Optimization Techniques for Parallel Irregular Reductions ⋆

E. Gutiérrez O. Plata 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

Different parallelization techniques has been proposed in the literature for irregular reductions in the context of shared memory multiprocessors. They may be classified into two broad families: those based on privatization of the reduction arrays and those based on the partitioning of the reduction arrays. Methods in the first family are simple but no data locality is exploited and their memory scalability is low. On the other hand, methods in the second family are more complex as they require an inspection phase but they exploit data locality and scale up better in memory. Focusing on partitioning–based methods, although they exhibit a good performance in a wide variety of irregular codes, some specific input data patterns may exist for which the performance is lowered. In particular these kind of access patterns may reduce the exploited parallelism by the method or introduce workload unbalances. In order to mitigate these negative effects, we propose three optimizations for a specific partitioning–based method (DWA–LIP). These optimizations try to increase the exploited parallelism, balance the workload and reduce the effect of high contention degree regions in the reduction arrays. Efficient implementations of the proposed optimizations for the DWA–LIP method have been tested experimentally, and compared with other methods for parallelizing irregular reductions.

Key words: Irregular reductions, ccNUMA shared memory multiprocessors, data locality, privatization, partitioning, load balance.

⋆ This work was supported by the Ministry of Education and Culture (CICYT), Spain, through grant TIC2000-1658

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

do i = 1,fDim
   Compute ξ1, ξ2, ..., ξnInd
   do j=1,nInd
      A(fj(i))=A(fj(i)) ⊕ ξj
   enddo
enddo
```

Fig. 1. A histogram reduction loop with multiple subscripted subscripts

1 Introduction

Many scientific and engineering applications are classified as irregular. This class of applications are characterized by the use of indirections to access data in memory. As a consequence memory access patterns are unknown during compile time. It is very common to find in these applications reduction operations associated with the irregular memory accesses. A reduction operation is defined from an associative and commutative operator acting on scalar variables (scalar reduction) or array elements inside a loop (histogram reduction).

Figure 1 shows a prototype of a histogram reduction loop, containing several irregular reductions. In that loop one reduction array A() is updated through $nInd$ indirection arrays, f1(), f2(), ..., f_{nInd}(). Due to the loop-variant nature of the subscript arrays, loop-carried dependences may be present. However, these possible dependences can be solved due to associative and commutative nature of the reduction operator.

The properties of the memory access pattern in the histogram loop are completely defined by the contents of the indirection arrays. However, those contents are in general unknown at compile time because they usually correspond to the input domain of the computational problem solved by the application. From the performance view point, data locality is one of the most important of the above properties. Two types of data locality may be considered: *intra–loop* and *inter–loop*. Intra–loop refers to the locality in accessing the reduction array inside an iteration (accesses to the reduction array by all reductions for the same loop iteration). Inter–loop, on the other hand, refers to the locality in accessing the reduction array across iterations.

Due to the huge number of processor cycles that irregular applications usually consume, much attention from the compiler research community has been devoted to develop efficient parallelization methods for these applications. In the context of shared memory multiprocessors, we can find two broad families of compiler parallelization techniques for irregular reductions. One family is

based on the privatization of the reduction array, as the solution to solve the dependences in the histogram loop. An important example technique in this group is *array expansion* [8,11]. This method expands the reduction array by one dimension. The size of this extra dimension is equals to the number of threads cooperating in the parallel execution of the reduction loop. Each thread executes the full histogram loop by writing the reduction array updates on its private copy. Finally, all private copies are accumulated on the original reduction array.

The other family of reduction parallelization techniques is based on the partitioning of the reduction array. Tipically, the reduction array is block-partitioned, with all the block with the same size (although this is not mandatory). This way dependences are also solved but avoiding the privatization of the reduction array. In these methods a runtime inspector selects the set of iterations to be executed by each thread, with the aim that each thread only updates one of the partitions of the reduction array. Example methods in this family are LocalWrite [5,7] and DWA–LIP [3,4].

As an example, the DWA–LIP method will be explained. The name DWA–LIP comes from *Data Write Affinity with Loop-Index Prefetching*. Consider that the blocks of the partitioned reduction array are indexed by the natural numbers. The runtime inspector (LIP) sorts all the iterations of the reduction loop into sets characterized by the pair $(B_{min}, \Delta B)$, where B_{min} (B_{max}) is the minimum (maximum) index of all blocks touched by the iterations in that set, and ΔB is the difference $B_{max} - B_{min}$. The execution phase (DWA) of the method is organized as a synchronized sequence of non-conflicting (parallel) stages. In the first stage, all sets of iterations of the form $(B_{min}, 0)$ are executed in parallel because they are all data flow independent. The second stage is split into two sub-stages. In the first one, all sets $(B_{min}, 1)$ with an odd value of B_{min} are executed fully parallel, followed by the second sub-stage where the rest of sets are executed in parallel. For the remaining sets (B_{min}, k), $k = 2, 3, ..nThreads$, a similar execution scheduling is followed, considering $nThreads$ threads cooperating in the computation. Fig. 2 depicts the inspector data structures for an example code with two indirection arrays, and Fig. 3 shows an OpenMP implementation of the execution phase of this method.

2 Performance Properties

In the context of shared memory architectures, there are many factors that determine the overall performance of a reduction parallelization technique. All these factors may be classiffied into two sets. A first class includes those aspects related to the characteristics of the irregular memory access pattern. A second class corresponds to those aspects related to the way in which data

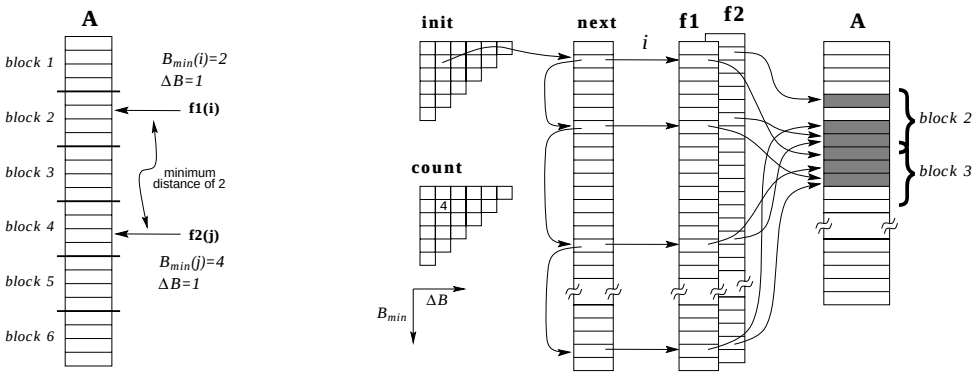

Fig. 2. DWA–LIP inspector data structures (for two indirection arrays)

```
integer f1(fDim), f2(fDim ),...,  fnInd(fDim)
real    A(ADim)
integer init (nThreads,0:nThreads−1)
integer count(nThreads,0:nThreads−1)
integer next(fDim)

do ΔB = 0, nThreads−1
   do s = 1, ΔB+1
c$omp parallel do
      do Bmin = s, nThreads−ΔB, ΔB+1
         i = init (Bmin,ΔB)
         cnt = count(Bmin,ΔB)
         do k = 1,cnt
            Compute ξ1, ξ2, ..., ξnInd
            do j=1,nInd
               A(fj(i))=A(fj(i)) ⊕ ξj
            enddo
            i = next(i)
         enddo
      enddo
c$omp end parallel
   enddo
enddo
```

Fig. 3. The execution phase of DWA–LIP in OpenMP (nThreads is the number of threads cooperating in the computation)

dependences are solved by the technique.

Performance factors that we want to highlight in the first class are: *intra–loop locality*, *inter–loop locality* and *workload balancing*. All reduction techniques schedule complete iterations among the threads. Thus, intra–loop locality is not exploited by any method. In general, this kind of locality may be improved by using an external renumbering algorithm applied to the input data domain [6,1].

Inter–loop locality is not exploited by privatization–based methods either, as they do not change the order of the loop iterations during parallel computation. However, partitioning–based methods involve a reordering of the execution of loop iterations when they are assigned to the threads. This assignment is accomplished in such a way that each thread executes most of the reduction array updates in the same block. Hence, these methods exploit inter–loop locality.

Partitioning–based techniques present a greater risk of workload unbalance than privatization–based ones. Methods based on privatization suffer from unbalance only if the reduction loop iterations have different computational costs, as in these methods all threads execute the same number of iterations. However, in the methods based in partitioning it may occur that a thread executes more iterations than other. This fact may introduce an additional source of unbalance.

Regarding the second class of performance factors described above, two of them deserve to be considered: *memory scalability* and *parallelism loss*. Privatization–based methods lacks from memory scalability, as the reduction array is fully replicated in all the threads (complete private copies). This fact represents an important problem when dealing with big data sets. In this sense, some partial solution were proposed in the literature, in order to reduce the memory pressure. Examples are the *reduction table* method [8] and the *selective privatization* [11]. The other methods, based on partitioning, do not replicate the reduction array. They only need an extra memory to store the inspector auxiliary data structures. However, in general this memory overhead is much lower than the corresponding to the privatization of the reduction array. In addition, the overhead is independent of the number of threads. Therefore, these methods behavours better in terms of memory scalability.

Methods that privatize the reduction array execute fully parallel the reduction loop. That is, they exploit optimally the loop parallelism. However, methods that partition the reduction array needs some mechanism to avoid write conflicts during updates. This mechanism may involve computation replication, as in LocalWrite or synchronization points in the parallel execution, as in DWA–LIP. In any case, the introduction of this mechanism represent certain loss of parallelism. The loss of parallelism is sensitive to the intra–loop data locality (lower intra–loop locality implies higher parallelism loss).

Fig. 4 shows a feasible space representation of the described performance factors, data (reference) locality, memory overhead, workload balance and parallelism loss, for two representative methods in both families: array expansion and DWA–LIP. In general, array expansion shows a good behaviour regarding parallelism exploitation and workload balance. However, it requires a great amount of additional memory and no data locality is taken into account. On

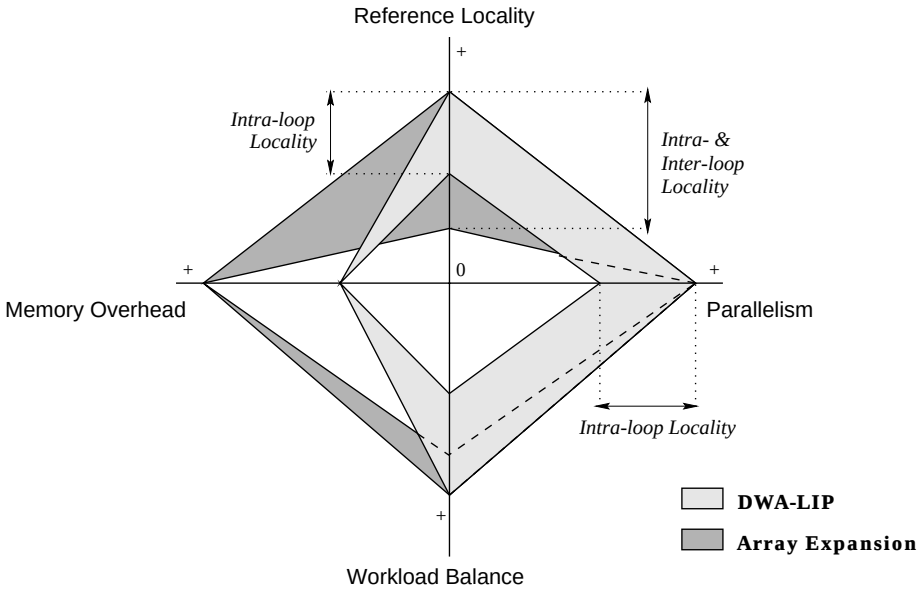

Fig. 4. Space representation of parallelism, reference locality, memory overhead and workload balance for array expansion and DWA–LIP

the other hand, DWA–LIP is much more scalable in terms of memory cost and can exploit inter–loop locality, but a low intra–loop locality of the input data set may cause significant parallelism loss and workload unbalance. In any case, the parallel execution time will be a complex function of all these performance factors, as well as some properties of the underlying parallel architecture. For instance, the overall performance on a ccNUMA machine is much more influenced by data locality factors than on a bus-based machine.

In practice, most of the times the input data sets for many irregular applications use to be well ordered and numbered, so as any of the discussed methods performs reasonably well. However, sometimes there are specific situations where the performance drops down drastically. In such situations it is crucial to optimize the behaviour of the method in hand. Next section we discuss some representative situations of this kind and the corresponding possible optimizations applied to the example case of DWA–LIP.

3 Optimizations for Partitioning–Based Methods

As it was pointed out in the previous section, parallelization methods like DWA–LIP use to have a good performance for a wide range of irregular problems. However, sometimes, certain input data domains may occur that introduce serious performance penalties in the parallel execution of the code.

We will consider three typical and frequent situations: low intra–loop locality, workload unbalance and high contention areas. As a partitioning–based paral-

lelization method usually block partitions the reduction array, a low intra–loop locality implies a high interaction among thread, as each thread will write in many different blocks. This high interaction introduces parallelism loss in the method, in term of computation replication or extra synchronization events.

In the case of DWA–LIP, this situation involves the appearance of an important number of reduction iterations in the pairs $(B_{min}, \Delta B)$, for high values of ΔB. As shown in Fig. 3, the parallel loop (B_{min}) is short for the execution of such iterations.

The second source of inefficiency is workload unbalance, that is, when reduction iterations write much more in some blocks than in others. This implies that some threads execute more iterations than others. In DWA–LIP, a workload unbalance means that reduction iterations are not equally distributed among $(B_{min}, \Delta B)$-pairs for each ΔB. In this case, workload unbalance will have a negative effect on the parallel loop (B_{min}) in Fig. 3.

An extreme case of workload unbalance, that deserves to be considered separately, is the contention problem. For some input domains it may exists a small area in the reduction array that is updated much more times than the remainder. This one is called the high contention area and commonly spans a few number of blocks. In the case of DWA–LIP, this problem implies a very high number of iterations in a few number of pairs $(B_{min}, \Delta B)$ with a high ΔB. This fact introduces workload unbalance and parallelism loss.

In the rest of this section, specific modifications of the DWA–LIP method to improve its performance for the above problems are explained and discussed.

3.1 Improving the parallelism

The solution to reduce the parallelism loss for data access patterns with a low intra–loop locality is based on increasing the length of the B_{min} parallel loop (Fig. 3). This can be achieved at the expense of certain memory overhead (although always less than in a privatization–based method). By replicating the reduction arrays by a fixed ρ number of times, less than the number of threads, we can guarantee that ρ threads can work in parallel, as they can write on different private memory spaces. This optimization technique based on the partial replication of the reduction arrays is called *partial array expansion*.

The partial array expansion allows us to decrease the number of synchronized substages for a given ΔB, that is, it can increase the number of iterations in the B_{min} parallel loop of the execution phase. By collapsing some substages for a given ΔB they can be executed in parallel, as they write now in different private copies of the reduction array. In the basic DWA–LIP method $\Delta B + 1$

```
real     A(ADim)
real     A_exp(ADim, ρ)
do ΔB = 0, nThreads−1
   do s = 1, Δ^exp
c$omp parallel do
      do B_min = s, nThreads−ΔB, Δ^exp
         i = init(B_min,ΔB)
         cnt = count(B_min,ΔB)
         do k = 1,cnt
            Compute ξ_1, ξ_2, ..., ξ_nInd
            B_priv=B_min mod ρ
            do j=1,nInd
               A_exp(f_j(i), B_priv)=A_exp(f_j(i),B_priv) ⊕ ξ_j
            enddo
             .....
            i = next(i)
         enddo
      enddo
c$omp end parallel
   enddo
enddo
```

Fig. 5. Partially expanded DWA–LIP computation phase

synchronized substages are executed for each ΔB. If we have ρ replicas of the reduction array, at least ρ threads can work in parallel and therefore the number of synchronized substages will be $\left\lfloor \frac{\Delta B}{\rho} + 1 \right\rfloor$ for each ΔB.

The scheduling of $(B_{min}, \Delta B)$-pairs can be done so as the original *DWA-LIP* computation phase remains practically unmodified. We need only to schedule iterations by assigning reduction private copies in such a way that write conflicts are avoided. The code in Fig. 5 shows this assignment. Now synchronized substages are defined based on the parameter $\Delta^{exp} = \left\lfloor \frac{\Delta B}{\rho} + 1 \right\rfloor$, whose meaning is the number of substages that can be executed in parallel for a given ΔB. As $\Delta^{exp} \leq \Delta B$ the method improves parallelism.

This execution scheme is depicted in Fig. 6, where the execution of the basic and the partially expanded DWA–LIP are compared. Note that if $\Delta B > \frac{nThreads-1}{2}$ then $\Delta^{exp} \leq \rho$ and for such a ΔB it would be more appropriate executing $(B_{min}, \Delta B)$-pairs in substages of ρ sets in order to exploit all the available parallelism. The private copies are ciclically assigned to the threads for each substage.

The effect of partial expansion on the DWA–LIP execution scheme is shown in Fig. 6 for a partial expansion index $\rho = 2$. In this case the number of private copies of the reduction array is 2, and so two threads can write concurrently in them. Consider, for example, $\Delta B = 1$. In the basic method there are two synchronized substages to avoid write conflicts in the reduction array (see

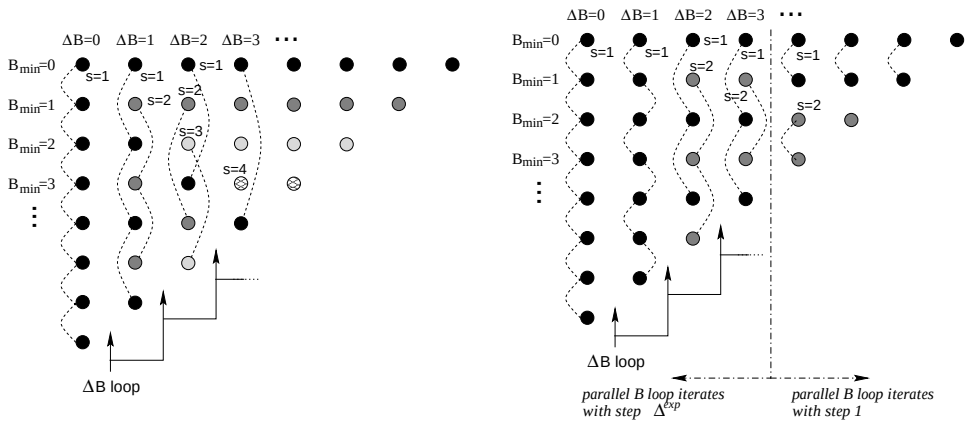

Fig. 6. Execution scheme of the basic (a), and the partially expanded DWA–LIP (b)

Fig. 6(a)). But in the partially expanded DWA–LIP this $\rho = 2$ substages can be collapsed in one, as now conflicts disappear by writing in different replicas of the reduction array (see Fig. 6(b)).

Although on increasing ρ we can expect more parallelism exploitation, however there is actually a limitation, as a maximum parallelism is obtained for $\rho = \frac{nThreads-1}{2}$. Also it must be taken into account that an initialization of private copies and a final reduction phase have to be introduced in the parallel code. Other practical issue is the ρ factor itself. It must be chosen as a trade off between the extra amount of memory needed for the reduction copies and the parallelism increment. If the intra–loop locality is not very low good results can be obtained with low values of ρ, much less than the number of threads. (observe that $\rho = 1$ is equivalent to the basic method).

3.2 Improving the workload balance

A non-uniform distribution of the workload among threads comes as a consequence of having many more reduction iterations writing in some blocks than in others. A naive solution to this problem, without losing locality properties, may be using a block partition of the reduction array but with blocks of different size. However, this solution would introduce strong modifications in the inspector, increasing significantly its computational cost. We can mostly preserve the scheme of the original inspector using a simpler approach, based on the partitioning of the reduction array into small equal-sized blocks, in a number ($nBlocks$) greater than the number of threads ($nThreads$). This way the inspector remains unchanged but working with $nBlocks$ blocks instead of $nThreads$ blocks, as originally. To simplify the treatment, $nBlocks$ is chosen as a multiple of $nThreads$.

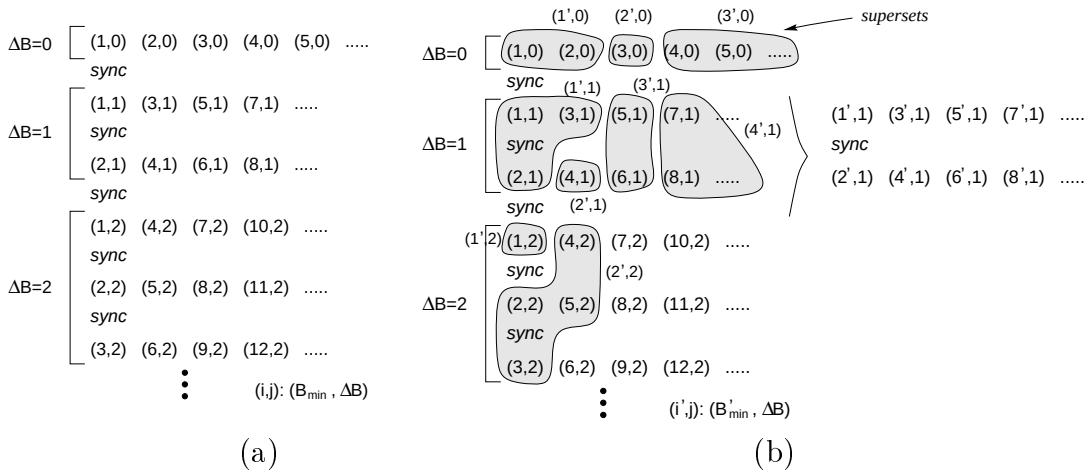

Fig. 7. Execution scheme of the basic DWA-LIP method (a), and the load balanced version (b). Iteration sets are denoted by (i, j) and balanced superset by (i', j).

```
do ΔB = 0, nBlocks−1
   do s = 1, Δ^LB + 1
c$omp parallel do
      do B_min = s, nThreads, Δ^LB_ΔB + 1
         Execute Superset (B'_min, ΔB)
      enddo
c$omp end parallel
   enddo
enddo
```

Fig. 8. Generic load balanced DWA–LIP computation phase

In order to minimize changes in the execution phase of the original DWA–LIP method, after building the $nBlocks$-block based $(B_{min}, \Delta B)$-pairs, we will group sets for each ΔB in a balanced way. We will group always contiguous sets and therefore the resulting superset of balanced grouping writes in a contiguous area of the reduction array. The balanced grouping of $(B_{min}, \Delta B)$-pairs will be called balanced superset and we will refer to the i-th balanced superset for a ΔB with the pair $(i', \Delta B)$. The balance grouping can be achieved by a simple histogram analysis of the $(B_{min}, \Delta B)$-pairs for each ΔB. Note that now ΔB will have values up to $nBlocks - 1$.

Fig. 7 shows how supersets are built. Fig. 7(a) depicts the execution scheme of the original DWA–LIP. For each ΔB, synchronized stages of $(B_{min}, \Delta B)$-pairs are executed. Fig. 7(b) sketches the building of $(i', \Delta B)$-supersets. For a given ΔB, adjacent $(B_{min}, \Delta B)$-pairs are aggregated making a $((i', \Delta B)$-superset. This aggregation is made in such a way that $((i', \Delta B)$-supersets contains similar number of iterations for each ΔB (condition for workload balance).

A pseudocode of the load balanced DWA–LIP execution phase is shown in

Fig. 8. The synchronized stages are executed in a similar way to the original version but now the parallel loop runs over the balanced $(i', \Delta B)$-supersets. As supersets are built as an aggregation of $(B_{min}, \Delta B)$-pairs writing in contiguous blocks, we can state the data independence of supersets in similar terms than in the original (non optimized) method. If we guarantee that each $(i', \Delta B)$-superset groups at least r $(B_{min}, \Delta B)$-pairs, two supersets $(i', \Delta B)$, $(j', \Delta B)$ will be free of write conflicts if $i' * r + \Delta B < j' * r$. The reason is that if each superset contains at least r sets, the i'-th superset could write at most in the $i' * r + \Delta B$-th block. Based on this property, supersets can be executed in parallel in synchronized stages of the form $(i', \Delta B)$, $(i' + \Delta^{LB}, \Delta B)$, $(i' + 2\Delta^{LB}, \Delta B)$, and so on, as it is displayed in Fig. 8, being the parameter $\Delta^{LB} = \left\lfloor \frac{\Delta B - 1}{r} + 1 \right\rfloor$. It is possible to prove that the optimal value for r is $\min\left(\Delta B, \frac{nBlocks}{nThreads}\right)$, that corresponds to $\Delta^{LB} = \left\lceil \Delta B \frac{nThreads}{nBlocks} \right\rceil$. Note that the execution phase is similar to the original one but using the new parameter Δ^{LB} instead of $\Delta B + 1$.

3.3 Improving the contention

For some input data domains, the reduction array may exhibit small areas with a large number of updates, many more than in the rest of the array. These are the high contention areas. This situation can be considered as an extreme case of workload unbalance, and then solved as discussed in the previous section. However, the high contention problem has associated a parallelism loss effect, due to the existence of many iterations with high ΔB (this is a difference, regarding the pure workload unbalance problem, discussed in the previous section). Hence, we need to devise a new solution in order to take into account both issues.

To mitigate the negative effect of high contention areas we can replicate on all threads those blocks of the reduction array that have access frequency peaks. These blocks correspond to the high contention areas. This way writes in these areas can be done in parallel with no conflicts. As contention blocks are the only replicated, the memory overhead is much lower than for a tipical privatization–based method, like array expansion. We call this technique local expansion.

The execution phase for a locally expanded DWA–LIP method is exactly the same than the original method, as the effect of the optimization is making iterations to migrate from $(B_{min}, \Delta B)$-pairs with high ΔB to others with a lower value of ΔB.

The high contention areas can be easily detected by the inspector from the information in the *count* matrix (see Fig. 2). Using a histogram analysis, blocks

with high contention can be easily selected. Once these blocks are replicated, iterations in $(B_{min}, \Delta B)$-pairs writing in them migrate to $(B_{min}, \Delta B)$-pairs with low ΔB, as write conflicts in that area disappear. The number of blocks to be partially expanded will be a trade off between the parallelism increment and the memory overhead.

4 Experimental evaluation

In previous works [3,4] we have shown that the DWA–LIP method performs well for typical irregular codes on ccNUMA share memory machines, specially for large size data sets. In all these tested cases, data access patterns exhibit a relatively high intra–loop locality and thus, as partitioning–based methods, like DWA–LIP, are able to exploit inter–loop locality, they provide, in general, better results than privatization–based methods. In addition DWA–LIP has lower extra memory requirements.

In this paper, however, we will consider special cases of irregular codes and data sets that result in a low performance using methods like DWA–LIP. This way, we can test and compare the optimization techniques proposed in the previous sections. All the experiments have been conducted on a SGI Origin2000 multiprocessor, with 250-MHz R10000 processors (4 MB L2 cache) and 12 GB main memory, using IRIX 6.5. Irregular codes are written in Fortran77 and parallelized by using OpenMP directives.

4.1 Improving the parallelism

EULER code from the motivating application suite of HPF-2 [2] has been chosen to test the partially expanded DWA–LIP. This code solve Euler differential equations on an irregular grid and it includes loops with two subscripted reductions. The reduction loops are placed inside an outer time-step loop. The input data set is the description of an irregular mesh describing the problem domain and it is read once at the beginning of the program. We have taken a mesh of 1161K nodes with a connectivity of 8 (ratio between edges and nodes). Although the original mesh have high intra–loop locality we have generated a new mesh with a much lower intra–loop locality by renumbering the nodes. In this way we can test the impact of this issue. In addition two reordering of the mesh edges have been tested in order to evaluate the influence of the inter–loop locality. One version is obtained applying a edge-coloring algorithm (low inter–loop locality). In the other version, the list of edges has been lexicographically sorted, resulting in an expected higher inter–loop locality.

Fig. 9 shows the speedup (referenced to the sequential execution time) for the computation phase of the basic DWA-LIP, its partially expanded version and array expansion. It must be taken into account that the low inter–loop locality of the colored mesh causes a slower execution of the sequential code than for the sorted mesh. Due to the low intra–loop locality, some amount of parallelism is lost in the basic DWA-LIP method ($\rho = 1$). That gives a bad performance, lower than for array expansion. However, for more than 8 processors we obtain better execution time with the partially expanded method using $\rho = 4$. For 16 threads and $\rho = 8$, the DWA-LIP–based parallelization outperforms array expansion. With the sorted mesh case the parallelism is limited due to the low intra–loop locality of data pattern. Such as it is observed when the ρ factor is increased a higher speed-up is achieved by the partially expanded DWA–LIP method. For 16 threads and $\rho = 8$ the partially expanded DWA-LIP provides almost the same performance than array expansion, but it required just a half of memory. For both the colored and sorted version the overhead of the inspector phase is about a 5% of the reduction loop execution time.

The extra memory needed by both methods is another important overhead aspect. For the tested EULER code, and considering a parallel execution on 16 threads, the partially expanded DWA–LIP method with $\rho = \frac{nThreads}{2}$ provides a similar speedup than array expansion, as was shown before. However, assuming that $nThreads \ll N$, being N the number of iterations of the reduction loop, array expansion needs around 3 times more extra memory than the other method.

4.2 Improving the workload balance

A kernel computing Legendre transforms (Spec Code [9]), used in numerical weather prediction, has been tested using the load balancing optimization approach. The reduction loop is a multiple nested loop where the innermost loop bounds are indirection arrays. Thus load unbalance appears as some iterations have greater workload than others.

Fig. 10 shows the speedup of the Legendre parallel reduction loop for several techniques. The workload unbalance results in a suboptimal performance for the basic DWA–LIP method. Also array expansion obtains poor speedup due both to the unbalanced behavior of the loop and that only the outermost loop of the irregular reduction is parallelized. When the optimization to balance the workloas is introduced into DWA–LIP, the performance is significantly improved. Considering an uniform partition of the reduction array into $nBlocks$ blocks and a execution using $nThreads$ threads, when the ratio $nBlocks/nThreads$ increases the speedup improves slightly. However, there is no additional improvement for values beyond 8. So a good improvement

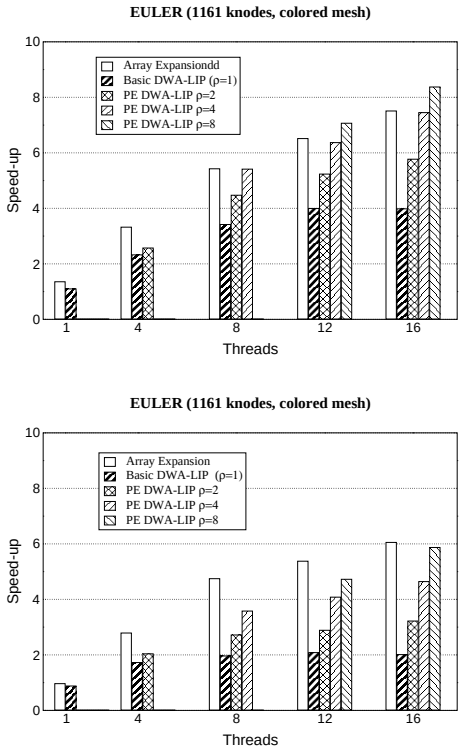

Fig. 9. Performance of the parallel reduction loop of the EULER code for DWA-LIP, partially expanded (PE) DWA–LIP and array expansion methods

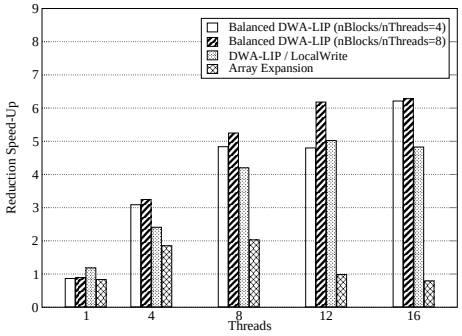

Fig. 10. Speedup for the parallel Legendre code using the load balanced DWA–LIP method and other techniques

is achieved with a tiny increment of the size of the inspector data structure. It is remarkable that the overhead of the DWA–LIP inspector phase is less than 1% of the reduction time, as the inspection phase runs only once because indirection patterns keep unchanged during the execution.

4.3 Improving the contention

A simple simulator of two-dimensional short-range molecular dynamics [10] (MD2) has been tested using DWA-LIP with local expansion. This code simulates the behavior of a set of particles inside a Lennard-Jones short-range

potential. For each particle it is computed the force contributions due to its neighbour particles that are stored in a list. This force computation involves a histogram reduction loop that contains two reduction arrays and two subscript arrays. In addition, the subscript array is dynamically updated every 10 time steps. It has been built artificially a high contention area in the particle domain that is composed by 640K particles in this experiment. To emphasize the importance of inter–loop locality, experiments were coducted using two orderings in the neighbour list: the original order and a randomized one.

The speedup for the execution phase of the locally expanded DWA–LIP compared to other techniques is shown in Fig. 11. The top part in the figure corresponds to the original code (original neighbour list) while the bottom part corresponds to the randomized neighbour list. The original data access pattern exhibits a relatively high inter–loop locality and only a small fraction of array elements are written by more than one thread in privatization–based methods. For this reason the selective privatization method performs so well, because it needs to replicate only a small number of reduction array elements. The poor behaviour of DWA–LIP is due to the bad distribution of iterations into $(B_{min}, \Delta B)$-pairs, caused by the high contention region. The local expansion of massively accessed blocks in DWA–LIP method improves significantly the performance of this method. The worse performance of array expansion is due to the final reduction operation and the low intra-loop locality in writing on expanded arrays. Nevertheless, in this case, the selective privatization technique performs best because it replicates only a few elements and almost all the reduction array updates are carried out over the original reduction array and not over the private element copies.

The randomization of the neighbour list has a strong impact in the efficiency of the tested techniques. Performance of selective privatization is drastically reduced, as the number of elements written by different threads increases significantly, and thus a big fraction of the reduction array is privatized. However, DWA–LIP and its optimized version keeps their performance at similar levels than before, as these methods exploit at runtime inter–loop locality. The inspector overhead is not included in the plots and for the two versions are about 1% for locally expanded DWA–LIP and 2.5% for selective privatization.

5 Conclusions

In the context of shared memory parallel architectures we have identified two families of parallelization methods of irregular reductions. One family is based on the privatization of the reduction array, while the other is based on the partitioning of such array.

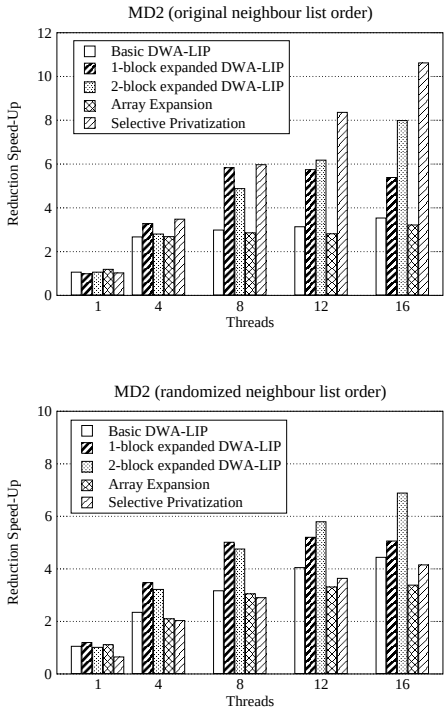

Fig. 11. Speedup for the MD2 reduction loop using DWA–LIP with local expansion and other methods.

Broad studies along the last few years have shown that methods in both families work reasonably well for many tipical irregular applications (however the low memory scalability of the privatization–based methods). However, there are not infrequent situations were the performance is very low. To overcome such situations we need to introduce optimizations to the basic methods.

In this paper we propose and discuss optimization techniques to solve problems related to low intra–loop locality, workload unbalance and high contention areas. These techniques modify and improve basic partitioning–based parallelization reduction methods. Specifically, we explain efficient implementations for the particular case of the DWA–LIP method.

Analytical and experimental results (in this paper we focused on the experimental approach) allow us to conclude that it is possible to improve the performance of a partitioning–based parallelization method, like DWA–LIP, with no significant loss of its basic properties (data locality exploitation, low memory overhead) and with no significant increase in the algorithmic complexity.

References

[1] C. Ding and K. Kennedy. Improving Cache Performance of Dynamic Applications with Computation and Data Layout Transformations. In *Proceedings of the ACM International Conference on Programming Language Design and Implementation (PLDI'99)*, pages 229–241, Atlanta, GA, May 1999.

[2] I. Foster, R. Schreiber and P. Havlak. HPF-2, Scope of Activities and Motivating Applications. *Technical Report CRPC-TR94492*, Rice University, November 1994.

[3] E. Gutiérrez, O. Plata and E.L. Zapata. An Automatic Parallelization of Irregular Reductions on Scalable Shared Memory Multiprocessors. In *Proceedings of the 5th International Euro-Par Conf. (EuroPar'99)*, Tolouse, France, pp. 422–429, August-September 1999.

[4] E. Gutiérrez, O. Plata and E.L. Zapata. A Compiler Method for the Parallel Execution of Irregular Reductions in Scalable Shared Memory Multiprocessors. In *Proceedings of the 14th ACM International Conference on Supercomputing (ICS'2000)*, Santa Fe, NM, pp. 78–87, May 2000.

[5] H. Han and C.-W. Tseng. Improving Compiler and Run-Time Support for Irregular Reductions. In *Proceedings of the 11th Workshop on Languages and Compilers for Parallel Computing (LCPC'98)*, Chapel Hill, NC, August 1998.

[6] H. Han and C.-W. Tseng. Improving Locality for Adaptive Irregular Scientific Codes. In *Proceedings of the 13th Workshop on Languages and Compilers for Parallel Computing (LCPC'00)*, Yorktown Heights, NY, August 2000.

[7] H. Han and C.-W. Tseng. A Comparison of Parallelization Techniques for Irregular Reductions. In *Proceedings of the 15th IEEE Int'l. Parallel and Distributed Processing Symposium (IPDPS'2001)*, San Francisco, CA, April 2001.

[8] Y. Lin and D. Padua. On the Automatic Parallelization of Sparse and Irregular Fortran Programs. In *Proceedings of the 4th Workshop on Languages, Compilers and Runtime Systems for Scalable Computers (LCR'98)*, Pittsburgh, PA, May 1998.

[9] N. Mukherjee and J.R. Gurd. A Comparative Analysis of Four Parallelisation Schemes. In *Proceedings of the 13th ACM International Conference on Supercomputing (ICS'99)*, Rhodes, Greece, pp. 278–285, June 1999.

[10] J. Morales and S. Toxvaerd. The Cell-Neighbour Table Method in Molecular Dynamics Simulations. *Computer Physics Communication*, 71:71–76, 1992.

[11] H. Yu and L. Rauchwerger. Adaptive Reduction Parallelization Techniques. In *Proceedings of the 14th ACM International Conference on Supercomputing (ICS'2000)*, Santa Fe, NM, pp. 66–77, May 2000.