Balanced, Locality-Based Parallel Irregular Reductions*

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

Department of Computer Architecture
University of Málaga
E-29071 Málaga, Spain
{eladio,oscar,ezapata}@ac.uma.es

Abstract. Much effort has been devoted recently to efficiently parallelize irregular reductions. Different parallelization techniques have been proposed during the last years that can be classified into two groups: LPO (*Loop Partitioning Oriented* methods) and DPO (*Data Partitioning Oriented* methods). We have analyzed both classes in terms of a set of performance aspects: data locality, memory overhead, parallelism and workload balancing. Load balancing is not an issue sufficiently analyzed in the literature in parallel reduction methods, specially those in the DPO class. In this paper we propose two techniques to introduce load balancing into a DPO method. The first technique is generic, as it can deal with any kind of load unbalancing present in the problem domain. The second technique handles a special case of load unbalancing, appearing when there are a large number of write operations on small regions of the reduction arrays. Efficient implementations of the proposed solutions to load balancing for an example DPO method are presented. Experiments on static and dynamic kernel codes were conducted making comparisons with other parallel reduction methods.

1 Introduction

Many scientific/engineering applications are based on complex data structures that introduce irregular memory access patterns. In general, automatic parallelizers obtain sub–optimal parallel codes from those applications, as traditional data dependence analysis and optimization techniques are precluded. Run–time techniques have been proposed in the literature to support the parallelization of irregular codes, like those based on the inspector-executor paradigm [15], or the speculative execution of loops in parallel [16].

Run–time techniques like mentioned above are general enough to be applied to many different classes of irregular computations. However, due in part to their generality, the efficiency of the parallelized codes is usually poor. Significantly better performance may be obtained from techniques tailor-made for specific irregular operations, computational structures and/or data access patterns [1, 11].

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

Reduction operations represent an example of such computational structures, frequently found in the core of many irregular numerical applications. The importance of these operations to the overall performance of the applications has involved much attention from compiler researchers. In fact, numerous techniques have been developed and, some of them implemented in contemporary parallelizers, to detect and transform into efficient parallel code those operations.

In this paper we classify the most important irregular reduction parallelization techniques into two main classes: *Loop Partitioning Oriented* (LPO) techniques and *Data Partitioning Oriented* (DPO) techniques. LPO methods assign blocks of reduction loop iterations to the cooperating threads, while DPO methods assign blocks of the reduction arrays to the threads (and the loop iterations that each thread execute are those that write mostly in the owned block). Further, both classes are analyzed in terms of a set of performance aspects: data locality (inter-loop and intra-loop), memory overhead, parallelism and workload balancing. These aspects have a strong influence in the overall performance and scalability of the parallel reduction code. We will discuss how the parallelization techniques usually try to optimize some of the above-mentioned aspects, missing the other(s). That is, the parallel code is usually not optimal in terms of both performance and scalability.

Load balancing is not an aspect sufficiently discussed and analyzed in the considered parallel reduction methods, specially those in the DPO class, which are very sensitive to that aspect. In this paper we propose two techniques to introduce load balancing into a DPO method. The first technique, based on the subblocking of the reduction arrays, is generic, as it can deal with any kind of load unbalancing present in the problem domain. The second technique handles a special case of load unbalancing, present when there are a large number of write operations on small regions of the reduction arrays. The proposed solution for these cases is based on the privatization of the blocks making up those regions.

In this paper we discuss efficient implementations of the proposed solutions to load balancing for the DWA–LIP DPO method, that were experimentally tested on different kernel codes and compared with other parallel reduction techniques. The main contribution from this research is that it is possible to improve the performance of a DPO method, like DWA–LIP, for unbalanced problems with no significant loss of data locality exploitation and no substantial increment in extra memory overhead and algorithmic complexity.

The remainder of the paper begins with a discussion and classification of the most important methods for irregular reduction parallelization. Using such classification, the methods are analyzed in terms of a set of relevant performance features. Next, we highlight the load balancing problem, and propose our solutions to solve it, as well as efficient implementations on a particular DPO method. Finally, experimental results that validate our analysis are presented.

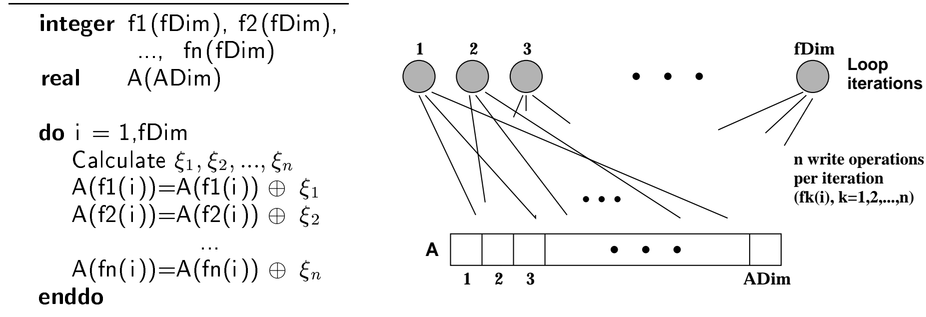

Fig. 1. A loop with multiple reductions and a schematic representation of the irregular memory access pattern

2 Methods for Reduction Parallelization

Different specific solutions to parallelize irregular reductions on shared-memory multiprocessors have been proposed in the literature. We may classify them into two broad categories: *loop partitioning oriented* techniques (LPO) and *data partitioning oriented* techniques (DPO). The LPO class includes those methods based on the partitioning of the reduction loop and further execution of the resulting iteration blocks on different parallel threads. A DPO technique, on the other hand, is based on the (usually block) partitioning of the reduction array, assigning to each parallel thread preferably those loop iterations that issue write operations on a particular data block (then it is say that the thread owns that block).

To facilitate the analysis of the above classes, we consider in the rest of the paper the general case of a loop with multiple reductions, as shown in Fig. 1 (the case of multiply nested loops is not relevant for our discussion). A() represents the reduction array (that could be multidimensional), which is updated through multiple subscript arrays, f1(), f2(), ..., fn(). Due to the loop-variant nature of the subscript arrays, loop-carried dependences may be present, and can only be detected at run–time.

Taking into account this example irregular reduction loop, Fig. 2 shows a graphical representation of generic techniques in the described two classes, LPO and DPO.

The simplest solution in the LPO class is based on critical sections, where the reduction loop is executed fully parallel by just enclosing the accesses to the reduction array in a critical section. This method exhibits a very high synchronization overhead and, consequently, a very low efficiency.

The synchronization pressure can be reduced (or even eliminated) by privatizing the reduction array, as it is done by the *replicated buffer* and *array expansion* techniques. The *replicated buffer* method replicates private copies of the full reduction array on all threads. Each thread accumulates partial results

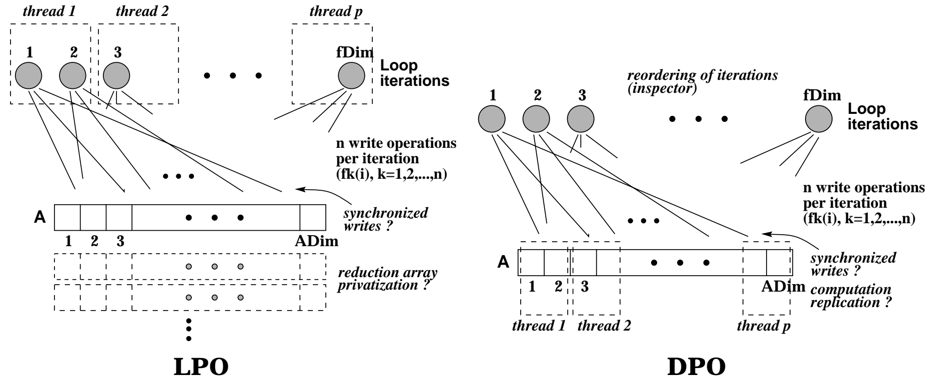

Fig. 2. General schematic representation of the LPO and DPO classes of reduction parallelization techniques

on its private copy, and finally the global result is obtained by accumulating the partial results across threads on the global reduction array (this last step needs synchronization to ensure mutual exclusion). The other method, (*array expansion*), expands the reduction array by the number of parallel threads. Now each thread accumulates partial results on its own section of the expanded array. This approach allows to obtain the final result in a similar way than the first method, but with no need of the final synchronization.

Note that these two methods transform the reduction loop into a fully parallel one, as the possible loop-carried dependencies disappear as a result of the privatization of the reduction array. However, they have scalability problems for large data sets, as the privatization affect the whole reduction array and on all threads (the memory overhead increases in proportion to the number of parallel threads).

Methods in the DPO class avoid the privatization of the reduction array, as it is partitioned and assigned to the parallel threads. In order to determine which loop iterations each thread should execute (mostly those that write in its assigned block), an inspector is introduced at runtime whose net effect is the reordering of the reduction loop iterations (through the reordering of the subscript arrays). The selected reordering tries to minimize write conflicts, and, in addition, to exploit data (reference) locality.

Two methods has been proposed in the literature in the DPO class. One method was termed LocalWrite [7, 8, 10], and is based on the *owner–computes rule*. Each thread owns a portion of the reduction array (block partitioning). The inspector has reordered the subscript arrays in such a way that, in the execution phase, the set of iterations assigned to that thread only updates array elements of the owned block. Note, however, that, in order to fulfill the computes rule, those iterations that updates more than one block of the reduction array must

	Inter-Loop Locality	*Intra-Loop Locality*	*Memory Overhead*	*Parallelism*	*Workload Balance*
LPO	variable	variable	high/medium/low	high	high
DPO	high	variable	low	high/medium/low	variable

Table 1. Performance characteristics for the LPO and DPO classes of parallel irregular reduction methods. The label `variable` means that the property is not intrinsically exploited by the method, but it depends on input data

be replicated across the owner threads. This computation replication introduces a performance penalty (parallelism loss).

An alternative method that avoids computation replication is DWA–LIP [4–6]. Consider that the blocks of the reduction array are indexed by the natural numbers. The inspector (named *loop-index prefetching* phase, or LIP) now 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 (or computation phase) 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 (optimal utilization of the threads). 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. A similar scheme is followed in the subsequent stages, until all iterations are exhausted

2.1 Performance Characteristics

Methods in the LPO and DPO classes have, in some sense, complementary performance characteristics. Methods in the first class exhibit optimal parallelism exploitation (the reduction loop is fully parallel), but no data locality is taken into account and lack scalability (memory overhead is proportional to the number of threads). However, as the reduction loop is uniformly partitioned, these methods usually exhibit balanced workload.

Methods in the second class, however, exploit data locality and exhibit usually much lower memory overhead, and it is not dependent on the number of threads (the inspector may need some extra buffering to store subscript reorderings, independently on the number of threads). However, either the method introduces some computation replication or is organized in a number of synchronized phases. In any case, this fact represents loss of parallelism. In addition, there is the risk that the number of the loop iterations that write some specific block is much different from the same in another block (workload unbalance).

Table 1 shows typical characteristics of methods in LPO and DPO classes considering four relevant performance aspects: data locality, memory overhead,

parallelism and workload balance. Data locality is in turn split into inter-loop and intra-loop localities. Inter-loop locality refers to the data locality among different reduction loop iterations. Intra-loop locality, on the other hand, corresponds to data locality inside one reduction loop iteration.

LPO methods basically exploit maximum parallelism in a very balanced way. Regarding memory overhead, they are very eager. Different solutions has been proposed recently to reduce this high memory overhead, based on the array expansion and replicated buffer basic methods. The *reduction table* method [11] assigns a private buffer to each thread of a fixed size (lower than the size of the reduction array). Then, each thread works on its private buffer indexed by using a fast hash formula. When the hash table is full, any new operation will work directly on the global reduction array within a critical section. Other method is *selective privatization* [18], where the replication include only those elements referenced by various threads. It first determine (inspector phase) which are those elements and then allocate for them private storage space. Each thread, then, works on its private buffer when updating conflicting elements, while it works on the global reduction array otherwise. This execution behavior implies a replication of each subscript array in order to store the new indexing scheme. Some sort of combination of the above both techniques has been also proposed in the literature [18].

Data locality is not exploited by a LPO method. This situation could be relieved by adding an external preprocessing stage before executing the irregular code. This stage is in charge of reordering the input data (that will fill the subscript arrays) with the aim of optimizing locality [9, 3]. However, these techniques have a high algorithmic complexity and normally they have difficulties to be used in dynamic codes.

DPO methods, on the other hand, are designed to exploit, at runtime, data locality, specially inter-loop locality, at the cost of reducing a fraction of parallelism (including computation replication). Intra-loop locality could be, additionally, exploited externally by means of a preprocessing reordering algorithm. Other interesting characteristic is that usually memory overhead is much lower than in basic LPO methods, improving significantly the scalability properties.

An important drawback of DPO methods is that they may exhibit workload unbalancing, penalizing their overall performance, depending on input data. This problem could be reduced, at least partially, by an external renumbering of input data [9, 3]. A different solution would be to introduce some load balancing support inside the DPO method. This approach is discussed in the next section.

3 Balancing Workload in DPO Methods

Generically, methods in the DPO class are based on an uniform block partitioning of the reduction array, as this way data locality may be exploited. However, as loop iterations are assigned to the parallel threads depending on the block they write in, this may introduce important workload unbalance. Note that this situation is not usual concerning typical numerical applications, but potentially

it could appear for specific memory access patterns (that represents non uniform problem domains).

In this section we present two approaches to improve the workload balancing of DPO methods. The first approach is oriented to balance generic non uniform load distributions. It is based on the subpartitioning of the reduction array blocks into subblocks of the same size. The second approach, on the other hand, is oriented to special cases where the load unbalance is due to a high number of write operations on small regions of the reduction array (these were called regions of *high degree of contention* in [18]). The solution proposed consists of the local replication of some of these regions.

In the rest of the section we will show how we can modify in an easy and efficient way a DPO method in order to implement the above mentioned approaches. To simplify the description of these modifications the DWA-LIP technique will be taken as the working DPO method.

3.1 Generic Load Balancing Approach

To balance the workload among the threads while keeping exploited data locality, a good approach would be to partition the reduction array into blocks of different size, with the aim of minimizing execution time. However, the inspector cost for such solution would be presumably excessively high.

A much more lighter and simpler approach would be to partition the reduction array into small subblocks, in a number multiple of the number of parallel threads. This way, blocks of different sizes may be built by grouping, in a suitable way, certain number of contiguous subblocks.

The problem is how we can implement such an approach in a DPO method without losing its beneficial properties and keeping at most its computational structure (that is, trying to not introduce too much algorithmic complexities). We will explain next the specific case of DWA-LIP. Fig. 3 shows the parallel computational structure of the DWA-LIP method (that is, the execution phase), with no load balancing support. The inspector was in charge of assigning the reduction array to the parallel threads by blocks of the same size. As noted (and explained in the previous Section), the computation proceeds with synchronized stages, each one composed of sets of iterations (of the form $(B_{min}, \Delta B)$) that are executed in parallel.

A seamlessly modification of the DWA-LIP method to support generic load balancing is shown graphically in Fig. 4. The execution phase is practically unmodified, as the inspector is in charge of all the work. The inspector now operates as before but considering subblocks instead of blocks. It builds the synchronized iteration sets as if the number of parallel threads is equal to the number of subblocks. As the number of actual threads is much lower (a fraction) then those may be grouped into balanced supersets of different size. In Fig. 4 we have called $(i', \Delta B)$ to the $i-$th balanced group of iteration sets, that is, the $i-$th balanced superset for a certain value ΔB. We observe each $(i', \Delta B)$ is an aggregation of sets of the form $(k, \Delta B)$, and so the iterations in that superset write in adjacent reduction array subblocks.

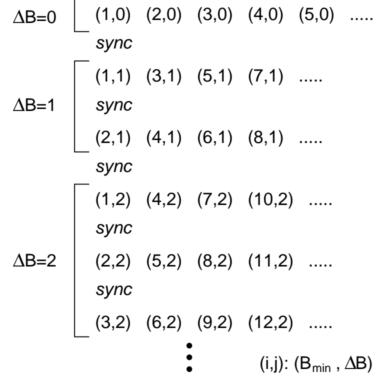

Fig. 3. Parallel flow computation in the original DWA-LIP method

The additional complexity introduced in the inspector by the load balancer is a small fraction of the original one, as the complexity of building supersets is $\mathcal{O}(nSubBlocks^2)$, being $nSubBlocks$ the total number of subblocks (and usually this number is much lower than the size of the reduction loop). The extra memory overhead becomes now $\mathcal{O}(nSubBlocks^2)$, instead of $\mathcal{O}(nThreads^2)$, being $nThreads$ the total number of threads in the system. This will not be significant while $nSubBlocks^2$ is much lower than the size of the reduction array.

The execution phase of the modified DWA-LIP handles the supersets into synchronized stages in the same way as the original DWA-LIP. In order to do that we will execute in parallel stages of supersets. In the original DWA–LIP, we have iterations sets of the form $(i + k(\Delta B + 1), \Delta B)$, $k = 0, 1, ...$, that are executed in parallel (they constitute a stage) because they issue conflict-free write operations. As a consequence, in the modified DWA–LIP, if we assure that the supersets of the form $(i', \Delta B)$ have at least r sets then all supersets of the form $(i' + k\Delta^{LB}, \Delta B)$, where $\Delta^{LB} = \left\lfloor \frac{\Delta B - 1}{r} + 1 \right\rfloor$, issue also conflict-free writes, and thus may be executed fully parallel. It can be proven the best value that maximizes parallelism is $r = \min\left(\Delta B, \frac{nSubBlocks}{nThreads}\right)$. With this value, we have $\Delta^{LB} = \left\lceil \Delta B \frac{nThreads}{nSubBlocks} \right\rceil$.

The final number of supersets in each parallel stage should not be greater than the number of actual threads. The new execution phase works similarly than the original one but operating on supersets.

3.2 Local Expansion Load Balancing Approach

There are situations that suffer from load unbalancing that deserves to be considered as a special case. This situation arises when we find that many loop iterations write on specific and small regions of the reduction array (regions of high contention). We may deal with this case using the approach proposed in the previous section, but it is not difficult to design a more effective solution.

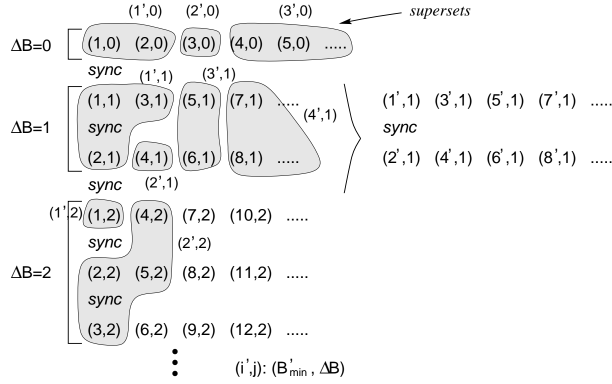

Fig. 4. Parallel flow computation in the DWA-LIP method after including generic load balancing support. The indices i of the iteration sets (i, j) correspond to subblocks.The indices i' of the pairs (i', j) corresponds to the balanced iteration supersets. In any case, index j is ΔB

This contention problem can be easily detected by adding to the inspector of the DPO method a stage of histogram analysis. Indeed, in the case of the DWA-LIP technique, this information is contained in the actual inspector data structure.

It can be observed that as smaller is the size of a contention region lower number of threads can execute the high number of iterations writing in such region (and thus, generating unbalancing). A easy way of relieving this problem consists of the replication on the threads of the block(s) containing the contention region. This way, write conflicts on that region disappear and thus the iterations can be redistributed on a greater number of threads.

With this approach, the data locality exploitation property of the DPO method is maintained without requiring the large amount of extra memory needed by a LPO method like array expansion or replicated buffer. Selective privatization also tries to replicate extra memory as low as possible, but no data locality is considered at all.

In the case of the DWA–LIP method, the replication of a reduction array block implies that the loop iterations in the affected sets $(B_{min}, \Delta B)$ are moved to sets with lower ΔB. This fact increases the parallelism available. In addition, the iterations of sets with $\Delta B = 0$ that write in the replicated block can be assigned to any thread, allowing this way a better balancing of the workload.

The extra memory overhead that the local replication introduce is equal to the size of the reduction array multiplied by the number of replicated blocks. If the problem is very unbalanced, this last number is much lower than the total number of blocks, and thus the total extra memory cost would be much lower than in LPO methods, like array expansion or replicated buffer.

Fig. 5 depicts the access pattern histogram for the sparse matrix *av41092* [2], showing that for different numbers of threads there always exist regions of high

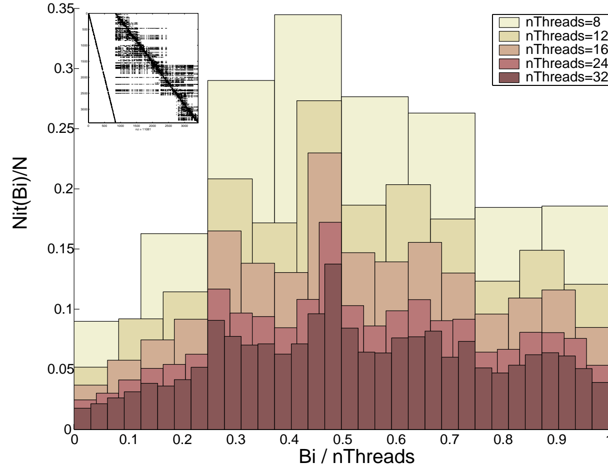

Fig. 5. Histogram of the reduction array access pattern for the sparse matrix *av41092* (from the Univ. of Florida Collection). B_i is the reduction array block index, $nThreads$ is the total number of threads, $Nit(B_i)$ is the number of iterations that write in block B_i, and N is the total number of loop iterations

contention. In Fig. 6 of the same figure shows the theoretical performance for the execution phase of the locally expanded DWA–LIP on the same sparse matrix, for different values of number of expanded blocks, chosen from those that exhibit higher contention. In the figure, `Tnorm` represents the evaluated parallel reduction execution time normalized to the execution on one thread.

4 Experimental Evaluation

We have experimentally evaluated the proposed load balancing solutions and compared with other parallel irregular reduction methods on a SGI Origin2000 multiprocessor, with 250-MHz R10000 processors (4 MB L2 cache) and 12 GB main memory, using IRIX 6.5. All parallel codes were implemented in Fortran 77 with OpenMP [14] directives, and compiled using the SGI MIPSpro Fortran 77 compiler (with optimization level O2).

The generic load balancing approach were implemented and tested using the Spec Code [13], a kernel for Legendre transforms used in numerical weather prediction. The code includes two reduction procedures, `LTD` and `LTI`, being the former a regular one and the latter an irregular one. The experimental results corresponds only to the second routine (`LTI`). The irregular reduction is inside a nested loop, being the indices of the innermost loop also indirections. There are several reduction arrays but only one subscript array. This means that DPO methods should work efficiently because no loose of parallelism (DWA–LIP)

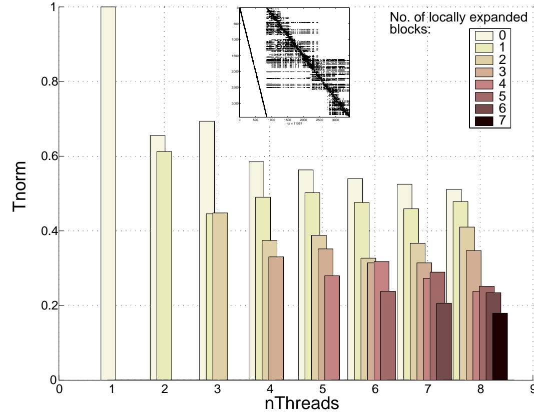

Fig. 6. Evaluation of the parallel execution time of the execution phase of the locally expanded DWA–LIP for the same sparse matrix than in Fig. 5

nor computation replication (LocalWrite) is expected, as there is only one subscript array.

Fig. 7 shows the resulting speedup for the execution phase of several reduction methods on the `LTI` procedure. Pure DPO methods shows suboptimal performance, which is due mainly to the workload unbalance. When introducing the generic load balancing solution into DWA–LIP, the performance is significantly improved. The `K` factor represents the ration between the number of reduction array subblocks and the total number of threads. When increasing `K`, the speedup improves slightly. However, there is no additional improvement for values beyond 8. Note, also, that the extra memory needed for the modified DWA–LIP method is proportional to the square number of subblocks, which for the tested code is much lower than the size of the reduction array.

Array expansion performs poorly, as only the outermost loop of the irregular reduction is parallelized. In this code the innermost loop is irregular and consequently array expansion exhibits high load unbalance. In addition, this technique does not take into account data locality. A possible solution to this problem consists of fusing both loops (using, for instance, loop flattening), but that requires to add an inspector phase to the method.

For this code, the indirection array appearing in the innermost loop and the reduction subscript array are computed only once in a initialization routine. Thus the inspector phase should be executed only once also, and consequently its impact in the overall performance is negligible. For the tested code, the sequential irregular reduction time was 19 sec. while the inspector took 0.18 sec.

The local expansion load balancing approach, on the other hand, was experimented on a simple 2D short-range molecular dynamics simulation [12, 17] (MD2). This application simulates an ensemble of particles subject to a Lennard-Jones short-range potential. To integrate the equations of motion of the particles,

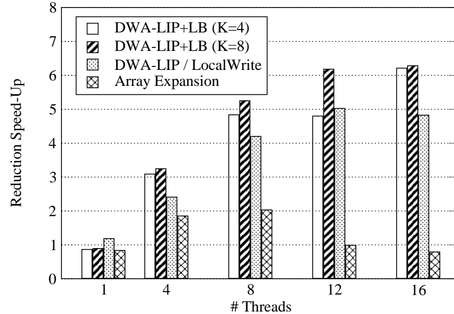

Fig. 7. Speedup of the generic load balancing approach implemented in the DWA–LIP method (DWA–LIP+LB) compared to the original DWA–LIP, LocalWrite and array expansion for the Legendre transformation (Spec Code [13])

a finite-difference leapfrog algorithm on a Nosé -Hoover thermostat dynamics is used. In the core of this code there is an irregular reduction nested loop due to the use of a neighbour list technique to update force contributions. Thus we have two reduction arrays and two subscript arrays. In addition, the subscript array is dynamically updated every 10 time steps. The number of particles simulated is 640K, and it has been introduced artificially a high contention region in the particle domain. To test the impact of the inter-loop locality, the iteration order of the original loop that runs over the neighbour list was randomized.

Fig. 8 the speedup for the execution phase of the local expanded load balancing technique implemented in the DWA–LIP method, compared to array expansion and selective privatization techniques. Part (a) in the figure corresponds to the original code (sorted neighbour list) while part (b) corresponds to the randomized code. As the inter-loop locality of the original code is relatively high, and the fraction of conflicting reduction array elements (elements written by more than one thread) is very low, then techniques like selective privatization performs very well. DWA–LIP works poorly due to the high unbalance of the load. When introducing local expansion, the situation improves significantly but it does not reach the level of selective privatization due to the cost of handling replicated blocks (while selective privatization works directly on the original reduction array most of the time). Array expansion performs worse due to the high overhead of operating on expanded arrays and the final collective operation.

When the neighbour list is randomized, the original inter-loop locality is lost. That produces a hard impact on the performance of selective privatization, as the number of conflicting elements increases drastically. However, DWA–LIP and its variants maintain their performance at similar levels than before, as these methods exploit at runtime inter-loop locality.

Note that beyond 8 processors, the high contention region covers more than one block. Thus, we need to locally expand two blocks in order to keep balancing the load (as shown in the above both plots).

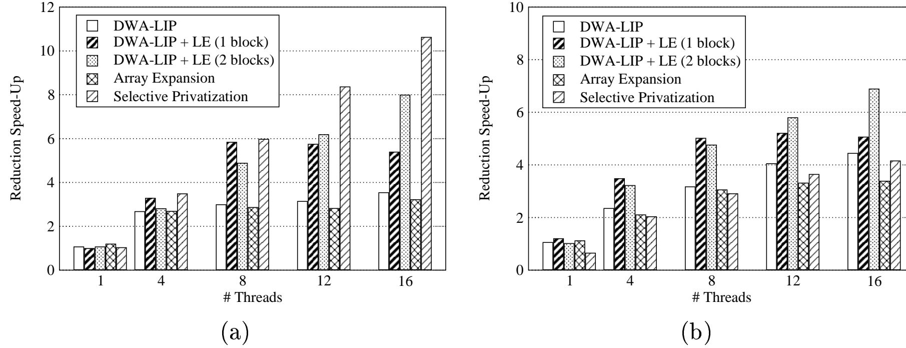

Fig. 8. Speedup of the local expansion load balancing approach implemented in the DWA–LIP method (DWA–LIP+LE) compared to the original method, array expansion and selective privatization for the MD2 simulation code. (a) corresponds to the original code, while in (b) the loop that runs over the neighbour list of particles was randomized

Regarding the cost of the inspector phase, the total sequential reduction time was 10 sec. (original) and 19 sec. (randomized). The inspector execution time for DWA–LIP was 1.25 sec. for all cases and variants, while the same for the selective privatization was 2.4 sec.

Finally, Fig. 9 depicts the extra memory overhead that the tested reduction methods exhibit. Note that selective privatization is very sensitive to the inter-loop locality of the original code, either in performance and in extra memory, while the DPO methods succeed to exploit it at runtime. In part (b) of the figure, the memory overhead due to the inspector data structures has been included. The main overhead in DWA–LIP corresponds to the size of the subscript array (in MD2, this size is three times larger than the size of the reduction arrays). In selective privatization, a copy of each subscript array is needed to translate the indices to the selective private replicas of the reduction arrays. In MD2, this overhead is twice than in DWA–LIP.

5 Conclusions

There is much interest in the compiler community to parallelize efficiently irregular reductions, as these operations appear frequently in the core of many numerical applications. Different parallelization techniques have been proposed elsewhere, that we have classified in this paper into two classes: LPO (*Loop Partitioning Oriented* techniques) and DPO (*Data Partitioning Oriented* techniques). Methods in the first class assign blocks of reduction loop iterations to the cooperating threads. Methods in the second class assign blocks of the reduction arrays to the threads, and further trying to execute most of the loop iterations that write in the owned blocks. We have analyzed both classes in terms

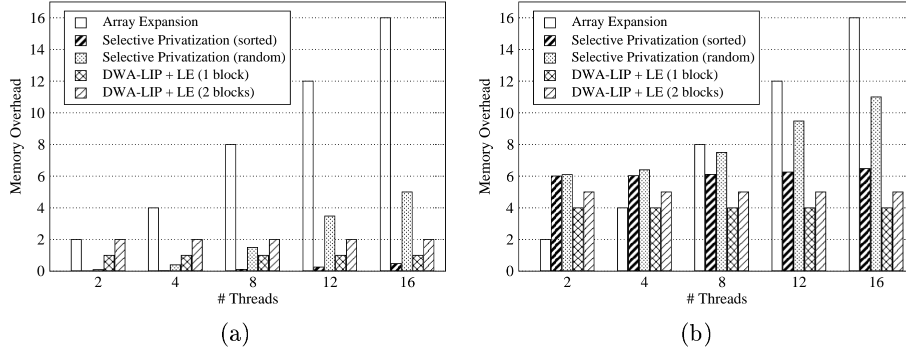

Fig. 9. Memory overhead for different reduction parallelization methods for the MD2 code (units are normalized to the total size of the reduction arrays): (a) concerning only replicated reduction arrays, and (b) including also the inspector data structures

of a set of performance aspects: data locality, memory overhead, parallelism and workload balancing.

The load balancing performance aspect was not sufficiently analyzed in parallel reduction methods, specially those in the DPO class, which are very sensitive to that aspect. In this paper we have proposed two new techniques to introduce load balancing into a DPO method. The first technique, based on the subblocking of the reduction arrays, is generic, as it can deal with any kind of load unbalancing present in the problem domain. The second technique handles a special case of load unbalance, appearing when there are a large number of write operations on small regions of the reduction arrays. The proposed solution is based on the privatization of the blocks making up those regions.

In this paper we show efficient implementations of the proposed solutions to load balancing for the DWA–LIP DPO method. Experimental results allow us to conclude that it is possible to improve the performance of DWA–LIP for very unbalanced problems with no significant loss of data locality and no substantial increment in extra memory overhead and algorithmic complexity.

Acknowledgments. The authors wish to thanks CEPBA (*European Center for Parallelism of Barcelona*, Spain) for the computing time provided on the SGI Origin2000.

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 1512, University for Illinois at Urbana-Champaign, Center for Supercomputing R&D., December 1996.
2. T. Davis, The University of Florida Sparse Matrix Collection. *NA Digest*, 97(23), June 1997.

3. 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.
4. 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 Conference (EuroPar'99)*, pages 422–429, Tolouse, France, August-September 1999.
5. 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)*, pages 78–87, Santa Fe, NM, May 2000.
6. E. Gutiérrez, R. Asenjo, O. Plata, and E.L. Zapata. Automatic Parallelization of Irregular Applications. *J. Parallel Computing*, 26(13–14):1709–1738, December 2000.
7. H. Han and C.-W. Tseng, Improving Compiler and Run-Time Support for Irregular Reductions Using Local Writes. In *Proceedings of the 11th Workshop on Languages and Compilers for Parallel Computing (LCPC'98)*, pages 181–196, Chapel Hill, NC, August 1998.
8. H. Han and C.-W. Tseng, Efficient Compiler and Run–Time Support for Parallel Irregular Reductions. *J. Parallel Computing*, 26(13–14):1709-1738, December 2000.
9. 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.
10. H. Han and C.-W. Tseng, A Comparison of Parallelization Techniques for Irregular Reductions. In *Proceedings of the 15th IEEE International Parallel and Distributed Processing Symposium (IPDPS'2001)*, San Francisco, CA, April 2001.
11. 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.
12. J. Morales and S. Toxvaerd. The Cell-Neighbour Table Method in Molecular Dynamics Simulations. *Computer Physics Communication*, 71:71–76, 1992.
13. 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)*, pages 278–285, Rhodes, Greece, June 1999.
14. OpenMP Architecture Review Board. OpenMP: A Proposed Industry Standard API for Shared Memory Programming. `http://www.openmp.org`, 1997.
15. R. Ponnusamy, J. Saltz, A. Choudhary, S. Hwang, and G. Fox. Runtime Support and Compilation Methods for User-Specified Data Distributions. *IEEE Transactions on Parallel and Distributed Systems*, 6(8):815–831, June 1995.
16. L. Rauchwerger and D. Padua. The LRPD Test: Speculative Run-Time Parallelization of Loops with Privatization and Reduction Parallelization. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 218–232, La Jolla, CA, June 1995.
17. S. Toxvaerd. Algorithms for Canonical Molecular Dynamics Simulations. *Molecular Physics*, 72(1).159–168, 1991.
18. H. Yu and L. Rauchwerger. Adaptive Reduction Parallelization Techniques. In *Proceedings of the 14th ACM International Conference on Supercomputing (ICS'2000)*, pages 66–77, Santa Fe, NM, May 2000.