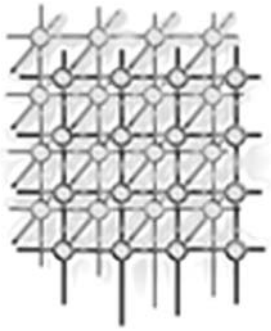
CONCURRENCY AND COMPUTATION: PRACTICE AND EXPERIENCE
Concurrency Computat.: Pract. Exper. 2003; **0**:1–20 *Prepared using* cpeauth.cls *[Version: 2002/09/19 v2.02]*

Data Partitioning-Based Parallel Irregular Reductions



Eladio Gutiérrez*, Oscar Plata and Emilio L. Zapata

Department of Computer Architecture, University of Málaga, E-29071 Málaga, Spain

SUMMARY

Different parallelization methods for irregular reductions on shared memory multiprocessors have been proposed in the literature in recent years. We have classified all these methods and analyzed them in terms of a set of properties: data locality, memory overhead, exploited parallelism, and workload balancing. In this paper we propose several techniques to increase the amount of exploited parallelism and to introduce load balancing into an important class of these methods. Regarding parallelism, the proposed solution is based on the partial expansion of the reduction array. Load balancing is discussed in terms of two techniques. The first technique is a generic one, as it deals with any kind of load imbalance present in the problem domain. The second technique handles a special case of load imbalance which occurs whenever a large number of write operations are concentrated on small regions of the reduction arrays. Efficient implementations of the proposed optimizing solutions for a particular method are presented, experimentally tested on static and dynamic kernel codes, and compared with other parallel reduction methods.

KEY WORDS: Irregular reductions; Data locality; Workload balancing; Shared-memory multiprocessor; NUMA machines

1. INTRODUCTION

In recent years much research effort has been devoted to developing language and compiler technologies for parallel computers. Parallel language technology has evolved with the aim of enabling users to program parallel computers using methods similar to those used in conventional computers. For instance, two established standards, High Performance Fortran

*Correspondence to: Dept. Arquitectura de Computadores, Universidad de Málaga, P.O. Box 4114, E-29071 Málaga, Spain
E-mail: {eladio,oscar,ezapata}@ac.uma.es
Contract/grant sponsor: Ministry of Education and Culture (CICYT) of Spain; contract/grant number: TIC2000-1658

(HPF) [10, 17] and OpenMP [19], are defined as extensions of conventional languages, Fortran or C, that implement data-parallel or task-parallel programming models. Research in compiler technology for multiprocessors, on the other hand, is usually associated with advances in parallel languages, because powerful translators are needed to produce effective parallel machine codes from explicitly parallelized programs (using HPF or OpenMP, for instance). A step forward, however, is taken if the compiler is capable of carrying out full parallelization. Numerical applications are usually based on complex data structures that introduce irregular memory access patterns. In general, automatic parallelizers obtain suboptimal parallel codes from these applications, because traditional data dependence analysis and optimization techniques are precluded. In order to increase the efficiency of these automatically generated parallel codes, run-time techniques have been proposed, such as those based on the inspector-executor paradigm [20], or the speculative execution of loops in parallel [21].

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 application 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. Reduction operations are defined from associative and commutative operators acting on simple variables (scalar reductions) or array elements inside a loop (histogram reductions). If there are no other dependencies but those caused by reductions, the loop can be transformed to be executed fully parallel.

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 said that the thread owns that block).

A set of properties may be defined such that the above classes of methods can be analyzed and classified. We have included in this set properties such as the exploitation of data locality (inter-iteration and intra-iteration), memory overhead, exploited parallelism, and workload balancing. All these properties have a clear influence on the overall performance of the parallelization method.

As we will see, DPO methods obtain better performance from mainly exploiting inter-iteration data locality with a reduced extra memory overhead (which improves the scalability of the method). However, depending on the application, the above is accomplished at the expense of losing a fraction of the exploitable parallelism (due to additional synchronizations or computation replication). In order to reduce this unwanted effect, we have developed various optimizations. One optimization, named *partial array expansion*, exploits parallelism more while increasing memory overhead. On the other hand, another two optimizations, named *generic load balancing* and *local expansion*, improve workload balancing in order to overall

reduce execution time, while trying to minimize the extra memory needed. One of the goals is to avoid losing the good scalability properties of DPO methods.

The rest of the paper continues with a discussion and classification of the most important methods for irregular reduction parallelization (LPO and DPO classes) on shared memory multiprocessors. Using such a classification, the methods are analyzed in terms of a set of relevant properties. Next, we highlight the problems of exploited parallelism and workload balancing that DPO methods suffer, and propose our solutions, as well as efficient implementations. Finally, experimental results that validate our analysis and optimizations are presented and discussed.

2. OVERVIEW OF IRREGULAR REDUCTION PARALLELIZATION

As mentioned, methods for the parallelization of irregular reductions on shared memory multiprocessors may be classified into two broad categories: LPO and DPO methods. DPO methods are based on the partition of data, so presumably it would be best to use them in NUMA machines. LPO methods, on the other hand, partition reduction loops, so the most appropriate thing would be to use them in uniform memory access multiprocessors. To facilitate the analysis of these classes, in the rest of the paper we will consider the general case of a loop with multiple reductions, as shown in Fig. 1 (the case of multiple nested loops is not relevant to our discussion). A() represents the reduction array (that could be multidimensional), which is updated through multiple subscript arrays, f1(), f2(), ..., fn() (also known as indirection arrays). The symbol $\oplus$ is used as the reduction operator (associative and commutative).

As the contents of the subscript arrays are unkonwn during compilation, loop-carried dependencies may be present and can only be detected at run-time. Subscript arrays are usually computed before executing the reduction and, in some cases, they may be modified during different executions of the reduction loop (for example, in an outer time-step loop). Nevertheless, to preserve the associative and commutative properties, subscript arrays must remain unmodified during the entire execution of one instance of the reduction loop.

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

2.1. Loop Partitioning-Oriented (LPO) methods

The first solutions proposed to parallelize reductions fall in the LPO class. The simplest solution is based on critical sections, where the reduction loop is executed in a fully parallel manner by just placing the accesses to the reduction array inside a critical section. This method exhibits a very high synchronization overhead and, consequently, very low efficiency.

The synchronization pressure can be reduced (or even eliminated) by privatizing the reduction array, as carried out by the use of *replicated buffer* [9, 8] and *array expansion* [3, 1] techniques. The *replicated buffer* method defines private copies of the full reduction array in each thread. 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,

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

do i = 1,fDim
   Calculate ζ1, ζ2, ..., ζn
   A(f1(i))=A(f1(i)) ⊕ζ1
   A(f2(i))=A(f2(i)) ⊕ζ2
   ...
   A(fn(i))=A(fn(i)) ⊕ζn
enddo
```

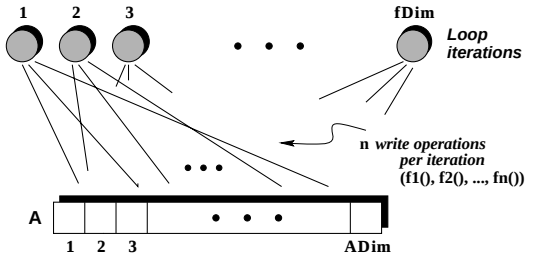


Figure 1. A loop with multiple reductions and a schematic representation of the irregular memory access pattern

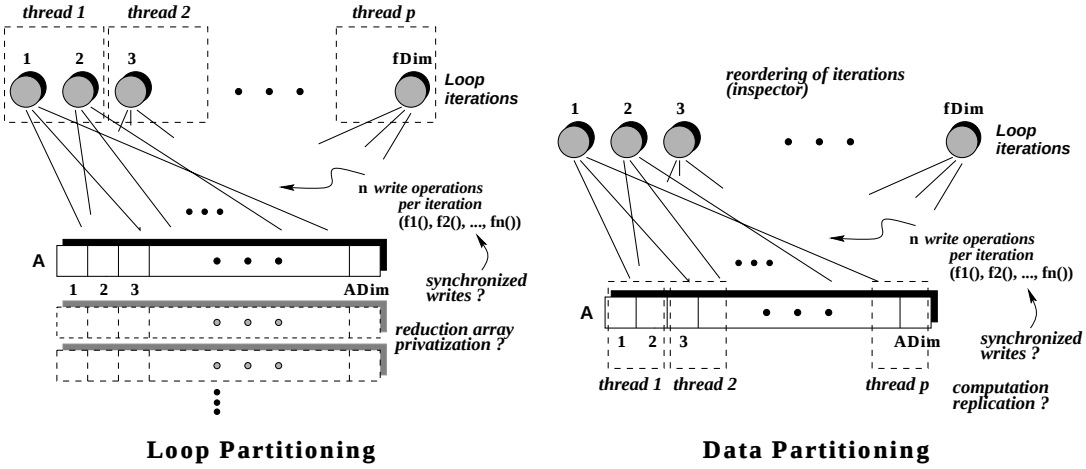


Figure 2. General schematic representation of the LPO (loop partitioning) and DPO (data partitioning) classes of reduction parallelization methods

(*array expansion*), expands the reduction array by the number of parallel threads. Each thread accumulates partial results on its own section of the expanded array. This approach allows us to obtain the final result in a similar way to the first method, but without needing the final synchronization.

Note that these two methods transform the reduction loop into a fully parallel one, as possible loop-carried dependencies disappear as a result of the privatization of the reduction array. However, they encounter scalability problems for large data sets because the

privatization affects the whole reduction array (the memory overhead increases in proportion to the number of parallel threads).

2.2. Data Partitioning-Oriented (DPO) methods

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. For this latter reason the reduction array is usually block partitioned, with the goal of assigning to each thread those loop iterations that write in only one of these blocks (as far as possible). This way, the use of the memory hierarchy is optimized. If the multiprocessor has a NUMA organization, these blocks will be distributed across local memories, so that each node of the machine executes a thread and stores the block written by that thread. This is the case considered in this paper. Methods in this class are LocalWrite [11, 12, 14] and DWA-LIP [5, 6, 7] (Data Write Affinity with Loop Index Prefetching).

The LocalWrite method is based on the *owner-computes rule*. Considering an uniform block partitioning, the above rule implies that each thread owns a portion of the reduction array. The inspector classifies the iterations into two groups: local iterations and boundary iterations. Consider, for instance, a loop with two reductions, A(f1(i)) = ... and A(f2(i)) = ... Given iteration i, let $T_1(i)$ and $T_2(i)$ be the threads that own the block written by the first and the second reduction, respectively. If both owners are the same thread the iteration i is local. Otherwise, it is a boundary iteration. In the execution phase local iterations are assigned to the threads that own the block written by them. However, boundary iterations are split in two, distributing the two reductions between them. Afterwards each one of the two iterations is assigned to the thread that own the corresponding block that is updated. The disadvantage of the splitting process is the replication of computations (computation of one original iteration is carried out by two split iterations). This fact introduces a performance penalty (parallelism loss). Note that a synchronization event is needed to separate all the loops, the local one and each one of the split boundary loops.

As the second method will be used throughout the rest of the paper, we will explain it in detail (DWA-LIP). Let us consider a regular block distribution of the reduction array A() into blocks B_1, B_2, ... B_N. Spatially consecutive blocks are indexed with consecutive natural numbers. For each iteration i of the reduction loop two parameters, $B_{min}(i)$ and $\Delta B(i)$, are defined, where $B_{min}(i) = \min\{1 \leq k \leq N |$ block B_k is referenced by $i\}$ and $\Delta B(i) = B_{max}(i) - B_{min}(i)$ being $B_{max}(i) = \max\{1 \leq k \leq N |$ block B_k is referenced by $i\}$. Note that $B_{min}(i)$ is the minimum index of all blocks updated by iteration i. The inspector (called the *loop-index prefetching* phase, or LIP) is in charge of distributing the iterations of the reduction loop among $(B_{min}, \Delta B)$–sets. Two iterations i and j belongs to the same $(B_{min}, \Delta B)$–set if $B_{min}(i) = B_{min}(j)$ and $\Delta B(i) = \Delta B(j)$. All iterations in the same $(B_{min}, \Delta B)$–set share at least one referenced block, and thus should be executed by the same thread.

During the execution phase (called the *data write affinity* phase, or DWA) of the method, iterations are organized as a synchronized sequence of non-conflicting (parallel) stages. Each stage considers all $(B_{min}, \Delta B)$–sets sharing the same value for ΔB. That is, all iterations $i_1, i_2, ...$ such that $\Delta B(i_1) = \Delta B(i_2) = ...$ It is easy to see that two iterations, i and j, such that $\Delta B(i) = \Delta B(j)$, can be executed concurrently if $B_{min}(i) + \Delta B(i) < B_{min}(j)$. The reason is that these two iterations write in non-overlapping areas of the reduction array. Therefore, in each synchronized stage, all sets of the form $(B_{min}, \Delta B)$, $(B_{min} + (\Delta B + 1), \Delta B)$, $(B_{min} + 2(\Delta B + 1), \Delta B)$, ..., fulfill the above condition and thus can be executed in parallel. Specifically, in the first stage, all $(B_{min}, 0)$–sets are executed in parallel. The second stage is split into two sub-stages. In the first sub-stage, the sets $(1, 1)$, $(3, 1)$, ... are executed in a fully parallel way, followed (after a synchronization point) by the second sub-stage, where sets $(2, 1)$, $(4, 1)$, ... are executed in parallel. A similar scheme is followed in the subsequent stages, until all iterations are exhausted.

Fig. 3 (a) shows the code of the execution phase for $nThreads$ threads. As shown, each stage is characterized by a value of ΔB, ranging from 0 to $nThreads - 1$. For each stage, a total of $\Delta B + 1$ sub-stages are executed, where iterations from independent $(B_{min}, \Delta B)$–sets are computed in parallel. The behaviour of the **parallel-do** OpenMP primitive introduces the synchronization point needed between stages. Fig. 3 (b) shows the data structures built during the LIP phase. The *count* matrix stores the number of iterations of each $(B_{min}, \Delta B)$–set. The list of iterations belonging to $(B_{min}, \Delta B)$–set is stored in the *next* array as a linked list, using the *init* matrix to point to the first iteration in each set. In the figure the sample (2,1)–set with 4 iterations is shown. A graphical representation of the execution flow of the iteration sets is depicted in Fig. 4 (a).

2.3. Properties of the reduction methods

Methods in the LPO and DPO classes have, in some sense, complementary performance characteristics. Methods in the former class exhibit optimal parallelism exploitation (the reduction loop is fully parallel), but data locality is not taken into account and the method lacks memory scalability. In addition, as the reduction loop is uniformly partitioned, these methods usually exhibit a balanced workload.

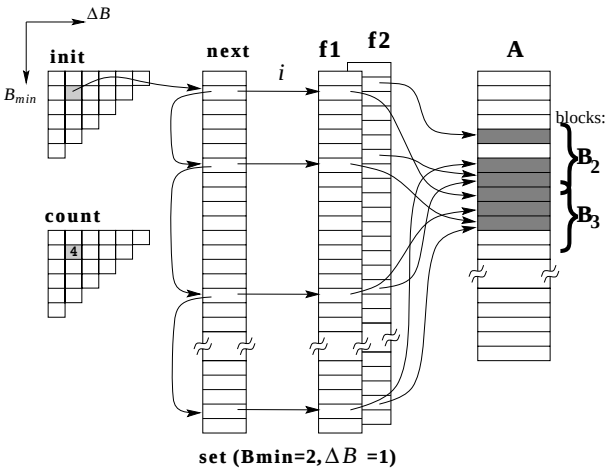
Methods in the second class, however, exploit data locality (write affinity), usually exhibit much lower memory overhead, and are not dependent on the number of threads (the inspector may need some extra buffering to store subscript re-orderings, but always not dependent on the number of threads). However, either the method introduces some computation replication or is organized in a sequence of synchronized phases. That is, the exploitable parallelism is reduced by some fraction, depending on the application and its input data set. In any case, this represents a loss of parallelism. In addition, there is the risk that the number of loop iterations that write in a specific block is very different from the number of iterations writing in another block (workload imbalance).

Table I shows typical characteristics of methods in the LPO and DPO classes taking into account four relevant properties: data locality exploitation, memory overhead, exploited parallelism, and workload balance. Data locality is in turn split into inter-iteration and intra-iteration localities. Inter-iteration locality refers to data locality across different reduction loop

```
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 ΔB = 0, nThreads−1
  do stg = 1, ΔB+1
C$OMP PARALLEL DO
     do B_min = stg, nThreads−ΔB, ΔB+1
        i = init (B_min,ΔB)
        cnt = count(B_min,ΔB)
        do k = 1,cnt
           Calculate ζ1, ζ2, ..., ζn
           A(f1(i))=A(f1(i))  ⊕ ζ1
           A(f2(i))=A(f2(i))  ⊕ ζ2
                    ...
           A(fn(i))=A(fn(i))  ⊕ ζn
           i = next(i)
        enddo
     enddo
  enddo
enddo
```

(a)



(b)

Figure 3. Execution phase of DWA-LIP (a) and its associated data structure (b) (for two subscript arrays)

Table I. 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

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

iterations. Intra-iteration 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 have been proposed recently to reduce this high memory overhead, based on the array expansion and replicated buffer methods. The *reduction table* method [15] assigns a private buffer to each thread of a fixed size (less than the size of the reduction array). Then, each thread works on its private buffer which is 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. Another method is *selective privatization* [22], where the replication includes only those elements referenced by various threads. It first determines (inspector phase) which elements these are and then allocates private storage space for them. Each thread 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 these two techniques has been also proposed in the literature [22].

Data locality is not exploited by LPO methods. This situation could be helped 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 [13, 2]. However, these techniques have high algorithmic complexity and normally are difficult to use in dynamic codes.

On the other hand, DPO methods are designed to exploit data locality, at runtime, – especially inter-iteration locality – at the cost of reducing a fraction of parallelism (by means of extra synchronizations or computation replication). Additionally, intra-iteration locality could be exploited externally by means of a preprocessing reordering algorithm. Another interesting characteristic is that memory overhead is usually much lower than in basic LPO methods, significantly improving the scalability of the methods.

The important drawbacks of DPO methods are, on the one hand, the fraction of exploitable parallelism lost due to possible extra synchronizations or computation replication, and, on the other hand, the workload imbalance that they may exhibit (this depends on how the input data set is structured). This last problem may be reduced, at least partially, by an external renumbering of input data [13, 2], but these methods are usually computationally expensive. In the rest of the paper, we discuss optimizing the DPO methods to tackle all these performance problems.

3. IMPROVING THE PERFORMANCE OF DPO METHODS

In some cases DPO methods may perform suboptimally, either due to loss of parallelism (many conflicting interblock writes) or to workload imbalance. In this section we propose solutions to these problems, thereby increasing the overall performance of the method. Although the main ideas described in this section could be applied to any DPO method, we take DWA-LIP as the basic method to be improved to simplify the discussion.

3.1. Solution to the loss of parallelism

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

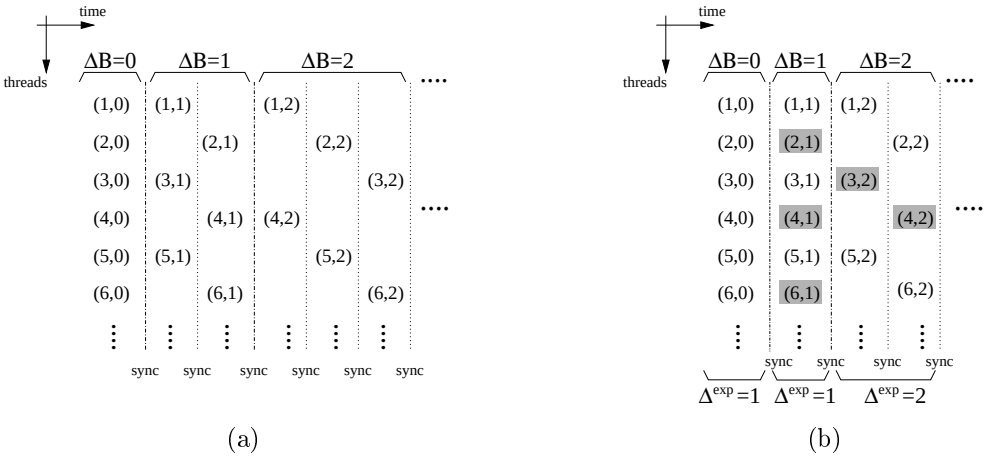


Figure 4. Model of the execution phase of DWA-LIP (a), and of the partially expanded DWA-LIP (b) with $\rho = 2$. The shadowed (i, j)–sets write in a different copy of the reduction array to that of unshadowed ones

Thus, parallelism may be increased if the reduction array is partially replicated (a fixed number of times, less than the number of threads). This improved method is called *partial array expansion*, or *partially expanded* DWA-LIP. The number of copies of the reduction array will be the *partial expansion factor* (ρ). This replication increases the parallelism exploitable by DWA-LIP because, for a particular ΔB value (that is, a column in init, Fig. 3 (b)), conflicting iteration sets may now be non-conflicting since 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 to achieve maximum benefit from this extra parallelism. Fig. 4 (a) depicts the dataflow of the execution phase of the original DWA-LIP method. Each column represents the sub-stages of non-conflicting iteration sets that can be executed in parallel. For each value of ΔB we have a total of $\Delta B + 1$ sub-stages, and in each of them non-conflicting iteration sets are executed in parallel. If the reduction array is replicated ρ times, then, for each column, ρ sets cease to be in conflict because each one may work on a different private copy. Bearing this fact in mind, it can be proved that the partially expanded method can be arranged with the same execution model as the original method but with a lower number of conflicting sub-stages. For each ΔB that number is $\Delta^{exp} = \left\lfloor \frac{\Delta B}{\rho} + 1 \right\rfloor$ (instead of $\Delta B + 1$).

There are different possibilities regarding assigning private copies of the reduction array to the new non-conflicting iterations sets. A simple one, that results in a compact code, consists in cyclically assigning each one of these sets to each private buffer, from top to bottom in the corresponding column. This execution model results in a parallelism exploitation less than ρ

from columns where $\Delta B > \frac{nThreads-1}{2}$. To avoid this loss of parallelism, for these columns the iteration sets are grouped into stages of at most ρ elements. All sets in each stage can be executed in parallel, working on different private arrays.

When $\rho = \frac{nThreads-1}{2}$ the method achieves the maximum exploitable parallelism, and although an extra amount of memory is used, the parallelism will not be improved by making ρ greater than this value (this value will be denoted as ρ_{sat}). For access patterns exhibiting high reference locality, the maximum parallelism of the method could be achieved with a value of ρ below ρ_{sat}. For these patterns, partially expanded DWA-LIP performs as well as or better than array expansion but with a much lower memory overhead. An access pattern is found in this class when the number of iterations in $(B_{min}, \Delta B)$–sets is 0 for all $\Delta B > B_l$, where $B_l < \rho_{sat}$. This condition can be easily checked on the *count* matrix of the DWA-LIP data structure.

Fig. 4 (b) depicts the new execution model, for $\rho = 2$. For the four leftmost columns, the execution model is similar to the original DWA-LIP (but assuming Δ^{exp}). In general, comparing this execution flow with that in Fig. 4 (a), we note a significant parallelism improvement.

The use of partial replication introduces a certain overhead to the basic DWA-LIP scheme due to the initialization of reduction array private copies and the final reduction of private copies into the global reduction array. As the number of private replicas is less than the number of threads, this overhead is not as considerable as in *array expansion*. Nevertheless, experimental results show that this overhead is not significant in relation to the improvement in parallelism.

The partial expansion technique can also be applied to other DPO methods, such as LocalWrite. In this case the goal is to increase the number of local iterations (that is, decreasing the number of boundary iterations) by using a number of copies, $\rho < nThreads$, of the reduction arrays. In this way we have less computation replication due to splitting the boundary iterations. To apply partial expansion to LocalWrite we can extend the definition of local iteration as follows (assuming two indirections): one iteration i writing elements $A(f1(i))$, $A(f2(i))$, – whose owners are $T_1(i)$, $T_2(i)$, respectively – is local if either $T_1(i) = T_2(i)$ (as in the original version) or $0 \leq T_2(i) - T_1(i) < \rho$. In the latter case, although references are owned by different threads, it is possible that a thread executes the complete (not split) iteration if these two references write in different private copies. Partially expanded LocalWrite, as defined, will require an initialization and a final reduction phase. However, as the number of boundary iterations is reduced, the computation replication overhead will be lower too.

3.2. Solutions to workload imbalance

Generically, methods in the DPO class are based on a uniform block partitioning of the reduction array, as data locality may be exploited in this way. However, as loop iterations are assigned to the parallel threads depending on the block they write in, this may introduce workload imbalance. In this section we present two approaches to improve workload balancing in DPO methods.

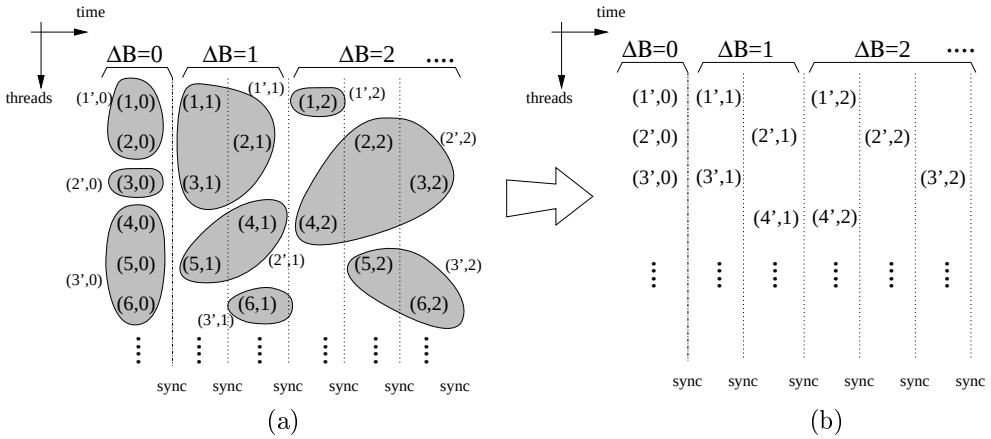


Figure 5. (a) Parallel flow computation in the DWA-LIP method, and (b) after including generic load balancing support (pairs (i, j) correspond to subblocks, while pairs (i', j) correspond to the balanced iteration supersets)

3.2.1. Generic approach

A generic approach to workload balancing could be partitioning the reduction array into contiguous blocks of different size, such that the parallel execution time is minimized. Note that changing the array distribution modifies the number of iterations in each $(B_{min}, \Delta B)$–set and so it can balance iterations for certain values of ΔB, but at the expense of increasing the number of iterations with a greater value for ΔB. This will produce loss of parallelism. This fact leads us to the problem of selecting an optimal block partition that minimizes the parallel execution time.

The minimization process has a high computational cost because it must find the partition that provides the optimal distribution of iterations into $(B_{min}, \Delta B)$–sets. To do this, parallel time must be evaluated iteratively during the optimization process for as long as different block sizes (and thus their iteration distribution) are tested. A simplification of this approach could be to partition the reduction array into small subblocks of the same size, by a multiple of the number of parallel threads. This way, blocks of different sizes may be built by grouping, in a suitable way, a certain number of contiguous subblocks.

The problem now is how to implement such an approach in a DPO method without losing its good properties. It is also desirable keeping the computational structure of the original method. We now decribe the specific case of DWA-LIP. A seamless modification of the DWA-LIP method to support the proposed generic load balancing is shown graphically in Fig. 5. 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 (see Fig. 5 (a)). 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) than the number of subblocks, then these subblocks may be grouped into balanced supersets of different size. In Fig. 5 (b) 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 of ΔB. We observe that 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 balanced DWA-LIP handles the supersets in synchronized stages in the same way as the original DWA-LIP. In order to do this, the execution proceeds in parallel stages of supersets. In 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. In consequence, in the balanced DWA-LIP, if make sure 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 $\Delta^{LB} = \left\lfloor \frac{\Delta B - 1}{r} + 1 \right\rfloor$, also issue conflict-free writes, and thus may be executed in a fully parallel manner. It can be proven that the best value that maximizes parallelism is $r = \min\left(\Delta B, \frac{nSubBlocks}{nThreads}\right)$, where $nSubBlocks$ is the number of subblocks and $nThreads$ is the number of threads. 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 in a similar way to the original one but operates on supersets.

In addition, a similar generic load balancing approach can be used with LocalWrite. In this case the subblock grouping must try to balance, independently, the execution of local and boundary iteration loops. A simple heuristic could be to independently obtain a simple histogram-based balancing for the local iterations and for each split boundary iteration.

3.2.2. Local expansion approach

There are situations in which load imbalance occurs that deserve to be considered as a special case. One of these situations 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 a histogram analysis stage to the DPO method inspector. Indeed, in the case of the DWA-LIP technique, this information is contained in the actual inspector data structure (to be precise, in the *count* matrix (Fig. 3(b))). It should be noted that the smaller the size of a contention region the fewer the number of threads that can cooperate in the execution of the high number of iterations writing in such regions (and thus, generating imbalance). An easy way to solve this problem consists in replicating the blocks in the contention region among all the threads. This way, write conflicts in that region are removed and thus the iterations can be redistributed on a greater number of threads.

With this approach, the DPO method continues to exploit data locality without requiring a large amount of extra memory, as in an LPO method, such as array expansion or replicated buffer. Selective privatization also tries to carry out the fewest possible number of replications of the reduction array elements, but it does not consider data locality whatsoever.

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

The extra memory overhead that local replication introduces is equal to the size of the reduction array multiplied by the number of replicated blocks. If the contention region is very narrow, this latter number is much lower than the total number of blocks, and thus the total extra memory cost would be much lower than in a typical LPO method.

The LocalWrite method could also benefit from local expansion when high contention regions exist. Because the local expanded blocks can be written in parallel by several threads, the boundary iterations that write in these blocks will become local iterations. Thus, better performance would be expected.

4. EXPERIMENTAL RESULTS

We have experimentally evaluated the solutions discussed for improving the performance of DPO methods (specifically, DWA-LIP) and compared them with other parallel irregular reduction methods on an SGI Origin 2000 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).

We have parallelized three representative irregular codes and their data sets for which the basic DWA-LIP method exhibits some inefficiency, and which can be solved with the optimization techniques discussed . These three codes are: a differential equation resolution code (EULER) for the partially expanded DWA-LIP; the Legendre transform code (Spec) for the generic load balance approach; and a two-dimensional molecular dynamics code (MD2) for the local expansion approach. Pseudocodes for the reduction kernels in these applications are depicted in Figure 6.

The partially expanded DWA-LIP method has been tested on the EULER code (from the motivating application suite of HPF-2 [4]). This code solves the differential Euler equation on an irregular grid, computing some physical magnitudes (such as velocities or forces) on the nodes described by a mesh. The tested data set – that is, the description of the mesh – has a relatively low intra-iteration locality, and thus loss of parallelism appears when the basic DWA-LIP method is used. 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. The subscript arrays are read from an external file and they are never modified in the time-step loop. Therefore, the inspector phase is computed only once just after the values of subscript arrays are known. The parallel EULER kernel has been executed using a 1161K nodes mesh with a connectivity of 8 (ratio between edges and nodes). Thus, we have a 1161-K-sized reduction array and 8×1161K iterations in the reduction loop. In order to compare the performance impact of the inter-iteration locality, two versions of the mesh have been used. One mesh was obtained after applying a coloring algorithm to the edges. In this way the content of

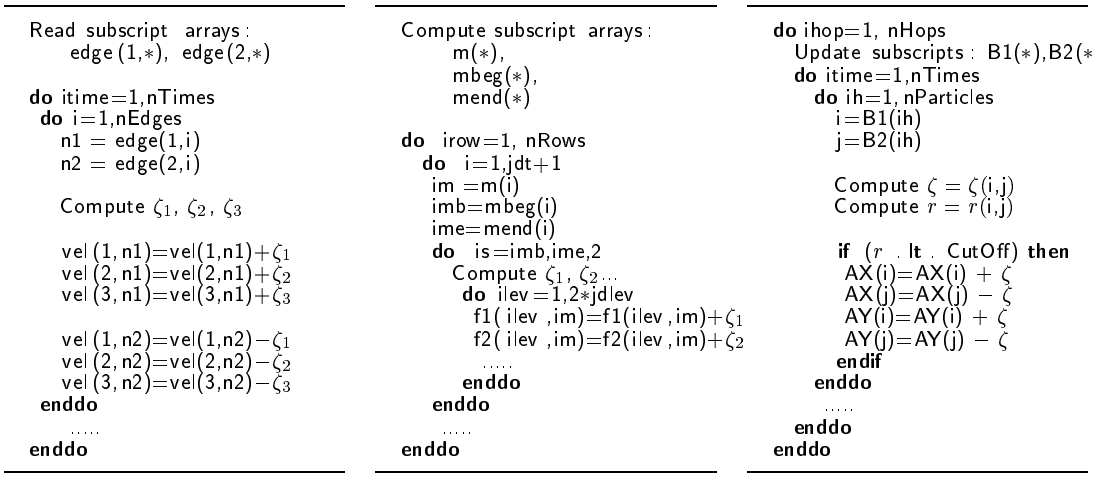


Figure 6. Pseudocodes for the reduction kernels tested: EULER (a); Legendre transform (b); and 2D Molecular dynamics (c).

subscripted arrays is reordered in batches (colors) of values that are not repeated. Thus, a low inter-iteration locality should be expected in the reduction loop as consecutive iterations update different and distant reduction array elements. In the second mesh, the list of edges has been lexically sorted, resulting in an expected higher inter-iteration locality.

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

Due to the low inter-iteration locality we expect the array expansion in the colored mesh to behave badly. When the number of processors is larger than 8, a lower execution time is achieved with the partially expanded method using $\rho = 4$. For 16 threads and $\rho = 8$, partially expanded DWA-LIP outperforms array expansion.

In the sorted mesh case, the main limitation is the parallelism loss caused by the low intra-iteration 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 5% of the computation phase time).

As the intra-iteration locality of the tested mesh is low, the selective privatization method will replicate a high number of elements of the reduction array, due to the fact that these

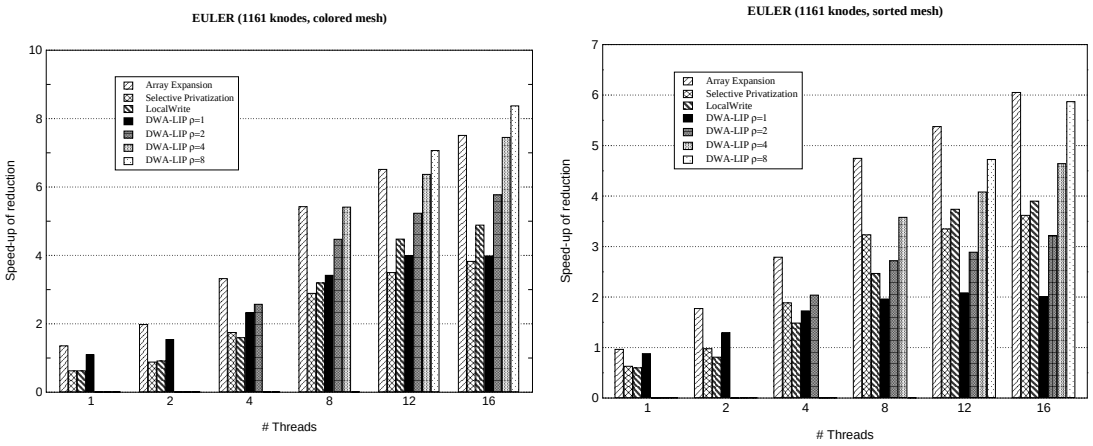


Figure 7. Speedup for the parallel EULER code using DWA-LIP, its partially expanded version (+PE), array expansion, selective privatization, and LocalWrite methods (colored mesh at the left and sorted mesh at the right)

elements will be modified by several threads, and even more for the colored mesh. This is the reason for the low performance of this method. LocalWrite suffers from loss of parallelism due to the computation replication in boundary iterations. In the tested code the reduction loop has two indirection arrays, thus half of the parallelism in the boundary iterations is lost. As the tested data has low intra-iteration locality, the number of these iterations is relatively high.

The generic load balancing approach was implemented and tested using the Spec Code [16], a kernel for Legendre transforms used in numerical weather prediction. The irregular reduction is inside a nested loop, the indices of the innermost loop also being indirections (see Fig. 6 (b)). For this reason workload imbalance is present because some outer loop iterations carry out more reduction operations than others.

Fig. 8 shows the resulting speedup for the execution phase of several reduction methods. Pure DPO methods show suboptimal performance, which is mainly due to workload imbalance. When the generic load balancing solution is introduced into DWA-LIP, the performance is significantly improved. The K factor represents the ratio between the number of reduction array subblocks and the total number of threads. By 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 imbalance. Finally, the overhead of the inspector phase is negligible (less than 1% of the reduction time), as it is executed only once.

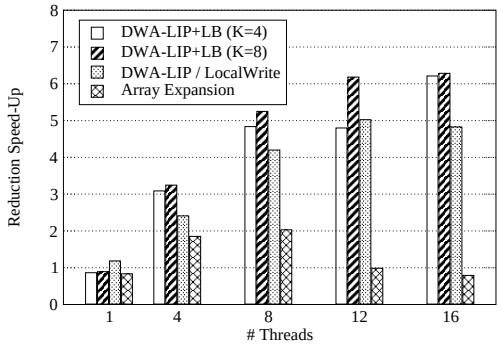


Figure 8. Speedup of the generic load-balanced DWA-LIP method (DWA-LIP+LB) compared to the original DWA-LIP, LocalWrite, and array expansion for the Legendre transformation

The local expansion load balancing approach, on the other hand, was tested on a simple 2D short-range molecular dynamics simulation [18] (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. This list stores the index of its neighbouring particles (interacting particles) for each particle. Two subscript arrays are used to implement the list (Fig. 6 (c)). As a 2D magnitude (force) is computed, two reduction arrays are present in the reduction loop. A time-step loop surrounds the reduction loop in order to compute the evolution of the particle system. In addition, the subscript array is dynamically updated every given number of time steps because the neighbours of each particle change slowly in time when particle positions are modified. In our experiments, the neighbour list was updated every 10 time steps. This fact involves the re-execution of the inspector every time the neighbour list is updated. The number of particles simulated was 640K. A high contention region in the particle domain has been introduced artificially. To test the impact of the inter-iteration locality, the iteration the order of the original loop that runs over the neighbour list was randomized.

Fig. 9 shows the speedup for the execution phase of the locally expanded load balancing technique implemented in the DWA-LIP method, compared to array expansion and selective privatization. The left part in the figure corresponds to the original code (sorted neighbour list) while the right part corresponds to the randomized code. As the inter-iteration 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, techniques such as selective privatization perform very well. DWA-LIP works badly due to the high imbalance 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 (whereas selective privatization works directly on the original reduction array most of the time). Array

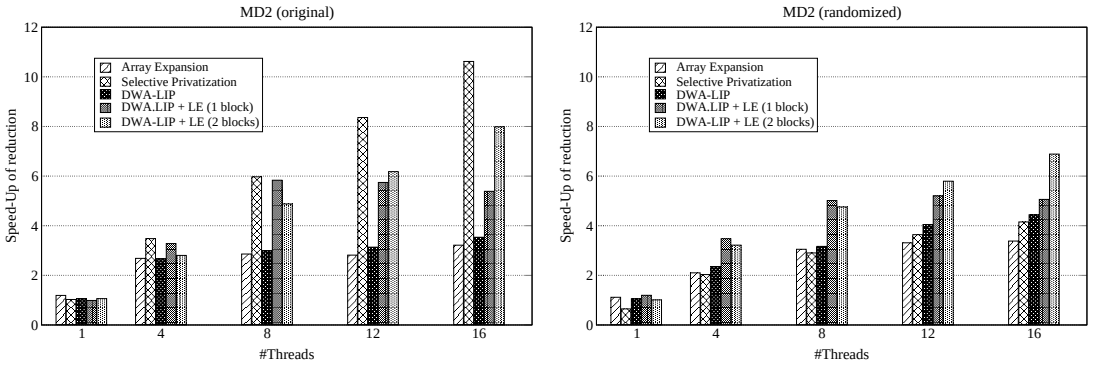


Figure 9. Speedup of the locally expanded DWA-LIP method (DWA-LIP+LE) compared to the original one and other methods for the MD2 simulation code. The left part corresponds to the original code, while in the right part the loop that runs over the neighbour list of particles was randomized

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-iteration locality is lost. This implies a strong impact on selective privatization performance as the number of conflicting elements increases drastically. However, DWA-LIP and its variants maintain their performance at similar levels, as these methods exploit inter-iteration locality at runtime. The impact of the inspector phase, in both cases, is around 1% for locally expanded DWA-LIP and 2.5% for selective privatization.

The extra memory needed by the optimized methods is another important overhead issue, as shown in Fig. 10 (EULER) and 11 (MD2). Memory overhead is measured taking the size of the reduction arrays in the sequential code as the unit. In array expansion the only overhead source is the replication of the reduction arrays in all threads, while in LocalWrite and DWA-LIP this source corresponds only to the inspector data structures. For the other methods, both overhead sources are present. In these latter cases, the lower part in the plot bars in the above figures represents the fraction of extra memory due to the first overhead source. The upper part, on the other hand, corresponds to the second source.

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 or better speedup than the quickest of the other methods. In this case, the relatively low intra-iteration locality of the input data set is the reason that selective privatization exhibits higher memory overhead than array expansion.

In the MD2 code, the impact of inter-iteration locality is stronger. In the case of the original version of MD2, selective privatization performs best because the fraction of the number of replicated elements is very low due to the good intra-iteration locality of input data. However,

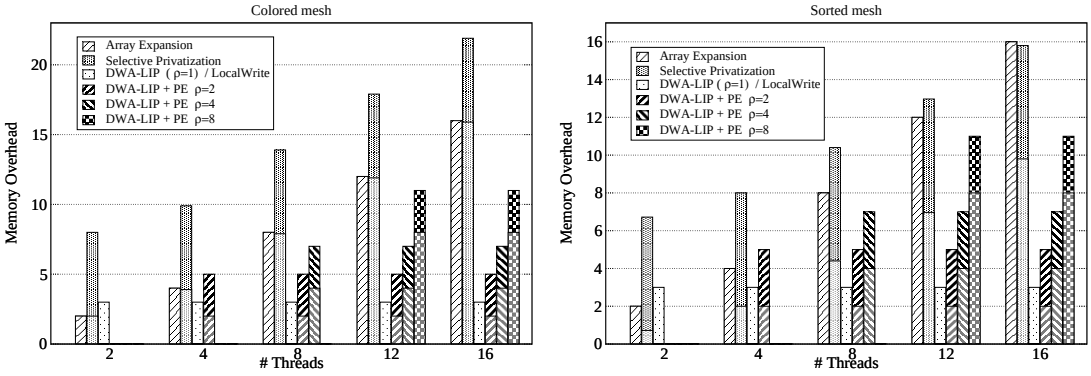


Figure 10. Memory overhead for parallel EULER code using DWA-LIP, its partially expanded version (+PE), array expansion selective privatization, and LocalWrite methods (colored mesh at the left and sorted mesh at the right)

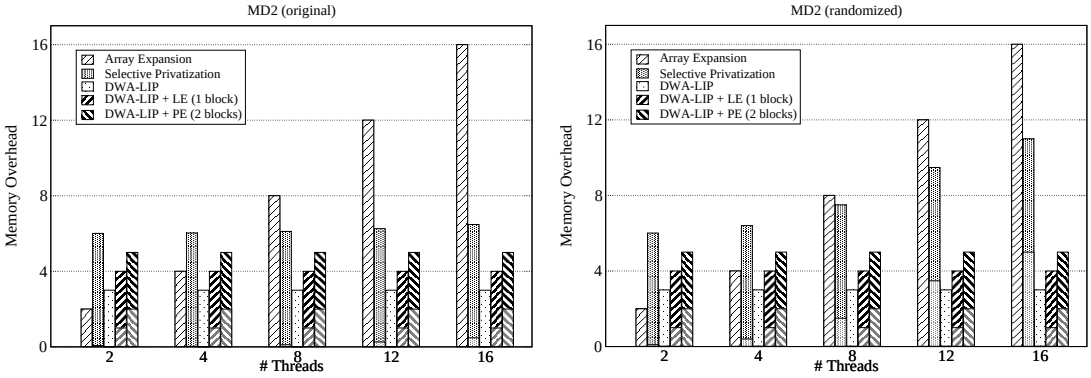


Figure 11. Memory overhead of the locally expanded DWA-LIP method (DWA-LIP+LE) compared to the original one and other methods for the MD2 simulation code. The left part corresponds to the original code, while in the right part the loop that runs over the neighbour list of particles was randomized

for the randomized version, DWA-LIP achieves better speed-up, needing less extra memory than the other methods.

5. CONCLUSIONS

In this paper we classify and analyze in terms of important performance properties methods for parallelizing irregular reductions in the context of shared memory multiprocessors. From this study, we identify possible performance problems in an important class of methods, specifically, those related to exploited parallelism and workload balancing.

Solutions to these problems are proposed, discussing specific implementations for the DWA-LIP (*Data Write Affinity with Loop Index Prefetching*) method. In its basic version, this technique can suffer from loss of parallelism and workload imbalance in some specific cases. An important fact is that the inspector phase of the method can reveal whether we are dealing with such a case. Thus, it is possible to automate the process of selecting the appropriate optimization technique. In addition, the proposed techniques introduce minor changes in the original parallel computation structure.

The experimental results allow us to conclude that it is possible to improve the performance of data partitioning-based methods with no significant loss of data locality and no substantial increment in extra memory overhead and algorithmic complexity.

REFERENCES

1. Blume W, Doallo, R, Eigenmann R, Grout J, Hoeflinger J, Lawrence T, Lee J, Padua D, Paek, Y, Pottenger B, Rauchwerger L, Tu, P. Parallel programming with Polaris. *IEEE Computer*, 1996; **29**(12):78–82.
2. Ding C, Kennedy K. Improving cache performance of dynamic applications with computation and data layout transformations. *Proceedings of the ACM International Conference on Programming Language Design and Implementation (PLDI'99)*, May 1999, Atlanta, GA, May 1999; 229–241.
3. Feautrier P. Array expansion. *Proceedings of the 2nd International Conference on Supercomputing (ICS'88)*, June 1988, St. Malo, France; 17–24.
4. Foster I, Schreiber R, Havlak P. HPF-2, scope of activities and motivating applications. *Technical Report CRPC-TR94492*, Rice University, November 1994.
5. Gutiérrez E, Plata O, Zapata EL. An automatic parallelization of irregular reductions on scalable shared memory multiprocessors. *Proceedings of the 5th International Euro-Par Conference (EuroPar'99)*, August-September 1999, Tolouse, France; 422–429.
6. Gutiérrez E, Plata O, Zapata EL. A compiler method for the parallel execution of irregular reductions in scalable shared memory multiprocessors. *Proceedings of the 14th ACM International Conference on Supercomputing (ICS'2000)*, May 2000, Santa Fe, NM; 78–87.
7. Gutiérrez E, Asenjo R, Plata O, Zapata EL. Automatic parallelization of irregular reductions. *Parallel Computing*, 2000. **26**(13–14):1709–1738.
8. Hall M, Anderson J, Amarasinghe S. Murphy B, Liao S-W, Bugnion E, Lam MS. Maximizing multiprocessor performance with the SUIF compiler. *IEEE Computer*, 1996; **29**(12):84–89.
9. Hall M, Amarasinghe S., Murphy B, Liao S-W, Lam MS. Detecting coarse-grain parallelism using an interprocedural parallelizing compiler. *Proceedings of the IEEE Supercomputing'95*, December 1995, San Diego, CA.
10. High Performance Fortran Forum. High Performance Fortran Language Specification, Version 2.0. 1996.
11. Han H, Tseng C-W. Improving compiler and run-time support for irregular reductions. *Proceedings of the 11th Workshop on Languages and Compilers for Parallel Computing (LCPC'98)*, August 1998, Chapel Hill, NC.

12. Han H, Tseng C-W. Efficient compiler and run-time support for parallel irregular reductions. *Parallel Computing*, 2000. **26**(13–14):1861–1887.
13. Han H, Tseng C-W. Improving locality for adaptive irregular scientific codes. *Proceedings of the 13th Workshop on Languages and Compilers for Parallel Computing (LCPC'00)*, August 2000, Yorktown Heights, NY.
14. Han H, Tseng C-W. A comparison of parallelization techniques for irregular reductions. *Proceedings of the 15th IEEE International Parallel and Distributed Processing Symposium (IPDPS'2001)*, April 2001, San Francisco, CA.
15. Lin Y, Padua D. On the automatic parallelization of sparse and irregular fortran programs. *Proceedings of the 4th Workshop on Languages, Compilers and Runtime Systems for Scalable Computers (LCR'98)*, May 1998, Pittsburgh, PA.
16. Mukherjee N, Gurd JR. A comparative analysis of four parallelisation schemes. *Proceedings of the 13th ACM International Conference on Supercomputing (ICS'99)*, June 1999, Rhodes, Greece; 278–285.
17. Mehrotra P, Rosendale JV, Zima H. High Performance Fortran: History, status and future. *Parallel Computing*, 1998; **71**(3–4):325–354.
18. Morales J, Toxvaerd S. The cell-neighbour table method in molecular dynamics simulations. *Computer Physics Communications* 1992; **71**:71–76.
19. OpenMP Architecture Review Board. OpenMP: A proposed industry standard API for shared memory programming. `www.openmp.org`, 1997.
20. Ponnusamy R, Saltz J, Choudhary A, Hwang S, Fox G. Runtime support and compilation methods for user-specified data distributions. *IEEE Transactions on Parallel and Distributed Systems* 1995; **6**(8):815–831.
21. Rauchwerger L, Padua D. The LRPD Test: Speculative run-time parallelization of loops with privatization and reduction parallelization. *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'95)*, June 1995, La Jolla, CA; 218–232.
22. Yu H, Rauchwerger L. Adaptive reduction parallelization techniques. *Proceedings of the 14th ACM International Conference on Supercomputing (ICS'00)*, may 2000, Santa Fe, NM; 66–77.