A Compiler Method for the Parallel Execution of Irregular Reductions in Scalable Shared Memory Multiprocessors

E. Gutiérrez O. Plata E.L. Zapata*
Department of Computer Architecture
University of Málaga
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 grouping reduction loop iterations into sets that are further assigned to the cooperating threads of computation. Iterations belonging to the same set are chosen in such a way that update different entries in the reduction array. That is, the loop distribution implies a conflict-free write distribution of the reduction array. The iteration sets are set up by building a loop-index prefetching data structure that allows to reorder properly the loop iterations. The proposed method is general, scalable, and easy to implement on a compiler. In addition it deals in a uniform way with one and multiple subscript arrays. In case of multiple indirection arrays, writes on the reduction array affecting different sets are solved by defining conflict-free supersets. A performance evaluation is presented. From the experimental results and performance analysis, the proposed method appears as a clear alternative to the array expansion and privatized buffer techniques, used on state-of-the-art parallelizing compilers, like Polaris or SUIF. The scalability problem that those techniques exhibit is missing in our method, as the memory overhead presented does not depend on the number of processors.

1. INTRODUCTION

Parallel computation is increasingly being used as a mean to obtain enough computing power to solve large scientific and engineering problems. The high complexity of these problems, as well as of the current high-performance architectures, strongly support the necessity of developing compiler techniques to efficiently map such applications onto a parallel architecture. However, many of these codes exhibit irregular access patterns to the data. Current commercial compilers [18, 19] are insufficiently developed to deal with this data accesses, leading to low parallel efficiencies when they are used on such programs.

*This work was supported by the Ministry of Education and Culture (CICYT) of Spain (TIC96-1125-C03)

Reduction operations are frequently found in the core of these applications, as in the next simple loop,

```
do i = 1, N
   A(f(i)) = A(f(i)) ⊗ expr
end do
```

The symbol $\otimes$ represents an associative operation (like sum, product, maximum, minimum, ...), and expr is an expression that should not contain any reference to the reduction array A(). The subscript array f() depends on the loop index i, and appears in the right and left hand side of the assignment sentence. Such pattern is termed as *histogram* reduction [17, 11]. Due to the loop-variant nature of the subscript array f(), loop-carried dependences may be present at run-time (if it is not a permutation array).

A general approach to parallelize irregular codes, including reductions, is based on the *inspector-executor model*. CHAOS [16] is a well known implementation of this model for distributed memory machines. Irregular reductions are parallelized by using an inspector to locate non-local data for each processor. Afterwards, an executor must gather non-local data before the reduction, and must scatter the results after it. This strategy introduces a significant overhead due mainly to its generality, caused by the inspector (communication schedule, global-to-local address translation) and the executor (communications and local-to-global address translation). This overhead is proportional to the number of non-local data accesses.

In a shared memory context, academic parallelizers like Polaris [2] and SUIF [7] recognize irregular reductions and parallelize them using the replicated buffer or the array expansion techniques. The first method replicates the reduction array on all the processors. Each processor computes a portion of the reduction on its private buffer. Later a global reduction is obtained by combining all partial reductions, using synchronization to ensure mutual exclusion [6]. Array expansion, on the other hand, expands the reduction array by the number of threads participating in the computation. This approach does not need any synchronization to obtain the global reduction, performing in general better than the replicated buffer method. However, both techniques have

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

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

(a)

```
real A(ADim)
integer f1(fDim), f2(fDim), f3(fDim), ...

do i = 1, fDim
   r = function(i,f1(i),f2(i),f3(i),...)
   A(f1(i)) = A(f1(i)) + r
   A(f2(i)) = A(f2(i)) + r
   A(f3(i)) = A(f3(i)) + r
   ...
end do
```

(b)

```
real A(ADim), Atmp(ADim,NumThreads)
integer f(fDim)
!– Phase 1
c$omp parallel do
   do p = 1, NumThreads
      do t = 1, ADim
         Atmp(t,p) = 0
      end do
   end do
c$omp end parallel
!– Phase 2
c$omp parallel do
   do i = 1, fDim
      r = function(i,f(i))
      Atmp(f(i),MyThreadNum) =
            Atmp(f(i),MyThreadNum) + r
   end do
c$omp end parallel
!– Phase 3
c$omp parallel do
   do t = 1, ADim
      do p = 1, NumThreads
         A(t) = A(t) + Atmp(t,p)
      end do
   end do
c$omp end parallel
```

(c)

Figure 1: ***A single loop with one (a) and multiple (b) irregular histogram reductions, and the parallelization of the first case using array expansion (c)***

scalability problems due to the high memory overhead they exhibit.

Fig. 1 shows a simple example of a histogram reduction loop, with only one subscript array (part (a)) or multiple subscript arrays (part (b)), and with only one reduction array. The computation loops over a number of points in the domain [1:fDIm], calculates the value r, and updates A() in some other different point, in the domain [1:ADim]. It is usual that in real scientific/engineering codes, this reduction loop is executed many times, in an iterative process. The subscript array/s f() or f1(), f2(), f3(), ... may be static (unmodified) during all the computation, or may change, usually slowly, through the iterative process.

Fig. 1 (c) shows the parallel code for the loop in (a) as obtained by the Polaris compiler (actually, a simplified and slightly optimized version is presented), that is, using the array expansion technique. A private copy of the full reduction array (A()) is used for each processor. This is accomplished by expanding such array by the number of threads participating in the computation. The parallel computation is organized around three phases. In phase 1, each processor initializes its own copy of the reduction array (Atmp()). In phase 2, each processor works in parallel on the reduction loop, updating its private copy of the reduction array. Partial reductions are hence computed. Finally, in phase 3, global reduction values are calculated by combining the private copies Atmp() on the global reduction array. With this arrangement, all loop-carried dependences that may exist in the loop due to possible replicated values in the subscript array f() are fulfilled. This method introduces two sources of overhead. Memory overhead due to the private copies of the reduction array, and computing overhead due to the initialization of such buffers and final combining of them on the global reduction array.

In this paper we present a new method to parallelize irregular reductions on scalable shared-memory machines, 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 array across the processors. The proposed method is general, scalable, and easy to implement on a parallelizing compiler.

The reminder of the paper is organized as follows. Next section is devoted to introduce and analyze the approach we propose to parallelize irregular reductions. A comparison and experimental evaluation of our method and the array expansion technique is presented and discussed. The paper is finally concluded with a discussion of related work.

2. SCALABLE PARALLEL IRREGULAR REDUCTIONS

This section describes the parallelization method we propose for irregular reductions. We begin by examining in a uniform way the case of having a single reduction loop containing one or multiple subscript arrays. Afterwards, we consider the more general case when there are multiple subscript arrays contained into multiply nested reduction loops.

2.1 Single Loop Reductions

Array expansion is based on the domain decomposition of the histogram reduction loop (that is, the decomposition of the [1:fDim] domain). This way, and due to the irregular data access pattern to the reduction array through subscript array/s (see Fig. 2 (a) for only one subscript array), private copies of such array are needed. Such private buffers can be avoided (and the corresponding initialization and final combination) if the domain decomposition of the loop is substituted for a data decomposition of the reduction array. The reduction array may be, for instance, block partitioned, assigning each of the resulting blocks to each of the cooperating threads. Afterwards, the computations of the histogram loop are arranged in such a way that each thread only computes those iterations that updates reduction array entries assigned to it. Note that the data distribution of the reduction array 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 (that generates particular memory reference patterns)

A simple form to implement this computation arrangement is called *data affiliated loop* in [12]. Each processor traverses all the iterations in the reduction loop (that is, the [1:fDim] domain) and checks whether the reduction array entry (or entries) referenced in the current iteration has been assigned to it. The owned assignments are executed while the rest of them are simply skipped. That is, the reduction/s in the

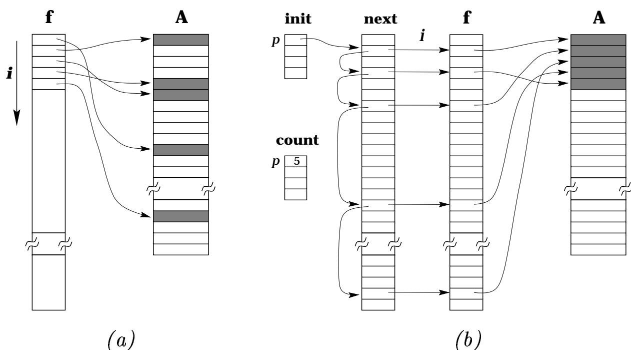

Figure 2: ***Graphical depiction for the irregular data access to the reduction array (a), and the write data affinity-based access to it by using the loop-index prefetching data structure (4 threads are assumed) (b)***

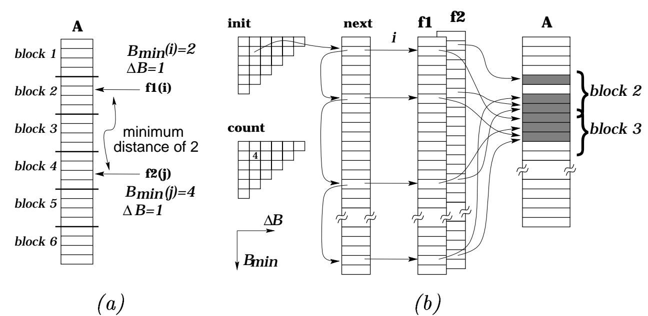

Figure 3: ***Example graphical depiction of how two sets of iterations of a multiple reduction loop can be safely executed in parallel (a), and a data structure implementation for storing such sets (b) (6 threads and 2 subscript arrays are assumed)***

histogram loop is (are) guarded by a conditional.

The above implementation is not efficient for large iteration domains. A better approach consists in exploiting *data write affinity* on the reduction array with the help of a *loop-index prefetching (LIP)* data structure (*DWA-LIP technique*). The LIP structure keeps track the set of iterations that write each one of the blocks of the reduction array (see Fig. 2 (b)). By using this prefetching structure, both private buffers and guard conditionals are avoided.

An efficient and general implementation of such LIP structure can be defined as follows. Let b be the block number function for array A(), $b(k) = \lfloor \frac{(k-1)T}{ADim} \rfloor + 1$, where k is an integer number in the range [1:ADim] and T is the total number of cooperating threads in the system. That is, block of number r is owned by thread r. Given iteration i of the reduction loop, we define B_{min} (B_{max}) as the minimum (maximum) number of all blocks written in such iteration, $B_{min}(i) = \min\{b(f1(i)), b(f2(i)), ...\}$, $B_{max}(i) = \max\{b(f1(i)), b(f2(i)), ...\}$. The difference between both limits is then $\Delta B(i) = B_{max}(i) - B_{min}(i)$.

In order to exploit parallelism from the reduction loop, the iterations are sorted into sets characterized by the pair $(B_{min}, \Delta B)$. Those sets of iterations of the form $(B_{min}, 0)$ are data flow independent and thus can be executed in parallel. In general, two sets of iterations, (b_1, db_1) and (b_2, db_2), are data flow independent if the writing areas in the reduction array are non-overlapping, that is $b_1 + db_1 < b_2$. Fig. 3 (a) depicts a graphical example, for a system with 6 threads running in parallel executing a reduction loop with two subscript arrays. Iteration i belongs to the set $(2, 1)$, while j is in the set $(4, 1)$. Hence arrays f1() and f2() write in blocks 2 or 3 of the reduction array in iteration i, while write in blocks 4 or 5 in iteration j. As a consequence, both iterations can be safely executed in parallel by two different threads.

The classification of iterations into those sets really represents a reordering of their execution. Fig. 3 (b) shows a data structure for storing the above sets of reduction iterations, that represents an implementation of the LIP structure. Two T-by-T upper triangular matrices, init() and count(), and one linear array, next(), implement a linked-list like data structure. The entry init(i,j) contains the first iteration in the set ($B_{min} = i, \Delta B = j$), while count($i$,$j$) contains the total number of iterations in the same set. Array next(), whose size is equals to that of the subscript array/s, that is, fDim, contains links to the rest of iterations of the set, starting from next(init(i,j)).

Fig. 4 (a) shows the parallel version of the reduction loop of Fig. 1 (b) using the DWA-LIP approach taken into account the sets $(B_{min}, \Delta B)$ of reduction iterations. The procedure is as follows. All sets $(B_{min}, 0)$, referenced by the entries in the first column of init(), are executed in parallel. Afterwards, all sets referenced by the even entries in the second column, which are all data flow independent, are executed in parallel, followed by the sets referenced by the odd entries. In the third column, the iterations corresponding to the entries with ΔB modulus 3 equal to zero can be executed in parallel, followed by those corresponding to entries with ΔB modulus 3 equal to one, and finally entries with ΔB modulus 3 equal to two. This scheme is continued in the fourth and following columns. Note that a major distinction with previous work is the fact that in our scheme iterations writing in many blocks are handled without replicating computations (see for instance [8]). Part (b) of this figure shows a simple code to compute the LIP linked list structure, that may be efficiently parallelized using array expansion.

In the case of a single subscript array (see Fig. 1 (a)), all entries in both matrices init() and count() are null except for the first column. Thus the LIP structure can be simplified by replacing the two triangular matrices by linear arrays, as shown in Fig. 2 (b). These arrays are dimensioned by the number of threads participating in the computation, T. Now, each thread has an associated entry in both arrays, init() and count(). The entry in init() points to the first entry in next() belonged to that thread. The index of this entry in next() corresponds to the first iteration in the reduction loop that writes in a reduction array entry assigned to that thread. That entry in next() contains the index the next entry that also belongs to the same thread. The process is repeated the number of times stored in the associated entry

```
      real A(ADim)
      integer f1(fDim), f2(fDim), f3(fDim), ...
      integer init(NumThreads,NumThreads)
      integer count(NumThreads,NumThreads)
      integer next(fDim)

      do dB = 0, NumThreads-1
        do s = 1, dB
c$omp parallel do
          do B = s, NumThreads-dB, dB+1
            i = init(B,dB)
            cnt = count(B,dB)
            do k = 1, cnt
              r = function(i,f1(i),f2(i),f3(i),...)
              A(f1(i)) = A(f1(i)) + r
              A(f2(i)) = A(f2(i)) + r
              A(f3(i)) = A(f3(i)) + r
              ...
              i = next(i)
            end do
          end do
c$omp end parallel
        end do
      end do
```

(a)

```
integer prev(NumThreads,NumThreads)

BlockSz = floor(ADim/NumThreads) + 1
do dB = 0, NumThreads-1
  do B = 1, NumThreads
    count(B,dB) = 0
  end do
end do
do i = 1, fDim
  block1 = (f1(i)-1)/BlockSz + 1
  block2 = (f2(i)-1)/BlockSz + 1
  block3 = (f3(i)-1)/BlockSz + 1
  ...
  Bmin = min(block1,block2,block3,...)
  Bmax = max(block1,block2,block3,...)
  dB = Bmax - Bmin
  if ( count(Bmin,dB) .eq. 0 ) then
    init(Bmin,dB) = i
  else
    next(prev(Bmin,dB)) = i
  end if
  prev(Bmin,dB) = i
  count(Bmin,dB) = count(Bmin,dB) + 1
end do
```

(b)

Figure 4: ***Parallel single loop with multiple histogram reductions using data write affinity on the reduction array with loop-index prefetching (DWA-LIP) (a), and the computation of the prefetching (LIP) data structures (b)***

in count().

Fig. 5 (a) presents the parallelization of the single reduction using the DWA-LIP method, that results from the the code shown in Fig. 4 (a) by using the simplified implementation of the LIP structure. In part (b) of the figure a simple code that implements the LIP data structure is also presented, where a small auxiliary array, prev(), is needed. Note that the second loop in this code contains a histogram reduction on the count() array. Hence it can be easily parallelized using some of the previously discussed techniques. As the reduction array has a size given by the number of threads computing the code, array expansion may be chosen without a significant memory overhead.

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

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

(a)

```
integer prev(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)

Figure 5: ***A single loop with only one histogram reduction parallelized using the DWA-LIP method (a), and the computation of the prefetching arrays (b)***

In a large class of scientific/engineering problems, the subscript arrays are static, that is, they are not modified during the whole execution of the program. In such codes, the LIP structure is computed only once (in some preprocessing stage of the program, for instance), 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 those arrays. The LIP data structure has to be recalculated periodically, at least, partially. However, it is usual that the dynamic nature of the problem changes slowly. This way, the overhead of recalculating the prefetching structure is partially compensated for its reuse in a number of executions of the reduction loop (this saving technique is frequently used in, for instance, molecular dynamics simulations).

2.2 Multiple Nested Loop Reductions

In many real codes irregular reductions are computed inside a multiple nested loop. Fig. 6 presents the case of a two-nested multiple reduction loop. Pieces of code like this appear, for instance, in molecular dynamics simulations, where the outer loop runs over the particles in the domain and, for each one, the inner loop runs over its neighboring particles.

This case can be reduced to a non-nested multiple reduction loop by fusing, in some way, the nested loops. The difficulty of using this transformation, however, comes from

```
real A(ADim)
integer f1(fDim), f2(fDim), f3(fDim), ...

do i = 1, OutB
   do j = lb(i), ub(i)
      r = function(i,j,f1(),f2(),f3(),...)
      A(f1()) = A(f1()) + r
      A(f2()) = A(f2()) + r
      A(f3()) = A(f3()) + r
      ...
   end do
end do
```

Figure 6: ***A two-nested multiple irregular histogram reduction loop***

the fact that the bounds of the inner loop may depend on the outer loop (see Fig. 6). A possible way of reducing the nested loop to a single one comes from transforming it into a while loop, by using a technique derived from *loop flattening* [9], presented in the context of parallelization on SIMD architectures.

A counter is associated to the nested reduction loop and used to number its iterations, replacing the nested loop by an equivalent while loop. At the beginning of each iteration, the value of the counter is used to compute the loop indices. This counter is incremented when the execution of each iteration is ended. Now the nested loop has been transformed into a non-nested one, and thus a similar implementation of the DWA-LIP method presented in the previous section may be applied.

The parallel implementation of loop in Fig. 6 is illustrated in Fig. 7. Variable t represents the counter of the equivalent flattened while loop. InnerSz() is an auxiliary array whose entry i stores a pointer to the first entry in array next() corresponding to the outer iteration $i+1$. These LIP arrays are computed as shown in Fig. 8.

3. ANALYSIS AND PERFORMANCE EVALUATION

The data write affinity with loop-index prefetching is an efficient technique to extract and exploit parallelism from irregular histogram reductions. Consider the single-loop multiple irregular reduction presented in Fig. 1 (b) as a general case. In case of nested loops, the bound fDim would be replaced by the total number of iterations of the nested reduction loop. Considering a total of T threads cooperating in the parallel execution of the reduction loop, the parallel execution time may be written as

$$T_{par} = T_{iter}(\frac{Nit_0}{T} + \frac{Nit_1}{\lceil T/2 \rceil} + \frac{Nit_2}{\lceil T/3 \rceil} + ...), \qquad (1)$$

where T_{iter} is the (average) execution time of an iteration and Nit_k is the total number of iterations considering all the sets with $\Delta B = k$ (that is, the total sum of all entries in column k of matrix count() in the LIP structure. This simplified expression assumes that the number of iterations is uniformly distributed along each column of array init(). Similarly, the sequential execution time of the same loop is

$$T_{seq} = T_{iter}(Nit_0 + Nit_1 + Nit_2 + ...). \qquad (2)$$

```
      real A(ADim)
      integer f1(fDim), f2(fDim), f3(fDim), ...
      integer init(NumThreads,NumThreads)
      integer count(NumThreads,NumThreads)
      integer next(fDim), InnerSz(OutB)

      do dB = 0, NumThreads-1
         do s = 1, dB
c$omp parallel do
            do B = s, NumThreads-dB, dB+1
               t = init(B,dB)
               cnt = count(B,dB)
               i = 1
               while (InnerSz(i) .lt. t)
                  i = i + 1
               end do
               jstrt = lb(i) - 1
               if (i .gt. 1) then
                  jstrt = jstrt - InnerSz(i-1)
               end do
               j = t + jstrt
               do k = 1, cnt
                  r = function(i,j,f1(),f2(),f3(),...)
                  A(f1()) = A(f1()) + r
                  A(f2()) = A(f2()) + r
                  A(f3()) = A(f3()) + r
                  ...
                  t = next(t)
                  j = t + jstrt
                  if (j .gt. (up(i+1)-1)) then
                     while (InnerSz(i) .lt. t)
                        i = i + 1
                     end do
                     jstrt = lb(i) - 1 - InnerSz(i-1)
                     j = t + jstrt
                  end if
               end do
            end do
c$omp end parallel
         end do
      end do
```

Figure 7: ***Parallel version of the nested loop shown in Fig. 6 using the DWA-LIP technique***

Real applications uses to exhibit locality properties in the input domain, which is showed in the fact that values Nit_k rapidly decreases as k increases. For such applications $T_{par} \approx T_{iter}\frac{Nit_0}{T}$ and thus speed up of the parallel code is near optimum (unless there is a significant non-uniform distribution of iterations writing in the different blocks of the reduction array). A similar conclusion is drawn when only a single irregular reduction appear in the application code, as $Nit_k = 0$ for $k = 1, 2, ...$ It is not very common to find problems where most iterations are classified with high values of ΔB. For example, this would represent, in a molecular dynamics field, that a particle would interact only with distant particles (ignoring the nearer ones).

Regarding memory scalability, as no private buffering is needed (as in array expansion), the memory overhead is relatively small and not linearly depending on the number of threads. In general, the extra memory needed to store the LIP data structure has a complexity of $\mathcal{O}(fDim + 2T^2)$, where $fDim$ is the total number of iterations in the reduction loop ($fDim = \sum_{k=0}^{T} Nit_k$). Frequently, $\mathcal{O}(fDim + 2T^2) \approx \mathcal{O}(fDim)$. Comparatively, the memory overhead complexity for array expansion is $\mathcal{O}(ADim * T)$, which increases linearly with the number of threads.

We have also experimentally evaluated the DWA-LIP parallelization technique on three different benchmark codes, corresponding to non-nested and nested, static and dynamic

```
integer prev(NumThreads,NumThreads)

BlockSz = floor(ADim/NumThreads) + 1
do dB = 0, NumThreads-1
   do B = 1, NumThreads
      count(B,dB) = 0
   end do
end do
InnerSz(1) = ub(1) - lb(1) + 1
do i = 2, OutB
   InnerSz(i) = InnerSz(i-1) + up(i) - lb(i) + 1
end do
t = 1
do i = 1, OutB
   jstrt = lb(i)
   do j = lb(i), up(i)
      t = t + 1
      block1 = (f1()-1)/BlockSz + 1
      block2 = (f2()-1)/BlockSz + 1
      block3 = (f3()-1)/BlockSz + 1
      ...
      Bmin = min(block1,block2,block3,...)
      Bmax = max(block1,block2,block3,...)
      dB = Bmax - Bmin
      if ( count(Bmin,dB) .eq. 0 ) then
         init(Bmin,dB) = t
      else
         next(prev(Bmin,dB)) = t
      end if
      prev(Bmin,dB) = t
      count(Bmin,dB) = count(Bmin,dB) + 1
   end do
end do
```

Figure 8: ***Computation of the prefetching arrays used in nested multiple reduction loops***

multiple reductions. The first code is EULER, from the HPF-2 motivating applications suite [5], which solves the differential Euler equations on an irregular mesh. We have selected one of the loops computing an irregular reduction inside a time-step loop (see Fig. 9). The basic operation on this code is the computation of physical magnitudes (such as forces) corresponding to the nodes described by a mesh. The magnitudes are computed over the mesh edges, each one defined by two nodes. Therefore two subscript arrays are needed to compute the magnitudes of each edge [1, 8]. This reduction loop is interesting from the parallelization point of view because it contains subscripted reads and writes. In order to avoid side effects different from the irregular reductions, all experiments presented in this section only consider one of the reduction loops included in the EULER code. Specifically, the loop shown in Fig. 9, which corresponds to a single static loop with reductions using two subscript arrays.

From the HPF-2 motivating applications suite we have extracted also the code NBFC [1, 3], which carries out a molecular dynamics simulation. Specifically, NBFC computes electrostatic interactions, that is, non-bonded forces, between particles. In order to reduce the floating-point count, a precomputed list of neighbor particles is used, which causes the existence of an irregular reduction when accumulating force contributions, as shown in Fig. 10. This piece of code corresponds to a two-nested loop with three irregular reductions through only one subscript array, the neighbor list of particles. However, the code is static, as the neighbor list is precomputed before entering the time step loop and never updated.

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

do i = 1, numEdges
   n1 = edge(1,i)
   n2 = edge(2,i)
   a1 = funct(edgeData(1,i), edgeData(2,i), edgeData(3,i),
         velocity(1,n1), velocity(2,n1), velocity(3,n1))
   a2 = funct(edgeData(1,i), edgeData(2,i), edgeData(3,i),
         velocity(1,n2), velocity(2,n2), velocity(3,n2))
   r1 = a1*velocity(1,n1) + a2*velocity(1,n2) + edgeData(1,i)
   r2 = a1*velocity(2,n1) + a2*velocity(2,n2) + edgeData(2,i)
   r3 = a1*velocity(3,n1) + a2*velocity(3,n2) + edgeData(3,i)
   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
```

Figure 9: ***A static single loop with two subscript arrays and one reduction array from the EULER code***

```
real x(numPart), y(numPart), z(numPart)
real dx(numPart), dy(numPart), dz(numPart)
integer neigh(numInteract), numNeigh(numInteract)

do t = 1, numTimeSteps
   do i = 1, numPart
      do j = numNeigh(i), numNeigh(i+1)-1
         r1 = distance(x(i), x(neigh(j)))
         r2 = distance(y(i), y(neigh(j)))
         r3 = distance(z(i), x(neigh(j)))
         dx(neigh(j)) = dx(neigh(j)) - r1
         dy(neigh(j)) = dy(neigh(j)) - r2
         dz(neigh(j)) = dz(neigh(j)) - r3
         dx(i) = dx(i) + r1
         dy(i) = dy(i) + r2
         dz(i) = dz(i) + r3
      end do
   end do
   do i = 1, numPart
      x(i) = x(i) + dx(i)*K
      y(i) = y(i) + dy(i)*K
      z(i) = z(i) + dz(i)*K
   end do
end do
```

Figure 10: ***A static two-nested loop with one subscript array and three reduction arrays from the NBFC code***

The last code we consider in this section is a simple 2D short-range molecular dynamics simulation [14, 20] (MD). This application simulates an ensemble of particles subject to a Lennard-Jones short-range potential. To integrate the equations of motion of the particles, a finite-difference leapfrog algorithm on a Nosé-Hoover thermostat dynamics is used. The core of the code, force contributions calculation, is sketched in Fig. 11. To speed up such calculations an array of pairs of interactive particles is built, neigh(), every THop time steps. During this strip of time iterations, the neighbor list is reused, introducing a write indirection during force computation. At the end of the strip, the neighbor list is updated with the help of a link-cell. Hence, this piece of code represents a single loop with a single subscript array inside. The interesting point here is that the reduction loop is dynamic, as the subscript array is updated periodically.

The experiments have been conducted on a SGI Origin2000

```
real rx(numPart), ry(numPart)
real vx(numPart), vy(numPart)
real fx(numPart), fy(numPart)
integer neigh(2*numInteract)
do ts = TimeStep, TimeStep+THop−1
   do k = 2, NeighListLength, 2
      i = neigh(k−1)
      j = neigh(k)
      r = distance(rx(i),rx(j),ry(i),ry(j))
      if (r  .lt. cutoff) then
         ff = force(r)
         fx(i) = fx(i) + ff
         fy(i) = fy(i) + ff
         fx(j) = fx(j) − ff
         fy(j) = fy(j) − ff
      end if
   end do
   do i = 1, numPart
      vx(i) = K1 * vx(i) + K2 * fx(i)
      vy(i) = K1 * vy(i) + K2 * fy(i)
      rx(i) = rx(i) + TS * vx(i)
      ry(i) = ry(i) + TS * vy(i)
   end do
end do
```

Figure 11: ***A dynamic single loop with one subscript array and two reduction arrays from the MD code***

multiprocessor, with 32 250MHz R10000 processors, a main memory of 8192 Mbytes, and a second-level cache of 4Mbytes for each processor. The OpenMP [15] shared memory directives have been used to carry out the parallelization of the loops [18, 19]. The array expansion based parallel code used for comparison purposes was obtained using the Polaris compiler. All parallel codes (the DWA-LIP based loops and the Polaris output) were compiled using the SGI MIPSpro Fortran77 compiler (with optimization level `O2`). The maximum optimization level (`O3`) were not applied because the MIPSpro compiler does not optimize at all the parallel codes when such level is specified.

In the case of the kernels extracted from the EULER and NBFC codes, the parallelization performance has been tested on two irregular meshes of different sizes, one with 891 Knodes and the other with 1161 Knodes. In the EULER code, 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 were placed consecutively in the indirection array. In this case, a low locality in accesses to the reduction array would be expected. In the second version, the list of edges has been sorted, and therefore a higher locality would be found in the accesses to the reduction array. For the NBFC code, the nodes of the input mesh were considered as particles, while the edges were the neighboring interactions between them. Both meshes have a connectivity ($numEdges/numNodes$) of 8. The input data to the MD code was generated in a different way. Two sets of pairs positions–velocities, of size 40K and 640K, were generated representing a uniform realistic ensemble of simple particles in a liquid state.

We show in Fig. 12 the experimental performance of the parallel EULER benchmark code for the colored and sorted versions of the input mesh of size 1161 Knodes. Part (a) of the figure shows the execution time (5 iterations of the time-step loop) of both methods, the array expansion and the proposed DWA-LIP. These times exclude the calculation of the LIP data structure, as this is done only once before en-

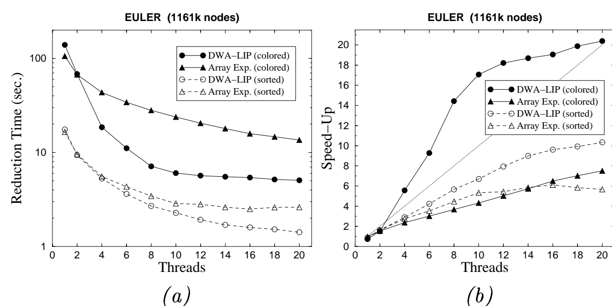

Figure 12: ***Parallel execution times (a) and speedups (b) for the parallel EULER code using the DWA-LIP and array expansion methods***

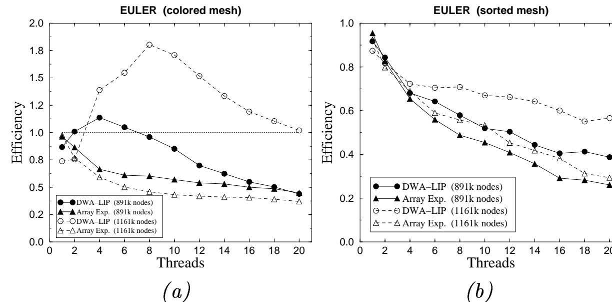

Figure 13: ***Parallel efficiencies for the colored (a) and sorted (b) meshes using DWA-LIP and array expansion***

tering into the reduction loop (static case). Part (b) shows speedups with respect to the sequential code, which was also compiled with optimization level `O2` (sequential time is 103.5 sec. and 15.3 sec. for the colored and sorted meshes, respectively, for the same 5 iterations of the time-step loop). The DWA-LIP method obtains a significant performance improvement because it exploits efficiently locality when writing in the reduction array. This fact reports a superlinear and sustained speedup.

Fig. 13 shows the parallel efficiencies for the colored (a) and sorted (b) meshes using the DWA-LIP and array expansion methods. For each class, we present results for two different mesh sizes. The sequential times for the small colored and sorted meshes of size 891 Knodes are 51.8 sec. and 11.8 sec., respectively. While our DWA-LIP method have good scalability for both orderings of the mesh, the array expansion technique exhibits an anomaly for the colored mesh. Due to the memory overhead of the expanded arrays as well as the low locality presented in the mesh, the efficiency decreases as the mesh size grows.

The sequential time costs of computing the LIP data structure 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. Fig. 14 shows parallel times (a) and speedup (b) of LIP data structure computation for both mesh sizes. In case of static meshes, the whole program suffers this computing overhead only once, at the beginning of the execution. In the

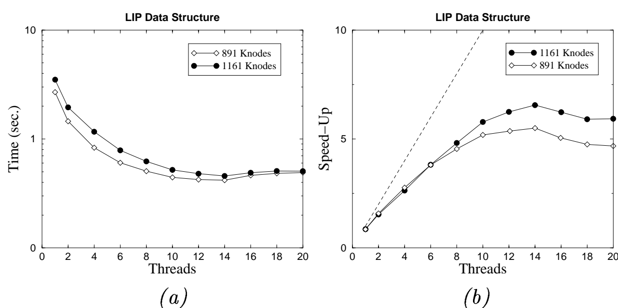

Figure 14: ***Parallel execution times (a) and speedups (b) for the loop-index prefetching***

dynamic case, the prefetching must be recomputed periodically. In many realistic situations, however, this updating process is not frequent. For instance, in the code MD is usual to recompute the neighbor list every 10 time steps or so. Then, the prefetching cost is a small fraction of the total time consumed in the reduction iterations executed between two consecutive updatings.

In the EULER code, array expansion has a significant memory overhead due to the replication of the reduction array in all the processors ($\mathcal{O}(Q * numNodes * T)$, where Q is the number of reduction arrays). The DWA-LIP method also has memory overhead due to the prefetching array ($\mathcal{O}(numEdges + T^2) \approx \mathcal{O}(numEdges)$). In the EULER reduction loop, $Q = 3$ and $numEdges \approx 8{*}numNodes$. Hence the memory overhead of array expansion is larger than that of the DWA-LIP method when the number of threads is greater than 3. In the EULER code, there are reduction loops with $Q = 5$, where the situation is even worse for the array expansion.

The performance obtained in the case of the dynamic irregular reduction code MD is shown in Fig. 15. The experiments were conducted with two different input particle sets, one with 40K particles and the other with 640K particles. In addition, some input parameters of the simulations were also changed. For 40K particles, the size of the link-cell was set to 13.0 units, while the cutoff distance was set to 5.5 units. With these numbers, the neighbor list was really big, having each particle a total of around 169 neighbors (an uniform particle distribution was considered). On the other hand, for 640K particles, the above parameters were set to 2.9 and 2.5 units, respectively. Now the connectivity is much lower, giving a total of about 8 neighbors per each particle. In any case, the size of the simulation box was 250.0×250.0 units, the number of time steps executed was 40 and the neighbor list were updated every 10 time steps. Speedups were measured with respect to the sequential version of the code (sequential time was 31 sec. and 49 sec. for the small and big problems, respectively). Note that the DWA-LIP performs much better than array expansion when the size of the problem is big enough. This is an important property, taking into account the better scalability behavior of DWA-LIP. The sequential execution times for the calculation of the LIP data structure were 11 sec. (40K particles) and 8.5 sec. (640K particles), being the former higher because it has a bigger connectivity (this overhead is not included in the

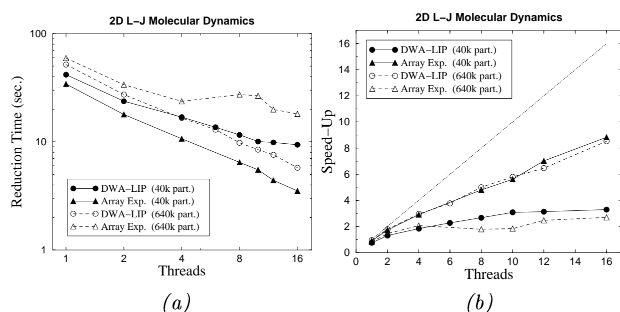

Figure 15: ***Parallel execution times (a) and speedups (b) for the parallel MD code using the DWA-LIP and array expansion methods***

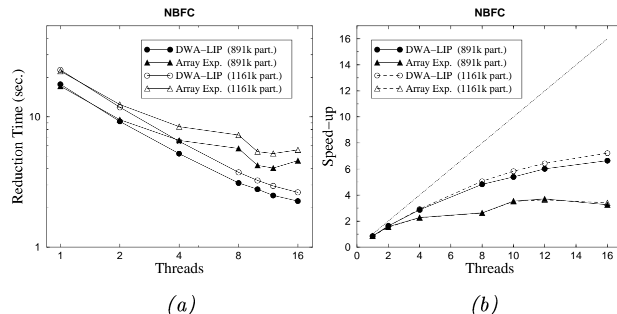

Figure 16: ***Parallel execution times (a) and speedups (b) for the parallel NBFC code using the DWA-LIP and array expansion methods***

speedup plots).

Similar performance data is illustrated in Fig. 16 for the third benchmark code NBFC. The time step loop was execute a total of 5 iterations using the same input data as in the EULER code, but considering the mesh as a set of interacting particles. However, this code is static and thus the neighbor list is built only once and never updated. Sequential execution times were also obtained to calculate speedups, 15 sec. for the small data set, and 19 sec. for the large one. The results, in this case, are better than with MD, due to the static nature of the NBFC code. In this case, computing LIP took 1.8 sec. (891K) and 2.3 sec. (1161K).

In general, array expansion has less memory overhead and, sometimes, performs better than DWA-LIP only in very small multiprocessors, and with small problem domains. However, the presented experimental results show than the DWA-LIP method has significant better performance and scalability than array expansion for many realistic situations. The good parallel behavior of DWA-LIP is justified by the fact that, in general, realistic problem domains exhibit short-range relations between data points. This can be showed with the help of tables 1 and 2. It can be seen that, for the analyzed input data sets, most of the reduction iterations modify only one of the blocks of the reduction array. And, in the case of modifying two blocks, for instance, both blocks are adjacent most of the time. This property also indicates that this kind of irregular reduction exhibit a

Threads \ ΔB	891k nodes 0	891k nodes 1	1161k nodes 0	1161k nodes 1
1	100%		100%	
2	99.5%	0.5%	99.7%	0.3%
4	98.6%	1.4%	99.1%	0.9%
8	96.7%	3.3%	98.0%	2.0%
16	93.1%	6.9%	95.8%	4.2%

Table 1: ***Percentage of the total number of reduction iterations for different values of ΔB (EULER code)***

Threads \ ΔB	40k particles 0	1	2	Remainder
1	100%			
2	95%	5%		
4	91%	7%	2%	
8	81%	16%	1%	2%
16	73%	14%	10%	3%

Threads \ ΔB	640k particles 0	1	2	Remainder
1	100%			
2	99%	1%		
4	99.5%	0.25%	0.25%	
8	99.2%	0.26%	0.26%	0.28%
16	98.9%	0.27%	0.25%	0.58%

Table 2: ***Percentage of the total number of reduction iterations for different values of ΔB (MD code)***

lot of exploitable parallelism.

4. RELATED WORK

State-of-the-art commercial compilers do not recognize irregular reductions, although, in some cases, some facilities to parallelize them are available to the programmer. For instance, SGI Fortran77 compiler [19] allows to parallelize a loop using data affinity. The processor traverses all iterations and check whether the data referenced in the current iteration belongs to it. The processor executes the calculation if the data belongs to it; otherwise it skips this iteration. A similar approach may also be exploited into HPF [10] (for example, using ON HOME) or OpenMP [15]. 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.

Recently, Han and Tseng [8] have introduced LocalWrite, as a data affinity based compiler and runtime parallelization technique for irregular reductions based on the owner compute rule. Optimizing the inspector and the executor implemented in CHAOS for irregular reductions, they present results for a shared-memory (four-processor DEC Alpha multiprocessor) and a distributed-memory (eight-processor IBM SP-2 multiprocessor) machines. Their approach is compared with the general inspector-executor technique (as implemented in CHAOS for distributed-memory machines) and the replicated (or privatized) buffer technique (SUIF compiler, for shared-memory machines). Based on these results, LocalWrite is not presented as a clearly superior alternative to parallelize irregular reductions, specially for shared-memory machines. In addition, they do not show how to deal with multiple reductions in a uniform, general and efficient way (only loops with two subscript arrays are considered and solved as a especial case, with an extra overhead due that computations are replicated).

A similar idea to LocalWrite is described by Ding and Kennedy [4], named locality grouping, in order to improve cache performance for irregular sequential codes. Basically, computations are reordered in such way that iteration clusters using close memory positions are built.

Mitchell *et. al.* [13] aims at the improvement of locality in non-affine array references. A combination of tiling, remapping and inspector-executor parallelization is used. In this paper, the technique is applied when subscripted subscripts appears only in the right hand side of the loop body. In addition, it is not applicable in the presence of several different non-affine references.

In a shared-memory context, a number of techniques were proposed to parallelize loops with histogram reductions [12, 1]: critical sections, data affinity, privatized buffer, array expansion and reduction table. The simplest method is based on critical sections. The access and update of the shared variable is enclosed by a lock/unlock pair. However, this method does not take advantage of the parallelism within reductions.

Parallelizing compilers like Polaris and SUIF [2, 7] implement the replicated buffer and array expansion methods. Besides the significant memory overhead they exhibit, both methods have lower performance and scalability than the proposed DWA-LIP. An important drawback of array expansion is that the three-phase code must be repeated for each reduction loop in the program.

Lin and Padua [12] propose the reduction table method, which combines array expansion with critical sections in order to reduce the memory overhead of the former. Potentially, this method may be effective if the subscript function exhibits high spatial locality. Unfortunately, this situation is not the general case.

5. CONCLUSIONS

The data write affinity with loop-index prefetching method proposed in this work parallelizes an irregular reduction loop exploiting write locality, similar to [8]. However, the inspector-executor model is not used to calculate the local iteration schedule. In our approach, the accesses to the subscript arrays are reordered using a linked list. This fact justifies the better performance and scalability of our method, and allows to deal in a uniform way complex cases, like reductions with multiple subscript arrays and nested reduction loops.

Compared with array expansion and array privatization, our method does not need to expand or replicate the reduction array, although there is the overhead (smaller) of building

the prefetching array. Moreover the loop-index prefetching may be computed only once and reused through the whole program (static codes), or recomputed periodically (dynamic codes). However, the overhead introduced is not significant in many realistic applications, as the experimental results have shown.

Finally, the implementation complexity of the proposed method is comparable to that of the array expansion approach. The parallel versions of the codes obtained using the DWA-LIP method is uniform enough to make easy its insertion on a parallelizing compiler.

6. REFERENCES

[1] R. Asenjo, E. Gutiérrez, Y. Lin, D. Padua, B. Pottengerg, and E. Zapata. On the automatic parallelization of sparse and irregular Fortran codes. Technical Report TR-1512, University for Illinois at Urbana-Champaign. Center for Supercomputing R&D, December 1996.

[2] W. Blume, R. Doallo, R. Eigemann, J. Grout, J. Hoeflinger, T. Lawrence, J. Lee, D. Padua, Y. Paek, B. Pottenger, L. Rauchwerger, and P. Tu. Parallel programming with Polaris. *IEEE Computer*, 29(12):78–82, December 1996.

[3] B. Brooks, R. Bruccoleri, B. Olafson, D. States, S. Swaminathan, and M. Karplus. Charmm: A program for macromolecular energy, minimization and dynamics calculations. *J. Computational Chemistry*, 4:217–183, 1983.

[4] C. Ding and K. Kennedy. Improving cache performance of dynaic applications with computation and data layout transformations. In *ACM Int'l. Conf. on Programming Languages Design and Implementation (PLDI'99)*, pages 229–241, Atlanta, GA, May 1999.

[5] I. Foster, R. Schreiber, and P. Havlak. HPF-2, scope of activities and motivating applications. Technical Report CRPC-TR94492, Center for Research on Parallel Computation, Rice University, November 1994.

[6] M. Hall, S. Amarasinghe, B. Murphy, S. Liao, and M. Lam. Detecting coarse-grain parallelism using an interprocedural parallelizing compiler. In *IEEE Supercomputing'95*, San Diego, CA, December 1995.

[7] M. Hall, J. Anderson, S. Amarasinghe, B. Murphy, S.-W. Liao, E. Bugnion, and M. S. Lam. Maximizing multiprocessor performance with the SUIF compiler. *IEEE Computer*, 29(12), December 1996.

[8] H. Han and C.-W. Tseng. Improving compiler and run-time support for irregular reductions. In *11th Workshop on Languages and Compilers for Parallel Computing (LCPC'98)*, Chaperl Hill, NC, August 1998.

[9] R. Hanxleden. Compiler support for machine independent parallelization of irregular problems. Technical Report CRPC-TR92301-S, Center for Research on Parallel Computation, Rice University, November 1992.

[10] High Performance Fortran Forum. *High Performance Fortran Language Specification, Version 2.0*, 1996.

[11] J. Ku. The design of an efficient and portable interface between a parallelizing compiler and its target machine. Technical report, Master's Thesis, Univ. of Illinois at Urbana-Champaign, Center for Supercomputing R&D, 1995.

[12] Y. Lin and D. Padua. On the automatic parallelization of sparse and irregular Fortran programs. In *4th Workshop on Languages, Compilers and Runtime Systems for Scalable Computers (LCR'98)*, Pittsburgh, PA, May 1998.

[13] N. Mitchell, L. Carter, and J. Ferrante. Localizing non-affine array references. In *Int'l. Conf. on Parallel Architectures and Compilation Techniques (PACT'99)*, Newport Beach, CA, October 1999.

[14] J. Morales and S. Toxvaerd. The cell-neighbour table method in molecular dynamics simulations. *Computer Physics Communications*, 71:71–76, 1992.

[15] OpenMP Architecture Review Board. *OpenMP: A Proposed Industry Standard API for Shared Memory Programming*, 1997.

[16] 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, June 1995.

[17] B. Pottenger and R. Eigenmann. Idiom recognition in the Polaris parallelizing compiler. In *9th ACM Int'l Conf. on Supercomputing*, pages 444–448, Barcelona, Spain, July 1995.

[18] Silicon Graphics, Inc. *MIPSpro Fortran77 Programmer's Guide*, 1994.

[19] Silicon Graphics, Inc. *MIPSpro Automatic Parallelization*, 1998.

[20] S. Toxvaerd. Algorithms for canonical molecular dynamics simulations. *Molecular Physics*, 72(1):159–168, 1991.