that reference reduction vector entries belonging to the same partition. The second subset contains the rest of iterations (boundary iterations). This way, all local iterations are executed only once in some processor, while the boundary iterations have to be replicated into two processors.

Figure 4 (a) shows this parallelization strategy applied to the reduction loop of Fig. 1 (b). Two loop-index prefetching arrays have been built, one referencing local iterations (init-l(), count-l() and next1()), and the other referencing boundary iterations. The second prefetching array is further split into two subarrays. The first one, init-b1(), count-b1() and next1(), reorders the subscript array f1() but restricted to boundary iterations. The other subarray is similar but reordering subscript array f2().

To compute those prefetching arrays, we can use a similar code than for single reductions, as shown in Fig. 4 (b). This code also can be parallelized using the expansion array technique (applied to init*() and count*()). Note that, in order to save memory overhead, the first boundary iteration prefetching subarray uses the same next1() array than the local iteration prefetching array. This way, we need only two copies of this large array instead of three (next1() and next2()).

```
real A(1:ADim)
integer f(1:fDim)
integer init-l(1:NumThreads)
integer init-b1(1:NumThreads)
integer init-b2(1:NumThreads)
integer count-l(1:NumThreads)
integer count-b1(1:NumThreads)
integer count-b2(1:NumThreads)
integer next1(fDim), next2(fDim)

doall p = 1, NumThreads
    i = init-l(p)
    cnt = count-l(p)
    do k = 1, cnt
        r = function(i,f1(i),f2(i))
        A(f1(i)) = A(f1(i)) + r
        A(f2(i)) = A(f2(i)) - r
        i = next1(i)
    end do
    i = init-b1(p)
    cnt = count-b1(p)
    do k = 1, cnt
        r = function(i,f1(i),f2(i))
        A(f1(i)) = A(f1(i)) + r
        i = next1(i)
    end do
    i = init-b2(p)
    cnt = count-b2(p)
    do k = 1, cnt
        r = function(i,f1(i),f2(i))
        A(f2(i)) = A(f2(i)) - r
        i = next2(i)
    end do
end doall
```

(a)

```
integer prev(1:NumThreads)

BlockSz = floor(ADim/NumThreads) + 1
do p = 1, NumThreads
    count-l(p) = 0
    count-b1(p) = 0
    count-b2(p) = 0
end do
do i = 1, N
    block1 = (f1(i)-1)/BlockSz + 1
    block2 = (f2(i)-1)/BlockSz + 1
    if ( block1 .eq. block2 ) then
        if ( count-i(block1) .eq. 0 ) then
            init-l(block1) = i
        else
            next1(prev(block1)) = i
        end if
        prev(block1) = i
        count-l(block1) = count-l(block1) + 1
    else
        if ( count-b(1,block1) .eq. 0 ) then
            init-b1(block1) = i
        else
            next1(prev(block1)) = i
        end if
        prev(block1) = i
        count-b1(block1) = count-b(1,block1) + 1
        if ( count-b(2,block2) .eq. 0 ) then
            init-b2(block2) = i
        else
            next2(prev(block2)) = i
        end if
        prev(block2) = i
        count-b2(block2) = count-b(2,block2) + 1
    end if
end do
```

(b)

Fig. 4. Parallel multiple histogram reduction using data write affinity on the reduction vector with loop-index prefetching (a), and the sequential computation of the prefetching arrays (b)

The replication of the boundary iterations introduces an additional computational overhead, which is represented by the third loop inside the doall loop in Fig. 4 (a) and the *else* section of the main *if* in Fig. 4 (b). However, in many realistic applications, there are many more local iterations than boundary ones, and hence, this additional computation overhead is very small.

3 Analysis and Performance Evaluation

The proposed technique has been applied to the code EULER (HPF-2 [3] motivating applications suite), which solves the differential Euler equations on an irregular mesh. In order to avoid other effects, we have selected in our experiments only one of the loops in the code, computing an irregular reduction inside a time-step loop (Fig. 5). The experiments have been conducted on a SGI Origin2000 (32 250MHz R10000 processors with 4 MB L2 cache and 8192 MB main

```
real vel_delta(1:3,numNodes)
integer edge(1:2,numEdges)
real edgeData(1:3,numEdges)
real velocity(1:3,numNodes)

do i = 1, numEdges
    n1 = edge(1,i)
    n2 = edge(2,i)
    r1 = funct1(edgeData(:,i), velocity(:,n1), velocity(:,n2))
    r2 = funct2(edgeData(:,i), velocity(:,n1), velocity(:,n2))
    r3 = funct3(edgeData(:,i), velocity(:,n1), velocity(:,n2))
    vel_delta(1,n1) = vel_delta(1,n1) + r1
    vel_delta(2,n1) = vel_delta(2,n1) + r2
    vel_delta(3,n1) = vel_delta(3,n1) + r3
    vel_delta(1,n2) = vel_delta(1,n2) - r1
    vel_delta(2,n2) = vel_delta(2,n2) - r2
    vel_delta(3,n2) = vel_delta(3,n2) - r3
end do
```

Fig. 5. An irregular reduction loop from the EULER code

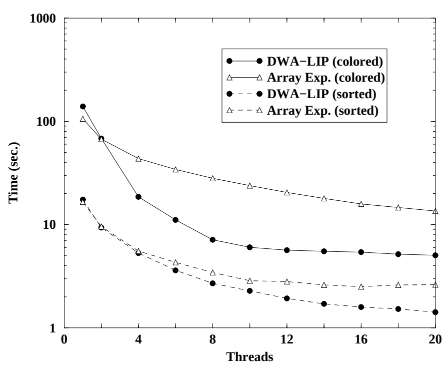
memory), and implemented using the MIPSpro Fortran90 shared-memory directives [11]. The array expansion parallel code was obtained using the Polaris compiler. Parallel codes were compiled using the MIPSpro compiler with optimization level 2.

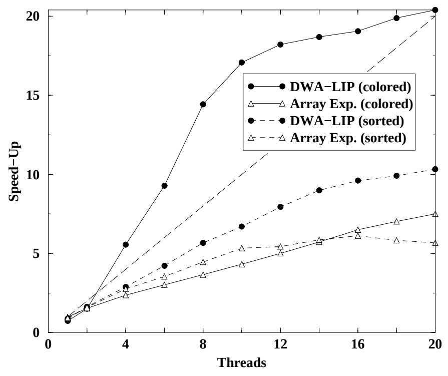
Test irregular meshes have been obtained using the mesh generator included with the EULER code (sizes 783K and 1161K nodes, with connectivity of 8). Two versions of each mesh has been tested: colored and sorted. In the first version, an edge-coloring algorithm has been applied, and the edges of the same color have been placed consecutively in the indirection array. In this case, a low locality in accesses to reduction array would be expected. In the second version, the list of edges has been sorted, improving the access locality.

Fig. 6 depicts the performance for the colored and sorted versions of the mesh of size 1161K nodes. Part (a) shows the execution time (5 iterations of the time-step loop) of both methods, the array expansion and the proposed data write affinity with loop-index prefetching (DWA-LIP). These times excludes the calculation of the prefetching array, as this is done only once, before entering into the iterative loop. Part (b) shows speedups with respect to the sequential code (sequential time is 103.5 sec. and 15.3 sec. for the colored and sorted meshes, respectively). DWA-LIP obtains a significant performance improvement because it exploits efficiently locality when writing in the reduction array.

Fig. 7 shows the efficiencies for the colored (a) and sorted (b) meshes using DWA-LIP. For each class, results from two different mesh sizes show good scalability. The sequential times for the small colored and sorted meshes of sizes 783K nodes are 51.8 sec. and 11.8 sec., respectively. The sequential time costs of computing the loop-index prefetching for both mesh sizes are 2.3 sec. (small mesh) and 3.0 sec. (large mesh). These times are a small fraction of the total reduction time, that can be further reduced parallelizing the code (see Fig. 8).

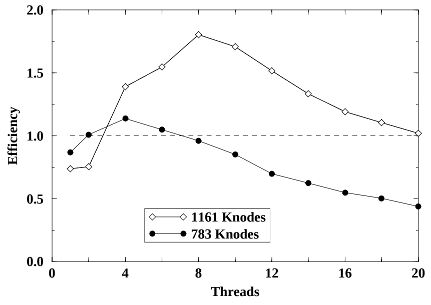
Array expansion has a significant memory overhead due to the replication of the reduction vector in all the processors ($\mathcal{O}(Q * NumNodes * NumThreads)$,

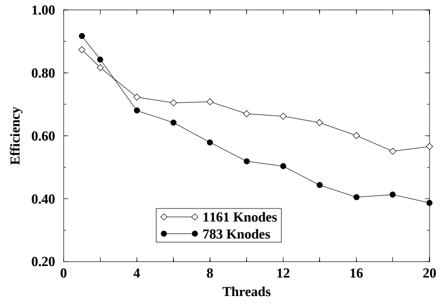




(a) *(b)*

Fig. 6. Parallel execution times (a) and speedups (b) for DWA-LIP and array expansion using colored and sorted meshes with 1161K nodes





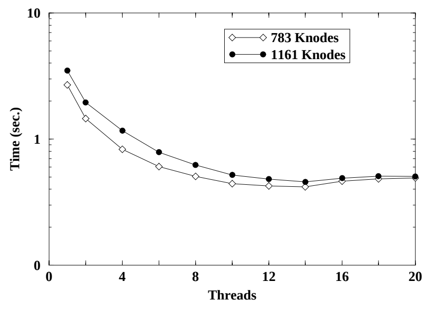
(a) *(b)*

Fig. 7. Parallel efficiencies for colored (a) and sorted (b) meshes using DWA-LIP

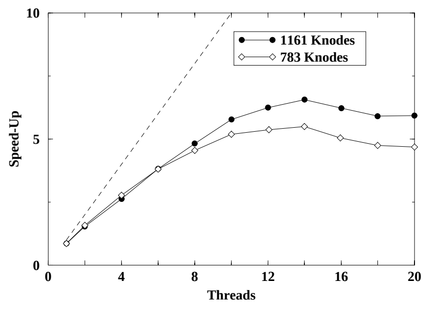
where Q is the number of reduction vectors). DWA-LIP also has memory overhead due to the prefetching array ($\mathcal{O}(K * NumEdges + NumThreads^2)$, where K is the number of subscript arrays). In the EULER reduction loop, $Q = 3$ and $K = 2$, being $numEdges \approx 8 * numNodes$. Hence memory overhead in array expansion is greater than in DWA-LIP when more than 5 threads are used. In the EULER code, there are reduction loops with $Q = 5$ and $K = 4$.

4 Conclusions

The method proposed in this work parallelizes an irregular reduction loop exploiting write locality, similar to [5]. In our approach, the accesses to the subscript arrays are reordered using a linked list, instead of a general inspector-executor mechanism. This fact justifies the better performance and scalability of our method.



(a)



(b)

Fig. 8. Parallel execution times (a) and speedups (b) for the loop-index prefetching

Compared with array expansion, our method does not need to replicate the reduction vector and exploits better data locality. Thus it has better performance and scalability. Despite the overhead of building the prefetching array, however it is computed only once and reused through the whole program. In dynamic systems the prefetching array must be recomputed periodically.

References

1. R. Asenjo, E. Gutierrez, Y. Lin, D. Padua, B. Pottengerg and E. Zapata, *On the Automatic Parallelization of Sparse and Irregular Fortran Codes*, **TR–1512**, Univ. of Illinois at Urbana-Champaign, Ctr. for Supercomputing R&D., Dec. 1996.
2. W. Blume, R. Doallo, R. Eigemann, *et. al.*, *Parallel Programming with Polaris*, **IEEE Computer**, 29(12):78-82, Dec. 1996.
3. I. Foster, R. Schreiber and P. Havlak, *HPF-2, Scope of Activities and Motivating Applications*, *Tech. Rep. CRPC-TR94492*, Rice Univ., Nov. 1994.
4. M.W. Hall, J.M. Anderson, S.P. Amarasinghe, *et. al.*, *Maximizing Multiprocessor Performance with the SUIF Compiler* IEEE Computer, 29(12), Dec. 1996.
5. H. Han and C.-W. Tseng, *Improving Compiler and Run-Time Support for Irregular Reductions*, **11th Workshop on Languages and Compilers for Parallel Computing**, Chapel Hill, NC, Aug. 1998.
6. *High Performance Fortran Language Specification, Version 2.0.* High Performance Fortran Forum, Oct. 1996
7. Y. Lin and D. Padua, *On the Automatic Parallelization of Sparse and Irregular Fortran Programs*, **4th Workshop on Languages, Compilers and Runtime Systems for Scalable Computers**, Pittsburgh, PA, May 1998.
8. *OpenMP: A Proposed Industry Standard API for Shared Memory Programming*, OpenMP Architecture Review Board, `http://www.openmp.org`), 1997.
9. R. Ponnusamy, J. Saltz, A. Choudhary, S. Hwang and G. Fox, *Runtime Support and Compilation Methods for User-Specified Data Distributions*, **IEEE Trans. on Parallel and Distributed Systems**, 6(8):815–831, Jun. 1995.
10. B. Pottenger and R. Eigenmann, *Idiom Recognition in the Polaris Parallelizing Compiler*, **9th ACM Int'l Conf. on Supercomputing**, Barcelona, Spain, pp. 444–448, Jul. 1995.
11. Silicon Graphics, Inc. *MIPSpro Automatic Parallelization.* SGI, Inc. 1998