On Improving the Performance of Data Partitioning Oriented 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 for reductions 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 properties: data locality, memory overhead, parallelism and workload balancing. In this paper we propose several techniques to increase the exploited parallelism and to introduce load balancing into a DPO method. Regarding parallelism, the solution is based on the partial expansion of the reduction array. For load balance, a first technique is generic, as it can deal with any kind of load unbalancing present in the problem domain. A 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 optimizing solutions for the* DWA–LIP *DPO method are presented, experimentally tested on static and dynamic kernel codes and compared with other parallel reduction methods.*

1. Introduction: Methods for Reduction Parallelization

Reduction operations represent an example of a computational structure 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.

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 first proposed solutions to parallel reductions fall in the LPO class, like those based on critical sections, or on the privatization of the reduction array (*replicated buffer* and *array expansion* techniques [8, 11]) Methods in the DPO class avoid the privatization of the reduction array. In order to determine which loop iterations each thread should execute, 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. Methods in this class are LocalWrite [5, 7] and DWA–LIP [3, 4].

As it will be used in the rest of the paper, we will

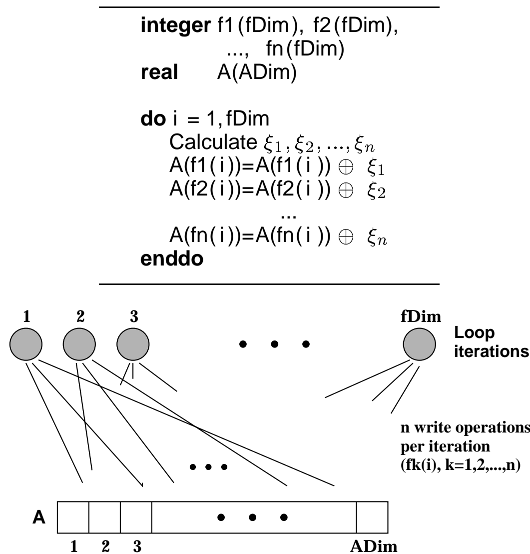


Figure 1. A loop with multiple reductions and a schematic representation of the irregular memory access pattern

explain in detail the DWA–LIP method. 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 (*data write affinity* phase, or 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. A similar scheme is followed in the subsequent stages, until all iterations are exhausted. Fig. 3 shows the data structures for an example code with two subscript arrays, and Fig. 4 explains an OpenMP implementation of this method.

2. Performance Properties of Reduction Methods

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 memory scalability. In addition,

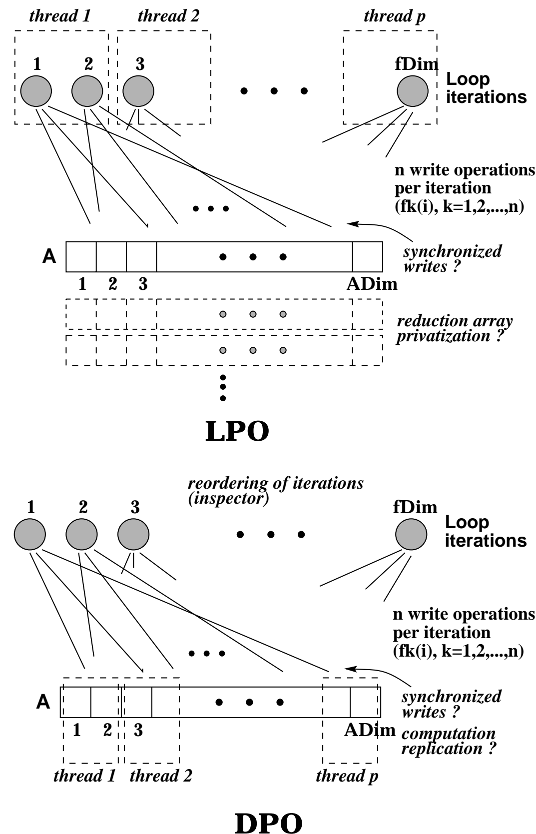


Figure 2. General schematic representation of the LPO and DPO classes of reduction parallelization techniques

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

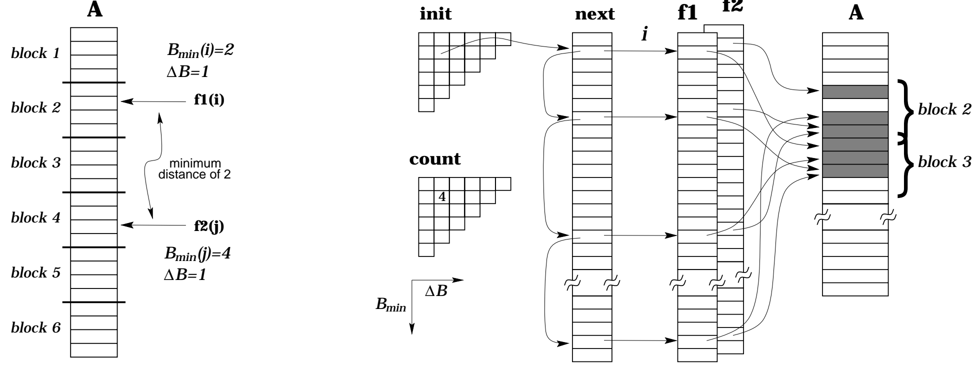


Figure 3. Data structures needed in the execution phase of DWA–LIP **(for two subscript arrays)**

	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

Table 1. Typical performance properties 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**

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

do △B = 0, nThreads−1
   do s = 1, △B+1
c$omp parallel do
      do B_min = s, nThreads−△B, △B+1
         i = init (B_min,△B)
         cnt = count(B_min,△B)
         do k = 1,cnt
            Calculate ξ1, ξ2, ..., ξn
            A(f1(i))=A(f1(i)) ⊕ ξ1
            A(f2(i))=A(f2(i)) ⊕ ξ2
                     ...
            A(fn(i))=A(fn(i)) ⊕ ξn
            i = next(i)
         enddo
      enddo
c$omp end parallel
   enddo
enddo
```

Figure 4. The execution phase of DWA–LIP **in OpenMP (**nThreads **is the number of threads cooperating in the computation)**

locality inside one reduction loop iteration.

Different solutions has been proposed recently to reduce the high memory overhead of LPO methods, like *reduction table* [8], *selective privatization* [11] and others [11]. 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 [6, 1]. Usually 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). 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.

3. Improving the Performance of DPO Methods

In some cases DPO methods may perform under optimal, either due to loss of parallelism (a lot of conflicting interblock writes) or to workload unbalance. In this section we will propose solutions to solve these problems, increasing consequently the overall performance of the method. To simplify the discussion, we will take DWA–LIP as the base DPO method to improve.

3.1. Solutions to Parallelism Loss

In DPO methods we can always trade memory overhead for parallelism exploitation, as privatization helps in eliminating write conflicts. In the case of DWA–LIP, write conflicts are represented by non-null entries in the second and so columns of the init triangular matrix (see Fig. 3). The execution of the loop iterations associated to these entries is accomplished in synchronized phases (to avoid write conflicts), using each time a fraction of the total number of available threads, loosing this way parallelism.

Parallelism may be, thus, increased if the reduction array is partially replicated (a fixed number of times, less than the number of threads). The number of copies of the reduction array will be the *partial expansion factor* (ρ). This replication increases the parallelism exploitable by DWA–LIP, as, for a particular ΔB value (that is, a column in init, Fig. 3), conflicting iteration sets may now be non-conflicting because they have the possibility of updating different private copies of the reduction array. In other words, as ρ private copies of the reduction array are available, there is always the opportunity of having, at least, ρ threads working in parallel.

The hard problem here is how to schedule the iteration sets so as we can benefit from this parallelism most of the time. Top part in Fig. 5 depicts the dataflow of the execution phase of the original DWA–LIP method. We observe that for each column of nodes, the number of conflicting super-sets (represented in the figure by linked nodes of the same color) is equal to $\Delta B+1$. If the reduction array is replicated ρ times, then, for each column, ρ super-sets stop being conflicting, as each one may work on a different private copy. Taking in mind this fact, we can prove that the new method shares the same execution model than the base one but considering that the number of conflicting super-sets in each column is now $\Delta^{exp} = \left\lfloor \frac{\Delta B}{\rho} + 1 \right\rfloor$.

There are different possibilities to assign private copies of the reduction array to super-sets of iterations sets. A simple one, that results into a compact code, consists in assigning cyclically each super-set to each private buffer, from top to bottom in the corresponding column. This execution model results in a parallelism exploitation lower than ρ from columns where $\Delta B > \frac{nThreads-1}{2}$. To avoid this parallelism loss, for these columns the iteration sets are grouped into super-sets of, at most, ρ elements. All set in each super-set can be executed in parallel, working on different private arrays.

Bottom part in Fig. 5 depicts the new execution model, for $\rho = 2$. For the four leftmost columns, the execution model is similar to the original DWA–LIP (but considering Δ^{exp}). In general, comparing this execution flow with that in the top part in Fig. 5, we note a significant increase in parallelism.

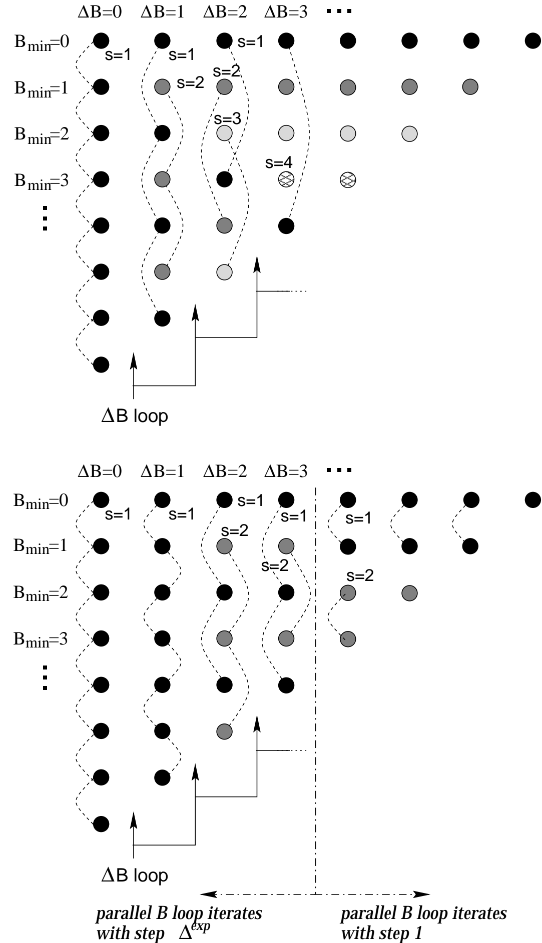


Figure 5. Model of the execution phase of DWA–LIP **(top), and of the partially expanded** DWA–LIP **(bottom)**

3.2. Solutions to Workload Unbalance

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. In this section we present two approaches to improve the workload balancing of DPO methods.

3.2.1 Generic Approach

A generic approach to balance workload could 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.

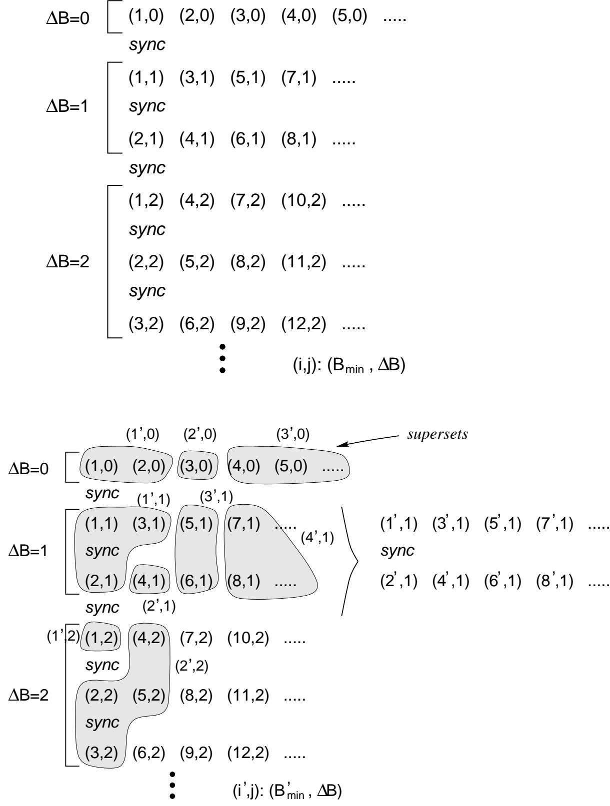


Figure 6. Parallel flow computation in the original DWA-LIP **method (top), and after including generic load balancing support (bottom). (pairs (i, j) correspond to subblocks, while pairs (i', j) corresponds to the balanced iteration supersets)**

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. We will explain next the specific case of DWA-LIP. Top part in Fig. 6 shows the parallel computational structure of the base DWA-LIP method. The inspector was in charge of assigning the reduction array to the parallel threads by blocks of the same size.

A seamlessly modification of the DWA-LIP method to support the proposed generic load balancing is shown graphically in the top part in Fig. 6. 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. 6 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.

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.

3.2.2 Local Expansion 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.

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.

4. Experimental Evaluation

We have experimentally evaluated the proposed solutions to improve the performance of DPO methods (specifically, DWA–LIP) 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 directives, and compiled using the SGI MIPSpro Fortran 77 compiler (with optimization level O2).

The partially expanded DWA–LIP method has been experimentally tested on the EULER code (motivating application suite of HPF-2 [2]). The code includes a single loop with two subscripted reductions on one array with three dimensions, which is placed inside an outer time-step loop. As an static problem, the inspector phase is computed only once. The parallel EULER kernel has been tested using a 1161K nodes mesh with a connectivity of 8 (ratio between edges and nodes). Two versions of the mesh has been generated. One of them is obtained after applying a coloring algorithm to the edges, so a low inter-loop locality should be expected. In the other version the list of edges has been sorted, resulting in a expected higher inter-loop locality.

In Fig. 7 we have plotted the speedup (referenced to the sequential execution time) for the computation phase of the base DWA-LIP, its partially expanded version and array expansion. Observe that the pure DWA-LIP method ($\rho = 1$) has a lower performance than array expansion technique. The reason is the parallelism loss due to iteration sets with a high ΔB parameter, resulting from the input data set used.

Due to the low inter-loop reference locality we expect a bad behavior of array expansion in the colored mesh. We observe that for more than 8 processors we can reach a 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.

In the sorted mesh case, the main limitation is the parallelism loss caused by the low intra-loop locality. Nevertheless we observe that for a given number of threads the parallel execution time of DWA–LIP decreases if the ρ factor is increased. This effect is more significant for a higher number of threads, so that both partially expanded DWA-LIP and array expansion provide almost the same speedup for 16 threads and $\rho = 8$. In both cases, the overhead of the prefetching phase is not significant (it represents about a 5% of time of the computation phase).

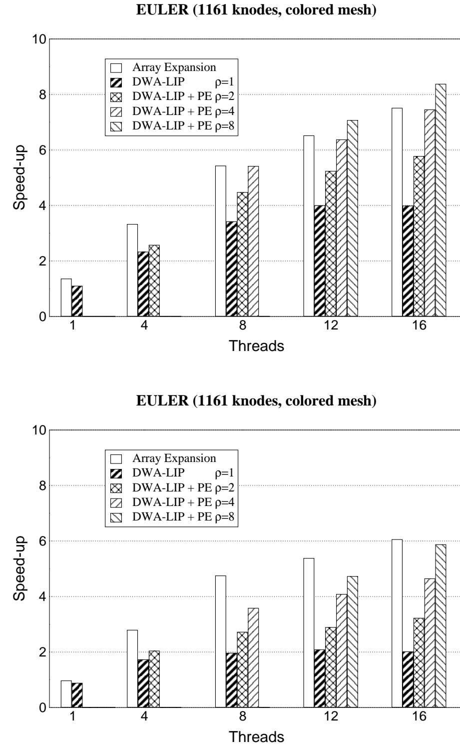


Figure 7. Speedups for the parallel EULER code using DWA-LIP, its partially expanded version (+PE) and array expansion methods (colored (top) and sorted (bottom) meshes)

The extra memory needed by both methods is another important overhead factor. 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 $nThreads \ll N$, array expansion needs around 3 times more extra memory than the other method.

The generic load balancing approach were implemented and tested using the Spec Code [9], a kernel for Legendre transforms used in numerical weather prediction. The irregular reduction is inside a nested loop, being the indices of the innermost loop also indirections.

Fig. 8 shows the resulting speedup for the execution

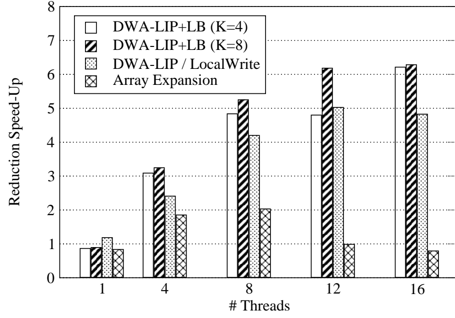
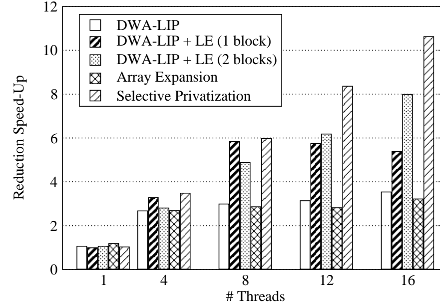


Figure 8. Speedup of the generic load balanced DWA–LIP **method (**DWA–LIP+LB**) compared to the original** DWA–LIP**,** LOCALWRITE **and array expansion for the Legendre transformation**



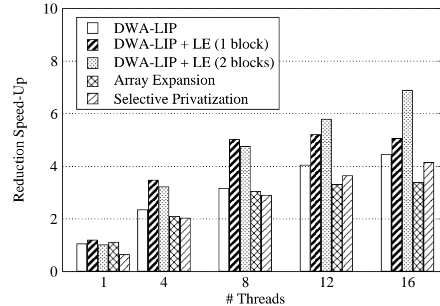


Figure 9. Speedup of the locally expanded DWA–LIP **method (**DWA–LIP+LE**) compared to the original method, array expansion and selective privatization for the MD2 simulation code. (top) corresponds to the original code, while in (bottom) the loop that runs over the neighbour list of particles was randomized**

phase of several reduction methods. 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. 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. Finally, the overhead of the inspector phase is negligible (less than 1% of the reduction time), as it is executed only once.

The local expansion load balancing approach, on the other hand, was experimented on a simple 2D short-range molecular dynamics simulation [10] (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. 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. The impact of the inspector phase, in both cases, is around 1% for locally expanded DWA–LIP and 2.5% for selective privatization.

5. Conclusions

In this paper we show efficient implementations for optimizing parallelism and workload balancing for DPO parallel reduction methods, showing specific implementations in the case of DWA–LIP method. Experimental results allow us to conclude that it is possible to improve the performance of DWA–LIP with no significant loss of data locality and no substantial increment in extra memory overhead and 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.