On Workload Balancing of Parallel Irregular Reductions*

E. Gutiérrez O. Plata E.L. Zapata

Abstract— **Much effort has been devoted recently to efficiently parallelize irregular reductions. 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). 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 aspect sufficiently analyzed in the literature in 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. Efficient implementations of the proposed solutions for the**DWA–LIP **DPO method are presented, and experimentally tested on static and dynamic kernel codes and compared with other parallel reduction methods.**

Key words— **Irregular reduction, workload balancing, locality exploitation**

I. Introduction

MANY scientific/engineering applications are based on complex data structures that introduce irregular memory access patterns. Run–time techniques have been proposed in the literature to support the parallelization of irregular codes, like those based on the inspector-executor paradigm [14], or the speculative execution of loops in parallel [15].

Run–time techniques 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], [10]. Reduction operations represent an example of such computational structures, frequently found in the core of many irregular numerical applications.

In this paper we classify the most important irregular reduction parallelization techniques into two main classes. 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.

Load balancing is not an aspect sufficiently discussed and analyzed in the considered parallel reduction methods. In this paper we propose two techniques to introduce load balancing. 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.

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

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

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

do i = 1,fDim
   Calculate ξ1, ξ2, ..., ξn
   A(f1(i))=A(f1(i)) ⊕ξ1
   A(f2(i))=A(f2(i)) ⊕ξ2
       ...
   A(fn(i))=A(fn(i)) ⊕ξn
enddo
```

Fig. 1. A loop with multiple reductions with an irregular memory access pattern

II. Methods for Reduction Parallelization

We may classify specific solutions to parallelize irregular reductions 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).

We consider in the rest of the paper the general case of a loop with multiple reductions, as shown in Fig. 1. 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 loop, Fig. 2 shows a graphical representation of techniques LPO and DPO.

The simplest solution in the LPO class is based on critical sections that enclose the accesses to the reduction array. 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 reduc-

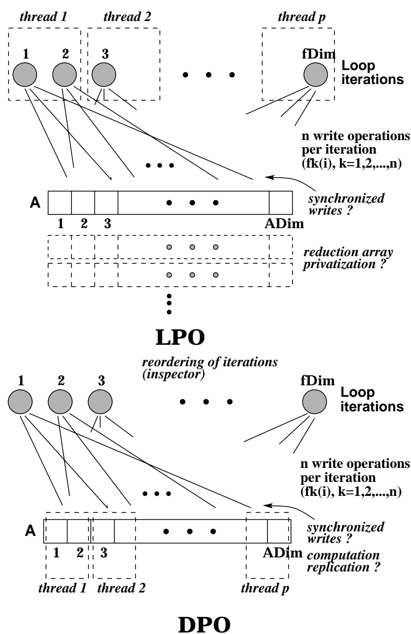


Fig. 2. General schematic representation of the LPO and DPO classes of reduction parallelization techniques

tion 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 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 arrays by the number of parallel threads instead of using private copies of them.

These two methods transform the reduction loop into a fully parallel one, removing the data dependencies as a result of the privatization of the reduction array. However 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.

Two methods has been proposed in the literature in the DPO class. One method was termed LOCAL-WRITE [7], [9], 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 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], [5], [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

A. 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 re-orderings, 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 in Fig. 3 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 [10] assigns a private buffer to each thread

	Inter-Loop Locality	Intra-Loop Locality	Memory Overhead	Parallelism	Workload Balance
LPO	extrinsic	extrinsic	High/Medium/Low	High	High
DPO	High	extrinsic	Low	High/Medium/Low	extrinsic

Fig. 3. Typical performance characteristics for the LPO and DPO classes of parallel irregular reduction methods. The term `extrinsic` means that the property is not intrinsically exploited by the method, but it depends on input data

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. Other method is *selective privatization* [16], 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. Some sort of combination of the above both techniques has been also proposed in the literature [16].

Data locality is not exploited by a LPO method. This situation could be relieved by adding an external preprocessing that reorders the input data [8], [3]. However, these techniques have a high algorithmic complexity.

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). Other interesting characteristic is that usually memory overhead is much lower than in basic LPO methods.

An important drawback of DPO methods is that they may exhibit workload unbalancing depending on input data. This problem could be reduced, at least partially, by an external renumbering of input data [8], [3]. A different solution would be to introduce some load balancing support inside the DPO method. This approach is discussed in the next section.

III. 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 workload unbalance, although this situation is not usual concerning typical numerical applications,

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 [16]). The solution proposed consists of the local replication of some of these regions. The DWA-LIP technique has been modified in an easy an efficient way to support these solutions.

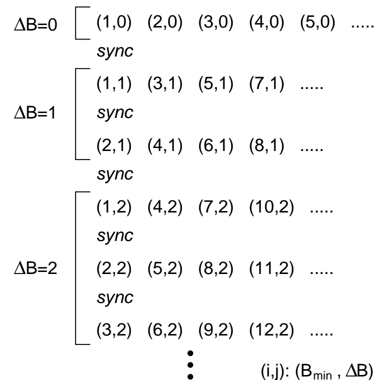


Fig. 4. Parallel flow computation in the original DWA-LIP method

A. 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 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 original computational structure. We will explain next the specific case of DWA-LIP. Fig. 4 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 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. 5. 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. 5 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

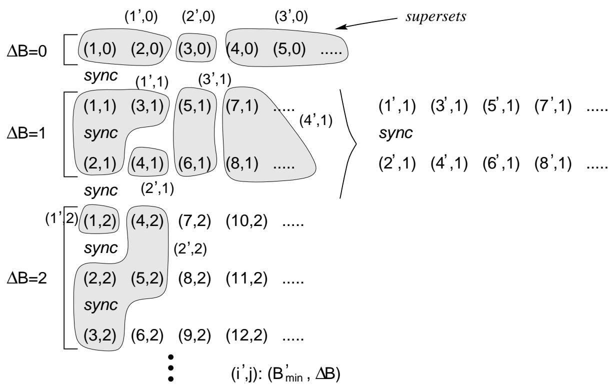


Fig. 5. Parallel flow computation in the DWA-LIP method after including generic load balancing support. The indices i of the iteration sets (i, j) correspond to subblocks.The indices i' of the pairs (i', j) corresponds to the balanced iteration supersets. In any case, index j is ΔB

so the iterations in that superset write in adjacent reduction array subblocks.

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 $\Delta^{LB} = \lfloor \frac{\Delta B - 1}{r} + 1 \rfloor$, 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 $\Delta^{LB} = \lceil \Delta B \frac{nThreads}{nSubBlocks} \rceil$.

B. 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).

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.

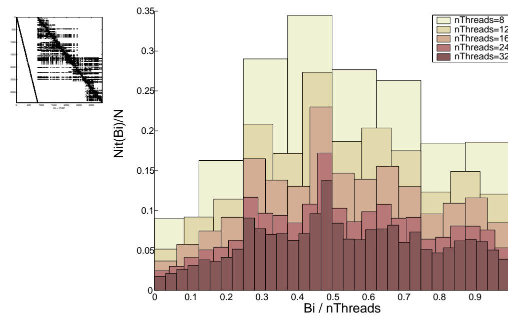


Fig. 6. Histogram of the reduction array access pattern for the sparse matrix *av41092* (from the Univ. of Florida Collection). B_i is the reduction array block index, $nThreads$ is the total number of threads, $Nit(B_i)$ is the number of iterations that write in block B_i, and N is the total number of loop iterations

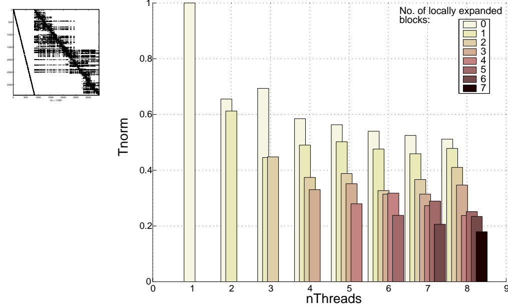


Fig. 7. Evaluation of the parallel execution time of the execution phase of the locally expanded DWA–LIP for the same sparse matrix than in Fig. 6

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. 6 depicts the access pattern histogram for the sparse matrix *av41092* [2], showing that for different numbers of threads there always exist regions of high contention. Fig. 7 shows the theoretical performance for the execution phase of the locally expanded DWA–LIP for the access pattern of the sparse matrix *av41092* [2] 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.

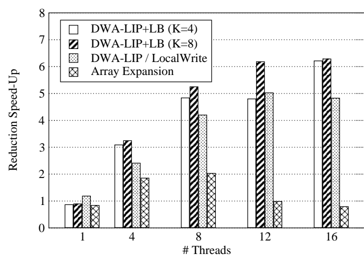
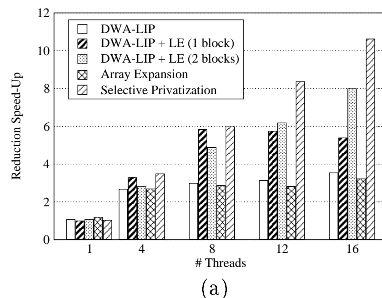
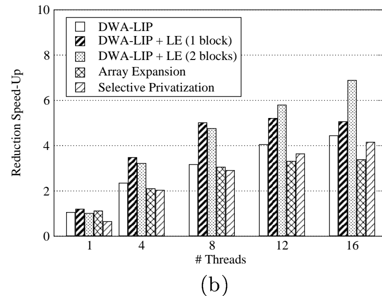


Fig. 8. Speedup of the generic load balancing approach implemented in the DWA–LIP method (DWA–LIP+LB) compared to the original DWA–LIP, LocalWrite and array expansion for the Legendre transformation (Spec Code)



(a)



(b)

Fig. 9. 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

IV. 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 [13] directives.

The generic load balancing approach were implemented and tested using the Spec Code [12], a kernel for Legendre transforms used in numerical weather prediction. The experimental results corresponds to the routine (`LTI`) which has an irregular reduction inside a nested loop. The indices of the innermost loop also are indirections. Because there is only one subscript array, DPO methods should work efficiently because neither loose of parallelism (DWA–LIP) nor computation replication (LocalWrite) is expected.

Fig. 8 shows the resulting speedup for the execution phase of several 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, although there is no additional improvement for values beyond 8.

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.

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. 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 [11] (MD2). This application simulates an ensemble of particles subject to a Lennard-Jones short-range potential. 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. 9 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 main-

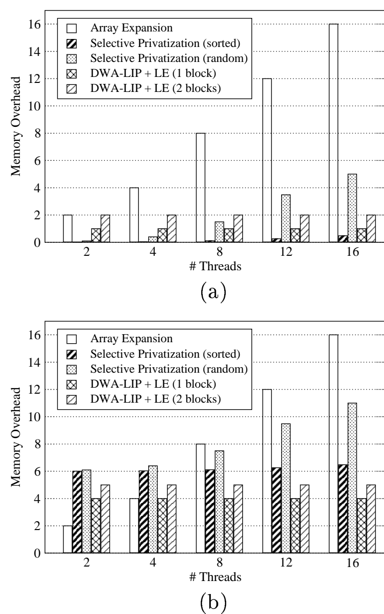


Fig. 10. Memory overhead for different reduction parallelization methods for the MD2 code (units are normalized to the total size of the reduction arrays): (a) concerning only replicated reduction arrays, and (b) including also the inspector data structures

tain their performance at similar levels than before, as these methods exploit at runtime inter-loop locality.

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. 10 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.

V. Conclusions

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.

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. Tech. Rep. 1512, Univ. of Illinois at Urbana-Champaign, CSRD, 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. *ACM Int'l. Conf. on Programming Language Design and Implementation*, pp. 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. *5th International Euro-Par Conference* , pp. 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. *14th ACM Int'l. Conf. on Supercomputing*, pp. 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, Efficient Compiler and Run-Time Support for Parallel Irregular Reductions. *J. Parallel Computing*, 26(13–14):1709-1738, December 2000.

[8] H. Han and C.-W. Tseng, Improving Locality for Adaptive Irregular Scientific Codes. *13th Workshop on Languages and Compilers for Parallel Computing*, Yorktown Heights, NY, August 2000.

[9] H. Han and C.-W. Tseng, A Comparison of Parallelization Techniques for Irregular Reductions. *15th IEEE Int'l. Parallel and Distributed Processing Symp.* , San Francisco, CA, April 2001.

[10] Y. Lin and D. Padua, On the Automatic Parallelization of Sparse and Irregular Fortran Programs. *4th Workshop on Languages, Compilers and Runtime Systems for Scalable Computers*, Pittsburgh, PA, May 1998.

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

[12] N. Mukherjee and J.R. Gurd, A Comparative Analysis of Four Parallelisation Schemes. *13th ACM Int'l. Conf. on Supercomputing*, pp. 278–285, Rhodes, Greece, June 1999.

[13] OpenMP Architecture Review Board. OpenMP: A Proposed Industry Standard API for Shared Memory Programming. `http://www.openmp.org`, 1997.

[14] 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.

[15] L. Rauchwerger and D. Padua. The LRPD Test: Speculative Run-Time Parallelization of Loops with Privatization and Reduction Parallelization. *ACM SIGPLAN Conf. on Programming Language Design and Implementation*, pp. 218–232, La Jolla, CA, June 1995.

[16] H. Yu and L. Rauchwerger. Adaptive Reduction Parallelization Techniques. *14th ACM Int'l. Conf. on Supercomputing*, pp. 66–77, Santa Fe, NM, May 2000.