An Analytical Model of Locality-Based Parallel Irregular Reductions

Eladio Gutiérrez, Oscar Plata and Emilio L. Zapata

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

Abstract

This paper deals with the analysis of the parallelization of irregular reductions, a frequent operation found in many irregular applications, in the context of a shared memory model. Locality exploitation is a classical problem in computer architecture and compiler design that presently still plays a fundamental role due to the growing gap between memory and processor speeds. This work contributes a formal description of the design space of locality-based parallel irregular reductions, which is used to achieve a quantitative analysis of them. The model allows to determine behavioural aspects in the methods that may influence their performance. Several parallel compiler techniques for irregular reductions are placed and examined within this model. Experimental evaluation is also presented, under various locality conditions, and the results were compared to those derived from the model.

Key words: Irregular reductions, Parallelism, Locality, Compiler techniques

1 Introduction

For the last decades locality exploitation has been one of the main goals for increasing the performance of applications, giving rise to a wide range of software optimizations. Nowadays two locality-related trends can be observed. On the one hand, applications process larger and larger data sets, and, on the other hand, the known processor-memory gap problem is more and more significant, that is, main memory speed is improving at a slower pace than processor speed. Both trends cause application performance to be increasingly dependent on the observed memory latency. At present, most of the strategies designed to minimize the impact of the observed memory latency is focused on hiding it. Well known architectural and compiler techniques for latency hiding are multithreading, multi-core processors, data and instruction speculation, prefetching, and others. However, the processor-memory gap is

increasingly more difficult and expensive to hide. On this subject, locality exploitation plays a fundamental role.

An important class of applications for which the minimization of the negative impact of the memory latency is very difficult to accomplish is made up of irregular applications. These applications are characterized by presenting non-regular accesses to memory that are, in general, derived from the utilization of indirections in the code, such as subscripted subscripts or reference pointers.

Reduction operations represent an example of a computational structure frequently found in the core of many irregular applications. The importance of these operations in the overall performance of the application has involved much attention from the compiler development community. In many cases, most of the application computing time is spent on these irregular reduction operations. Therefore, when these applications are parallelized in order to speed them up, it is a matter of great importance optimizing such operations. This paper deals with the analysis of the parallelization of irregular reductions, considering automatic solutions that can be implemented in a parallelizing compiler. However, it is out of the scope of this work those techniques that require semantic knowledge of the application, like those based on reordering, packing, partitioning ... of the input data which used to be carried out in previous stages prior to the execution of the application [1,4,19,20,24].

Our analysis of the automatic parallelization techniques for irregular reductions is based on a shared memory model. This model has been implemented in the last years in a wide spectrum of multiprocessor architectures, symmetric or non-symmetric, with uniform or non-uniform memory accesses. The importance of these architectures and programming model has been expressed in the design of standards, like OpenMP [23], and shared memory software layers. More recently, the shared memory model continue to be valid, as the current trend of processor architectures, like multithreaded multi-core processors, appear to be an SMP of a number of virtual cores, from the viewpoint of the operating system and compiler [3].

The main contribution of this paper is a formal description of the design space of methods for shared-memory, locality-based parallelization of irregular reductions. These methods try to optimize the utilization of the memory hierarchy. The proposed model may be useful in some ways. It allows to determine behavioural aspects in the methods that may degrade their performance. Many of these aspects emerge from the exploitation of data locality via data partitioning. Examples are parallelism loss, load imbalance, computation replication, and so on. As a consequence of this evaluation, weak points of the methods can be identified and, as a result, improvements can be designed. The model, besides, establishes a framework to compare the performance behaviour of the different methods. This way, for a given locality features of

```
REAL    A(1:ADim)
INTEGER f1(1:fDim1, 1:fDim2,... ,fDim_nLoops)
INTEGER f2(1:fDim1, 1:fDim2,... ,fDim_nLoops)
        ...
INTEGER f_nInd(1:fDim1, 1:fDim2,... ,fDim_nLoops)

do i1 = 1,fDim1
   do i2 = 1,fDim2
      ...
      do i_nLoops = 1,fDim_nLoops
         Compute ξ1, ξ2, ... ξ_nInd
         A(f1(i1, i2, ... i_nLoops)) = A(f1(i1, i2, ... i_nLoops)) ⊕ ξ1
         A(f2(i1, i2, ... i_nLoops)) = A(f2(i1, i2, ... i_nLoops)) ⊕ ξ2
         ...
         A(f_nInd(i1, i2, ... i_nLoops)) = A(f_nInd(i1, i2, ... i_nLoops)) ⊕ ξ_nInd
      enddo
      ...
   enddo
enddo
```

Fig. 1. Multiple nested reduction loop with several indirections

the input data set, the model can provide the necessary information to decide which method performs better according to the performance aspects we desire to strengthen.

The next section introduces the irregular reductions together with a general discussion about performance. Next, an analytical model to evaluate locality-based parallelizations is presented, from which several compiler implementations and improvements are derived. In Section 6, a performance analysis of all these implementations is introduced, that is supported with the experimental results in Section 7. Finally, we draw some conclusions.

2 Parallel Irregular Reductions: A Performance Perspective

Reduction operations are characterized by the application of a commutative and associative operator to a set of values, in such a way that they can be grouped or reordered without changing the result. A general example of these kind of operations is shown in Fig. 1, where a reduction operator ($\oplus$) is applied to the elements of an array inside a multiply nested loop. In this piece of code, the array A() is modified several times in each iteration by, exclusively, the reduction operator, and hence it is called the reduction array. The irregular nature of this code lies in the use of multiple subscript arrays, $f_1()$, $f_2()$, ..., $f_n()$, to access to the elements of A(). We will refer to these arrays as indirection arrays. Due to the existence of these subscripted subscripts, loop-carried dependencies may be present. These indirections must be the only source of dependencies in order the whole loop to be considered a reduction.

Due to the associativity and commutativity properties, iterations of a reduction loop can be reordered and thus the loop is parallelizable provided that all possible dependencies are overcome. In the context of a shared memory model, a variety of solutions has been proposed in the literature to solve this

parallelization. We may classify these methods into three categories. The first class includes the simplest solution which consists in enclosing each access to the reduction array in a critical section. This way the reduction loop is executed fully parallel without breaking any dependence. The second class tries to minimize synchronizations by privatizing the reduction array and distributing loop iterations among parallel threads. Each thread carries out its assigned computations on a private copy of the reduction array, obtaining the final result by combining partial values across threads. Two representative examples in this class are *Replicated Buffer* [13] and *Array Expansion* [6,2]. Methods in the third class try to avoid the privatization of the reduction array by partitioning it and distributing it among threads. 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 iterations (through the reordering of the subscript arrays). The resulting reordering must avoid write conflicts in order to preserve dependencies. We can find methods in this class both for distributed-memory and shared-memory architectures. For example, in [27], a parallel irregular reduction algorithm for multithreaded architectures is proposed. However, since the focus of this work is on shared-memory implementations, methods like LocalWrite [14,15] and SynchWrite [11] will be the subject of study in this paper.

Many factors influence overall performance of the parallel irregular reduction methods. Such factors affect in different ways the various categories described previously. A first factor is synchronization overhead, that results in thread execution delays due to critical sections or barriers. This factor has an important effect on performance of the first class of methods, based on critical sections. The high cost of synchronization in typical shared memory multiprocessors makes unfeasible the use of these methods in practice. A second factor is memory overhead, associated to the extra memory required by the parallelization method. This extra memory come from either auxiliary data structures or privatized variables replicated in each thread. This last source of overhead is specially important for methods in the second class, based on the privatization of the reduction array, and limits their scalability, as memory requirements grow linearly with the number of threads. A third factor is locality exploitation that, in a reduction loop, may exist between iterations (inter–iteration locality) or inside a same iteration (intra–iteration locality). Only the third class of methods, based on the partitioning of the reduction array, takes into consideration data locality, in particular, inter–iteration locality. Nevertheless, depending on the way the reduction array is distributed, intra–iteration locality may give rise to harmful side effects in this class of methods. Among these effects we find parallelism loss, computation replication and workload imbalance, and are very related to the input data access patterns. All these effects, that will be discussed in detail in a later section, may be mitigated by means of a preprocessing reordering of the input data [4,24,16], that can have a high computational cost. As these preprocessing stages usually

	Based on mutual exclusion (**Critical sections**)	**Based on reduction array privatization** (**Array Expansion, Replicated Buffers**)	**Based on reduction array partitioning** (LocalWrite, SynchWrite)
Advantages	Simple implementation	Simple implementation High concurrency	Able to exploit locality Low memory requirements
Disadvantages	High synchronization cost and potential serialization No locality exploitation	High memory overhead Low scalability No locality exploitation	Potential parallelism loss, computation replication and workload imbalance

Table 1
Typical performance properties for parallel reduction methods

require external knowledge about data, they are out of scope of this work.

In the literature different efforts for improving the performance of the basic parallelization methods can be found. In the case of privatization-based methods, work is mainly aimed at reducing the memory overhead. This way, techniques like *Reduction Table* [18] and *Selective Privatization* [18,26] try to minimize the number of replicated elements of the reduction array, but at the cost of introducing a new source of overhead through an inspection stage. Also, heuristic adaptive solutions for deciding which method will perform best for a given input data set has been proposed [26], as well as other improved strategies based on critical sections [17]. Table 1 summarizes the discussion made in this section.

3 Evaluating Methods that Exploit Locality

This section presents a formal model to guide the analysis of irregular reduction parallelization methods that exploit inter–iteration data locality. With this aim a suitable distribution function of the reduction array onto threads must be defined: $\Psi : \{A(1), A(2), ...A(ADim)\} \rightarrow P$, where $A(1{:}ADim)$ is the reduction array and $P = \{1, 2, ...nThreads\}$ is the set of thread identifiers cooperating in the computation. At this point some definitions that link the distribution function and the iteration space of the reduction loop (each iteration is represented by $\vec{i} = (i_1, i_2, ...i_{nLoops})$) are needed.

Definition 1 The *write access set* of the iteration $\vec{i}$ is defined as the set of subscripts for which A is written in such iteration, that is, $Acc_{\vec{i}}(A) = \{m \in [1, ADim] \mid A(m)$ is written in iteration $\vec{i}\}$.

Definition 2 Two iterations, $\vec{i}$ and $\vec{j}$, are *write affine* if their write access sets are mapped to the same subset of threads, that is, $\Psi(Acc_{\vec{i}}(A)) = \Psi(Acc_{\vec{j}}(A))$.

Definition 3 Two iterations, $\vec{i}$ and $\vec{j}$, are *write dissimilar* if their write access sets are mapped to disjoint subsets of threads: $\Psi(Acc_{\vec{i}}(A)) \cap \Psi(Acc_{\vec{j}}(A)) = \emptyset$.

Lemma 1 The binary relation between iterations stated in Definition 2 satisfies the reflexive, symmetric and transitive laws, and hence is an equivalence relation, which will be named *affinity relation*.

The affinity relation allows us to divide the set of iterations (denoted by $\mathcal{S}$) into equivalence classes. Each class is made up of those iterations that write to array reduction elements mapped to the same subset of threads. That is, for each $Q \subset P$, there exists an associated *affinity equivalence class* given by $\mathcal{C}_Q = \{\vec{\imath} \in \mathcal{S} \mid \Psi(Acc_{\vec{\imath}}(A)) = Q\}$. We will denote $\mathcal{S}/\text{aff}$ the *affinity quotient set*, the set of all affinity equivalence classes.

DEFINITION 4 Two affinity equivalence classes, $\mathcal{C}_Q$ and $\mathcal{C}_R$, are defined *dissimilar* if for each pair of iterations, $\vec{\imath} \in \mathcal{C}_Q$ and $\vec{\jmath} \in \mathcal{C}_R$, are write dissimilar.

LEMMA 2 Two classes, $\mathcal{C}_Q$ and $\mathcal{C}_R$, are dissimilar if and only if $Q \cap R = \emptyset$.

From the locality viewpoint, we will exploit better the memory hierarchy if the iterations in a certain affinity class write in array reduction elements close to each other. Selecting a block distribution for Ψ lead us to this situation. From the parallelism viewpoint, we have to look for equivalence classes that update non overlapping sets of reduction array elements, as the only true data dependencies are caused by writes in the reduction array. These classes are precisely the dissimilar classes defined previously. Hence, iterations from different dissimilar classes can be executed directly in parallel. Ideally, to obtain maximum parallelism with minimum overhead, we need as many dissimilar classes as the wanted number of threads, and with similar cardinality (for a balanced execution). In order to design a method to establish such set of dissimilar classes, the following definition is introduced.

DEFINITION 5 The *dissimilarity graph*, denoted as $DG(\mathcal{S}/\text{aff}) = (N_{DG}, E_{DG})$, is defined as an undirected graph whose vertices are affinity equivalence classes ($N_{DG} = \mathcal{S}/\text{aff}$), and there exists an edge between two vertices in the graph if such vertices correspond to not dissimilar classes.

The potential data dependencies between iterations for a particular distribution function are captured in the dissimilarity graph. To extract this information a vertex-coloring algorithm can be applied. After that, vertices with the same color are not connected and thus they correspond to dissimilar classes. For example let us consider a 2-indirection reduction loop to be executed on four threads. The largest quotient set that can be given rise is, $\mathcal{S}/\text{aff} = \{\mathcal{C}_{\{1\}}, \mathcal{C}_{\{2\}}, \mathcal{C}_{\{3\}}, \mathcal{C}_{\{4\}}, \mathcal{C}_{\{1,2\}}, \mathcal{C}_{\{1,3\}}, \mathcal{C}_{\{1,4\}}, \mathcal{C}_{\{2,3\}}, \mathcal{C}_{\{2,4\}}, \mathcal{C}_{\{3,4\}}\}$. Its dissimilarity graph is shown in Fig. 3 (a). After applying a vertex-coloring algorithm we obtain gangs of dissimilar classes, represented by vertices of the same color (non connected). Hence, denoting by $VertexColoring()$ the set of gangs of classes of the same color, we have, $VertexColoring(DG(\mathcal{S}/\text{aff})) = \Big\{ \{\mathcal{C}_{\{1\}}, \mathcal{C}_{\{2\}}, \mathcal{C}_{\{3\}}, \mathcal{C}_{\{4\}}\}, \{\mathcal{C}_{\{1,2\}}, \mathcal{C}_{\{3,4\}}\}, \{\mathcal{C}_{\{1,3\}}, \mathcal{C}_{\{2,4\}}\}, \{\mathcal{C}_{\{1,4\}}, \mathcal{C}_{\{3,2\}}\} \Big\}$.

From this last result emerges the parallel execution schedule of the reduction loop. The dissimilar classes of a gang associated to a given color are executed in parallel. However, the different gangs of classes have to be executed

```
inspection phase:
        DIS(S/aff) = VertexColoring(DG(S/aff))
...
computation phase:
        for gang ∈ DIS(S/aff)
            forall C_Q ∈ gang
                Execute iterations ∈ C_Q
            end
            #pragma barrier
        end
```

Fig. 2. Affinity-based parallelization of a reduction loop

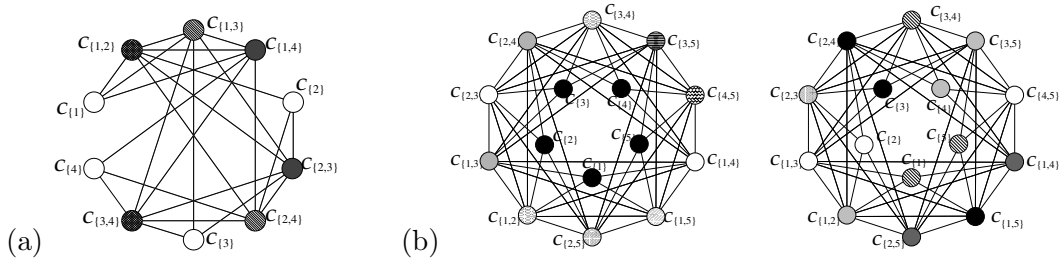


Fig. 3. For a 2-indirection reduction loop: (a) Optimum vertex coloring of dissimilarity graph for all possible non-empty classes with 4 threads; (b) different (non-optimum) colorings with 5 threads

in sequence. Therefore, in order to approach an ideal case, we should try to fulfill three conditions: minimizing the number of colors (minimum synchronization overhead), approaching the cardinality of the gangs to the number of threads (maximum concurrency), and having a similar size for the classes in the same gang (workload balancing). This execution schedule may be implemented following an inspector/executor paradigm, as shown in Fig. 2, where the equivalence classes and their coloring are carried out by the inspector.

In practice, the inspector phase of this approach suffers from important limitations. Minimizing the number of colors involves the calculation of the vertex chromatic index of the dissimilarity graph ($Chrom(DG(\mathcal{S}/\text{aff}))$) that is known as a NP-hard problem, although different near-optimal colorings may be obtained with heuristics of polynomial complexity [9]. Actually, the process is not so easy because such heuristics may provide different solutions depending on some input parameters. The selection of the best solution (in terms of execution time) is difficult due to the workload balancing condition. An example of this situation is depicted in Fig. 3(b) where the same graph with two different colorings are shown. Determining which is the best depends on the cardinality of the classes. Other problem of the inspector is its low scalability, as the maximum number of vertices of the dissimilarity graph is $\sum_{p=1}^{nInd} \binom{nThreads}{p}$. It grows very quickly with the number of threads ($nThreads$) and indirections ($nInd$). This size has effect in the coloring time and memory requirements. Finally, topological properties of the dissimilarity graph limit the exploitable parallelism. Note that, in the worst case, there always exist a totally-connected subgraph of $nThreads$ vertices embedded in the dissimilarity graph. As the

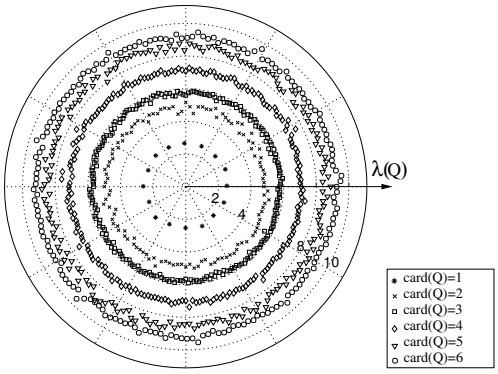


Fig. 4. Relative latency factor, $\lambda(Q)$, for affinity classes $\mathcal{C}_Q$, as a function of Q, which takes into account the effect of the memory hierarchy

vertex chromatic index of a complete graph is equals to its number of vertices, then $Chrom(DG(\mathcal{S}/\text{aff})) \geq nThreads$. This introduces a lower bound to the number of barriers in the method. It is significant that for a number of threads power of two and for a 2-indirection loop an optimum coloring can be found with $Chrom(DG(\mathcal{S}/\text{aff})) = nThreads$ (see an example in Fig. 3(a)).

From the pseudocode in Fig. 2 we can derive the following expression for the parallel execution time of the computation phase:

$$T^{PAR} = T^{SEQ}_{iter} \sum_{gang \in VertexColoring(DG(\mathcal{S}/\text{aff}))} \max_{\mathcal{C}_Q \in gang} \left\{ Card(\mathcal{C}_Q)\lambda(Q) \right\}. \quad (1)$$

Here T^{SEQ}_{iter} represents the mean execution time per iteration of the original (sequential) reduction loop without the effect of the memory hierarchy, that is, the mean effective computational payload per iteration. The effect of the memory hierarchy is considered apart by means of the factor $\lambda(Q)$, that represents the average memory access latency per iteration for a given affinity class. This factor takes into account the exploited intra–iteration locality. This locality generally depends on the relative positions of the reduction array elements that a given iteration writes in. For this reason, we have established λ as a function of the set Q that defines the affinity class. If the locality influence was negligible then λ would be 1. In order to achieve a good parallel execution time, the coloring must not only pursue a minimum number of colors but it must also try to balance gangs of classes and build gangs with classes of similar $\lambda(Q)$. As we discussed, these objectives are hard to achieve in practice.

This relative latency factor per iteration, $\lambda(Q)$, could be determined experimentally. By way of illustration, a measurement of this parameter is shown in the polar plot in Fig. 4, for the platform described in section 7 and over different subsets Q. For Q with higher cardinality, only a sampling has been considered, due to the large number of possible subsets (remember that this number grows according to the combinatorial expression $\binom{nThreads}{card(Q)}$). This latency factor is a relative measurement that takes as reference a base affinity

class for which all iterations write into a reduction array that fits completely in the memory level closest to the processor (L1 cache). The experiment was conducted for $nThreads = 16$ and $nInd = 6$, with a reduction array of $5 \cdot 10^6$ coefficients and for a reduction loop where the reduction operations are predominant (which is a very frequent case in real codes). We can observe that this latency factor depends on the number of blocks written by a given iteration, that is, on the $Card(Q)$ defining the affinity class $\mathcal{C}_Q$. Note also that this factor saturates from certain value of Q. In general, for more scattered accesses, a higher latency factor is expected.

4 Compiler Implementations

In order to find a practical implementation of the write affinity parallelization, some simplifications need to be introduced. Mainly the computational cost and memory overhead of the inspector must be lightened. For this purpose two approaches may be followed. The first approach is based on transforming the whole reduction loop into a set of one-indirection reduction loops. For these loops, the affinity classes are of the form $\mathcal{C}_Q$ with $Card(Q) = 1$, being thus $\mathcal{S}/\text{aff} = \{\mathcal{C}_{\{1\}}, \mathcal{C}_{\{2\}}, ... \mathcal{C}_{\{nThreads\}}\}$. All these classes are dissimilar and thus can be executed fully parallel. The second approach is based on the simplification of the affinity relation in such a way that inspector costs are reduced whereas the amount of parallelism is kept high. Essentially, both approaches involves a reordering of the original iterations increasing the exploitation of intra–iteration locality. Having in mind locality exploitation and according to previous discussions, a block distribution function ($\Psi^{BLOCK}(k) = \left\lfloor \frac{(k-1)nThreads}{ADim} \right\rfloor + 1$) is considered from now on.

The first approach corresponds to splitting the reduction loop. Associative and commutative properties of reduction operations allow the safe splitting of a multiple-indirection loop, like the one of Fig. 1, into $nInd$ one-indirection loops. A full loop splitting means computation replication, increasing significantly the execution time. However, it is feasible to reduce the impact of this replication by using solutions such as LocalWrite [14,15]. This solution divides the reduction loop iterations into two spaces according to the *owner–computes rule*. The first space includes all the iterations that reference reduction array elements mapped to the same thread by Ψ^{BLOCK} (local iterations). The second space contains the remainder iterations (boundary iterations). All classes $\mathcal{C}_Q$ for the local iterations have $Card(Q) = 1$, and thus, they are dissimilar. In turn, only the boundary iterations are split. Hence, LocalWrite replicates computations for only those terms involved in the boundary iterations, although the associated overhead would be important if the number of boundaries is high. A pseudocode of LocalWrite for a two-indirection reduction loop is shown in Fig. 5(a), where $\mathcal{S}^{\mathcal{LOCAL}}$ represents the local iteration space and $\mathcal{S}_1^{\mathcal{BOUND}}$, $\mathcal{S}_2^{\mathcal{BOUND}}$ the two boundary iteration spaces after splitting. $\mathcal{S}(k)$ denotes the affinity class $\mathcal{C}_{\{k\}}$ of an iteration space $\mathcal{S}$.

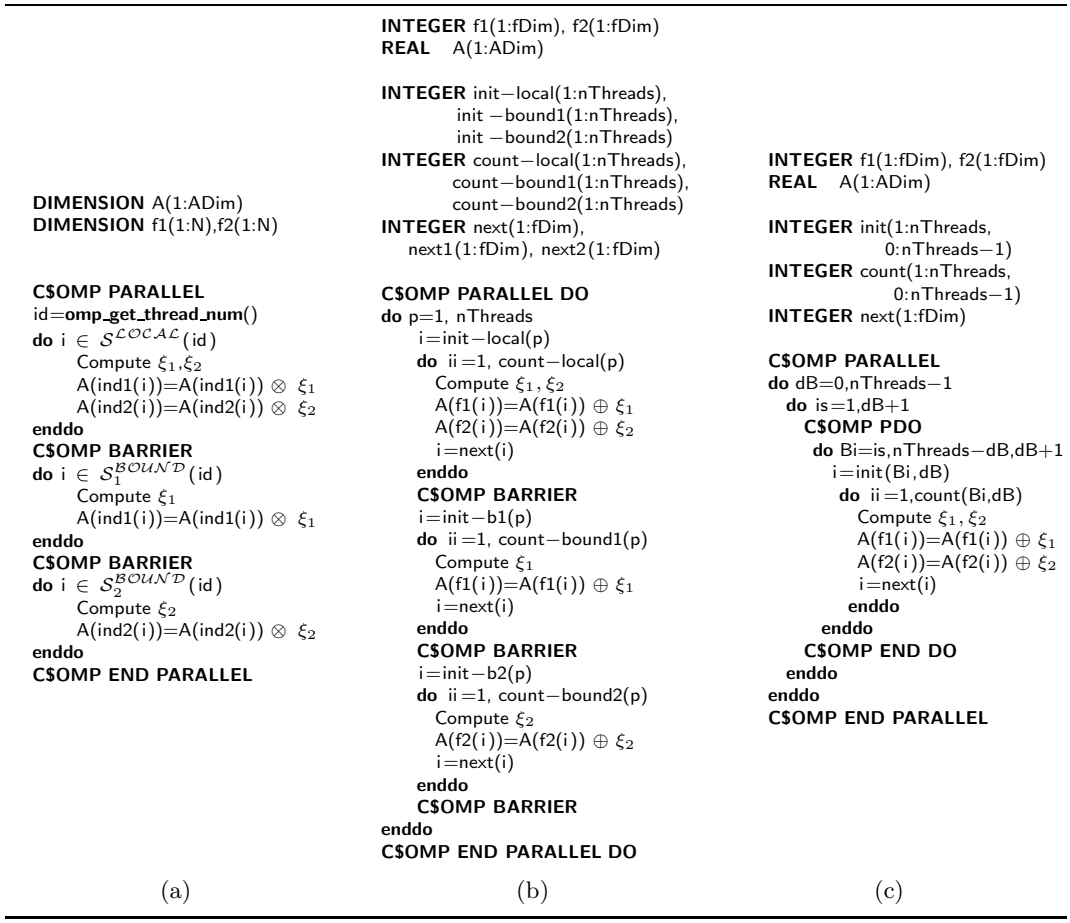


Fig. 5. Computation phase for the parallelization of a two-indirection reduction loop: (a) LocalWrite, (b) DWA-LIP, (c) SynchWrite

An implementation of the LocalWrite solution, called Data Write Affinity with Loop Index Prefetching (DWA-LIP) [10], is shown in Fig. 5(b). Inspection phase (LIP) builds loop-index prefetching arrays, containing information about iteration spaces. In this figure, three loop-index prefetching arrays are defined for each iteration space: init(), count() and next(). The dimension of the first two arrays is $nThreads$, while the last array has at most the same size as the subscript array f(). These three arrays are used to implement linked lists that represent the affinity classes (see Fig. 6). In the case of two indirections three linked lists are needed, one for the local iteration space and two for each split boundary spaces. Each thread has its associated entry in arrays init() and count(). Each entry in init() points to the first entry in next() that corresponds to the first iteration in the affinity class to be executed by such a thread. In turn, each entry in next() points to the next iteration in the same affinity class, belonging to such a thread. The array count() contains for each thread the cardinality of the corresponding affinity class for a given iteration space.

The second approach, based on simplifying the affinity relation, gives rise to the SynchWrite solution [11]. The new affinity relation is defined as follows,

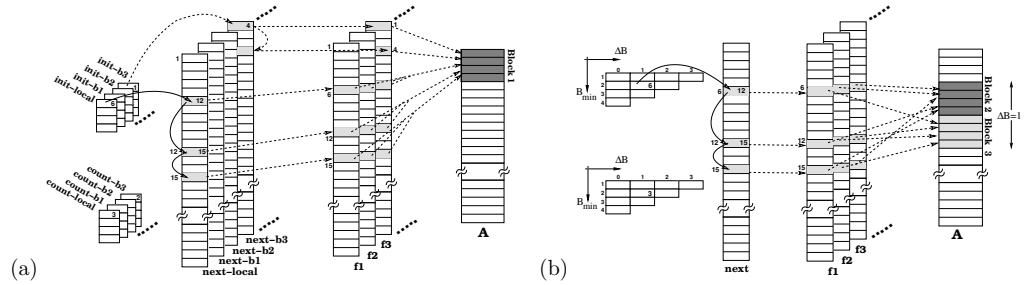


Fig. 6. Inspector data structure for (a) LocalWrite (implemented as DWA-LIP) and (b) SynchWrite solutions

Definition 6 Two iterations, $\vec{i}$ and $\vec{j}$, are *write affine* if $(B_{min}(\vec{i}), \Delta B(\vec{i})) = (B_{min}(\vec{j}), \Delta B(\vec{j}))$, being $B_{min}(\vec{i}) = min(\Psi^{BLOCK}(Acc_{(\vec{i})}(A)))$ and $\Delta B(\vec{i}) = max(\Psi^{BLOCK}(Acc_{(\vec{i})}(A))) - min(\Psi^{BLOCK}(Acc_{(\vec{i})}(A)))$.

This restricted affinity relation is defined from the vector $(B_{min}, \Delta B)$. The first term, B_{min}, corresponds to the first block written by a given iteration. The second term, ΔB, is the span between the first and the last block written by such iteration. This vector summarizes the access set $Acc_{(\vec{i})}(A)$, informing about the area of the reduction array accessed. Therefore iterations with the same pair $(B_{min}, \Delta B)$ are expected to have similar locality features.

This new affinity relation is also an equivalence relation and thus equivalence classes can be defined. Compared with the general method described in section 3, the new restricted affinity classes can be computed more easily. We denote $\mathcal{C}_{(B_{min},\Delta B)}$ the equivalence class associated to a pair $(B_{min}, \Delta B)$. Observe that two affine iterations according to Def. 2 are affine according Def. 6, while the opposite may not be true. On the other hand, the dissimilarity between classes can be also re-formulated as follows,

Lemma 3 It is a sufficient condition for two iterations $\vec{i}$, $\vec{j}$ to be dissimilar according to Def. 6, that $B_{min}(\vec{i}) + \Delta B(\vec{i}) < B_{min}(\vec{j})$ where $1 \leq B_{min}(\vec{i}) \leq B_{min}(\vec{j}) \leq nThreads$.

Lemma 3 expresses the dissimilarity between iterations that write in non-overlapped areas of the reduction array. Note that the proposed restriction of the affinity relation allows to check the dissimilarity property with a simple condition. This fact results in a simpler parallel execution scheme. The inspector of SynchWrite builds the classes $\mathcal{C}_{(B_{min},\Delta B)}$. Similarly to the DWA-LIP technique this stage can be implemented by using some prefetching arrays (see Fig. 5(b)). A linked list defined by the arrays init, count and next represents a given class $\mathcal{C}_{(B_{min},\Delta B)}$. Unlike the DWA-LIP method, only one array next is needed independently on the number of indirections (see Fig. 6(b)).

The SynchWrite computation phase is shown in Fig. 5(c). It runs over all affinity classes executing in parallel those that are dissimilar, according to lemma 3. Locality is exploited by traversing the classes having similar local-

ity features. The outermost loop in Fig. 5(c) runs over all values of ΔB. For each ΔB, a gang of dissimilar classes are executed in parallel, with a barrier synchronization between gangs. For a given ΔB, such gangs are of the form $\mathcal{C}_{(is,\Delta B)}, \mathcal{C}_{(is+(\Delta B+1),\Delta B)}, \mathcal{C}_{(is+2(\Delta B+1),\Delta B)}, ...$, with at most $\Delta B+1$ gangs of dissimilar classes. The innermost loop in Fig. 5(c) traverses the linked lists of the SynchWrite data structure. The body of this loop just includes the computational payload of the original loop.

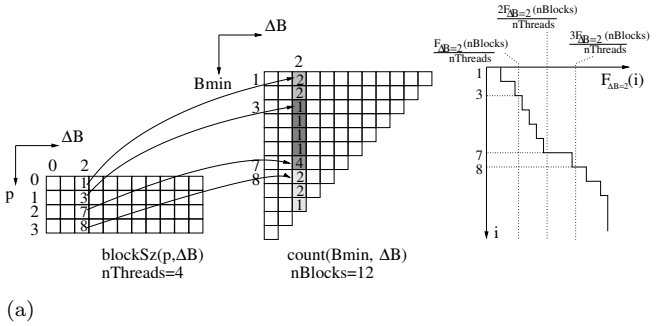
5 Implementation Improvements

Affinity-based parallelization solutions are sensitive to reference patterns exhibited by the input data set, that may introduce performance inefficiencies in the execution. In this section we analyze some improvements to solve three of these cases: workload balancing, low intra–iteration locality and high contention imbalance. Although the following proposals could be applied generally the study is focused on SynchWrite [12].

5.1 Computational workload balancing

The affinity-based methods based on a uniform block data distribution may exhibit workload imbalance. In order to minimize the impact on the original inspector, we propose to define a variable-size block partitioning, where each block is made up of a composition of small contiguous subblocks. This can be seen as a finer discrete block partition of the reduction array in $nBlocks$ subblocks ($nBlocks > nThreads$). We will apply this procedure for each execution gang (each value of ΔB), by grouping together affinity classes $\mathcal{C}_{(B_{min},\Delta B)}$ that write in contiguous subblocks. The aim is a similar cardinality for such groups. The inspector needs to be slightly changed in order to support this modification. We can compute the statistical distribution function for each ΔB from this matrix *count*. This statistical function informs us about how affinity classes must be grouped. The inspector is thus modified as follows:

(1) SynchWrite data structure is built, as in section 4, but with the distribution function $\Psi(k) = \left\lfloor \frac{(k-1)nBlocks}{ADim} \right\rfloor + 1$, using $nBlocks > nThreads$.
(2) For each ΔB, the statistical distribution function is computed as $F_{\Delta B}(i) = \sum_{Bmin=1}^{i} count(Bmin, \Delta B)$, with $\Delta B, i \in \{1, 2, ...nBlocks\}$. This function, $F_{\Delta B}(i)$, is monotonically increasing with a maximum in $F_{\Delta B}(nBlocks)$.
(3) For each ΔB, affinity classes $\mathcal{C}_{(B_{min},\Delta B)}$ writing in adjacent subblocks are grouped, following the function$F_{\Delta B}(i)$. Workload balancing is achieved by locating the points where $F_{\Delta B}(i)$ is a multiple of $\frac{F_{\Delta B}(nBlocks)}{nThreads}$. We need a new array $blockSz()$, in such a way that $blockSz(p, \Delta B)$ stores the starting index $p \in \{1, 2, ..., nThreads - 1\}$ for the p-th balanced group of classes. For operating reasons, we define $blockSz(0, \Delta B) = 1$ and $blockSz(nThreads, \Delta B) = nBlocks + 1$.



(a)

```
INTEGER f1(1:fDim), f2(1:fDim), ...
REAL    A(1:ADim)

INTEGER init(1:nBlocks,0:nBlocks−1)
INTEGER count(1:nBlocks,0:nBlocks−1)
INTEGER next(1:fDim)
INTEGER blockSz(0:nThreads−1,0:nBlocks−1)

C$OMP PARALLEL
do ΔB=0,nBlocks−1
   Δ^LB=ceil(ΔB∗nThreads/nBlocks)
   do is=1,Δ^LB + 1
      C$OMP PDO
      do Bi=is,nThreads,Δ^LB_ΔB + 1
         Execute C^LB_(Bi,ΔB)
      enddo
      C$OMP END DO
   enddo
enddo
C$OMP END PARALLEL
```

(b)

Fig. 7. (a) Load balancing approach using the information in matrix *count*; (b) Balanced SynchWrite computation phase.

(4) For each ΔB, the balanced group is $\mathcal{C}^{LB}_{(p,\Delta B)} = \bigcup_{blockSz(p-1,\Delta B)}^{blockSz(p,\Delta B)-1} \{\mathcal{C}_{(k,\Delta B)}\}$.

An example of this balancing procedure is shown in Fig. 7(a), where 12 sub-blocks and 4 threads are used. For $\Delta B = 2$ the statistical distribution function and the values for the array $blockSz()$ are plotted.

With the goal of keeping the computation phase similar to that of the original method, we impose that $blockSz(p, \Delta B)$ is strictly monotonically increasing in p. This condition is fulfilled by enforcing the restriction of having a minimum number (that we call r) of affinity classes in each balanced group. We can schedule the execution of the balanced groups in a similar way to the original method. For that, we say that, just like classes, two balanced group of classes are dissimilar when their iterations are dissimilar. The following lemma states a condition to determine the dissimilarity of balanced groups of classes from the information stored in $blockSz()$.

Lemma 4 A sufficient condition for two balanced groups of affinity classes, $\mathcal{C}^{LB}_{(b_1,\Delta B)}$, $\mathcal{C}^{LB}_{(b_2,\Delta B)}$, with $b_2 > b_1$, to be dissimilar is: $b_2 - b_1 > \Delta^{LB}$, being $\Delta^{LB} = \left\lfloor \frac{\Delta B - 1}{r} + 1 \right\rfloor$, and $r \geq 1$, where r is the minimum number of classes in each balanced group.

Hence, if a minimum number of classes, r, for each balanced group is guaranteed, the dissimilarity condition stated in lemma 4 is very close to that of lemma 3. Thus, the balanced computation phase can follow the same schedule as the original one, but using balanced groups and Δ^{LB} instead of ΔB. The amount of exploitable parallelism depends on the value of r chosen. It can be proven that a good trade-off value is $r_{\Delta B} = min\left(\Delta B, \frac{nBlocks}{nThreads}\right)$, which gives $\Delta^{LB}_{\Delta B} = \left\lceil \Delta B \frac{nThreads}{nBlocks} \right\rceil$. So, the computation phase of balanced SynchWrite is concisely shown in Fig. 7(b). The code resembles the original SynchWrite

of Fig. 5(c), but using Δ^{LB} to traverse gangs of dissimilar balanced groups.

5.2 Approaching low intra–iteration locality

For those memory access patterns exhibiting low intra–iteration locality, the cardinality of affinity classes $\mathcal{C}_{(B_{min},\Delta B)}$ with high ΔB is large. As ΔB is higher the number of dissimilar classes is lower, causing SynchWrite to lose performance due to an effective parallelism loss. This parallelism loss may be mitigated by replicating the reduction array. This fact allows to execute in parallel non dissimilar affinity classes, if each one works on a different reduction array copy. In order to keep the locality exploitation properties of SynchWrite and to reduce the impact in memory overhead, it is convenient to replicate the reduction array a number of times less than $nThreads$. We will name this improvement as *partially expanded* SynchWrite, in contrast to the full replication of methods like *Array Expansion*. We denote ρ (*partial expansion index*) the number of replicas of the reduction array. This basic idea gives rise to a new scheduling of reduction iterations as explained in the following lemma.

Lemma 5 Let ρ be the number of available copies of the reduction array. It is sufficient for two affinity classes $\mathcal{C}_{(b_1,\Delta B)}$, $\mathcal{C}_{(b_2,\Delta B)}$, with $b_1 < b_2$, to be executed in parallel that $b_2 - b_1$ is a multiple of Δ^{EXP}, where $\Delta^{EXP} = \left\lfloor \frac{\Delta B}{\rho} + 1 \right\rfloor$.

Lemma 5 resembles lemma 3, so the computation phase of partially expanded SynchWrite schedules affinity classes in a similar way as the original but using the new parameter Δ^{EXP} instead of $\Delta B+1$. Thus, classes can be scheduled safely in parallel in gangs of the form $\mathcal{C}_{(is,\Delta B)}$, $\mathcal{C}_{(is+\Delta^{EXP},\Delta B)}$, $\mathcal{C}_{(is+2\Delta^{EXP},\Delta B)}$, $\mathcal{C}_{(is+3\Delta^{EXP},\Delta B)}$, ..., if we assume a cyclic assignment of reduction array replicas to classes. Note that the number of classes in each gang is $\frac{nThreads-\Delta B}{\Delta^{EXP}}$, and because $\Delta^{EXP} \leq \Delta B + 1$ the amount of parallelism is thus increased.

Note however that for values of ΔB with gangs of less than ρ classes (that is, $\frac{nThreads-\Delta B}{\Delta^{EXP}} < \rho$), certain amount of parallelism is still lost because the number of array reduction replicas is larger than the number of classes in the gang. In such cases, which correspond to $\Delta B > \frac{nThreads-1}{2}$, it would be more efficient to execute gangs of exactly ρ classes, following a block based scheduling, in gangs of the form, $\mathcal{C}_{(is,\Delta B)}$, $\mathcal{C}_{(is+1,\Delta B)}$, $\mathcal{C}_{(is+2,\Delta B)}$, ..., $\mathcal{C}_{(is+(\rho-1),\Delta B)}$. For the rest of cases ($\Delta B \leq \frac{nThreads-1}{2}$) the scheduling remains as explained in the previous paragraph. With all these considerations, Fig. 8 shows the partially expanded SynchWrite computation phase. In addition to the new scheduling of classes we have to include an initialization and final reduction stages associate to the ρ reduction array replicas. As ρ increases Δ^{EXP} decreases, making gangs to have more classes. Nevertheless, the partially expanded method reaches the maximum exploitable parallelism when $\rho = \frac{nThreads-1}{2}$. This corresponds to a saturation value, ρ_{sat}, above which the amount of parallelism will not increase despite spending more memory.

5.3 Dealing with contention imbalance

There is a special situation that suffers from computational workload imbalance that deserves to be considered specifically. Such a situation takes place when high number of iterations write in small particular regions of the reduction array. These areas are known as high contention regions [26]. Although we may handle this situation using the approach in section 5.1, a specific solution would be more effective. Fig. 9(a) depicts a simple example of such a situation. The edges of the graph represent reduction iterations, whereas each vertex corresponds to one entry of the reduction array where each iteration writes in. Because every iteration carries out a reduction on one particular entry (vertex 4), this entry of the reduction array becomes a high contention region. This fact results in a matrix count, of the SynchWrite data structure, with a workload imbalance pattern, as shown with shadowed boxes in Fig. 9(a).

Our proposal to solve this problem is the replication of the blocks that take part in the contention regions. This way write conflicts in those regions disappear and, hence, some affinity classes that were non dissimilar may now be executed in parallel. We will refer to this solution as *locally expanded* SynchWrite. All the blocks selected to be replicated are expanded $nThreads$ times, resulting in a memory overhead smaller than a full privatization.

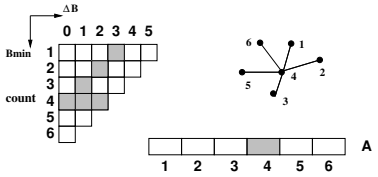
The high contention problem can be easily detected by adding to the prefetching phase of SynchWrite a histogram analysis stage. Indeed this information is stored in the actual inspector data structure (matrix count). We can define an histogram function that returns for each block Bi ($1 \leq B_i \leq nThreads$) the number of iterations included in affinity classes $\mathcal{C}_{(B_{min},\Delta B)}$ involved in such a block (that is $Bi \in [B_{min}, B_{min} + \Delta B]$). This function is computed as, $Nit(B_i) = \sum_{\Delta B=0}^{nThreads-Bi} Card(\mathcal{C}_{(B_i,\Delta B)}) + \sum_{\Delta B=1}^{Bi-1} Card(\mathcal{C}_{(Bi-\Delta B,\Delta B)})$. For instance, Fig. 9(b) shows the evaluation of this histogram function $Nit(Bi)$, normalized to the total number of iterations, for the access pattern of the sparse matrix *av41092* [7]. Note that high contention regions result in peaks in the plot (statistical modal values). These peaks determine which blocks need to be replicated, and our memory limits will give how many of these blocks are replicated. After selecting these blocks, the inspector data structure must be recomputed to take into account the dependencies that disappear due to the local replication. A pseudocode of the locally expanded SynchWrite computation phase is shown in Fig. 9, in which only the reduction array block $A(k_p : k_q)$ is locally replicated.

6 Performance Evaluation

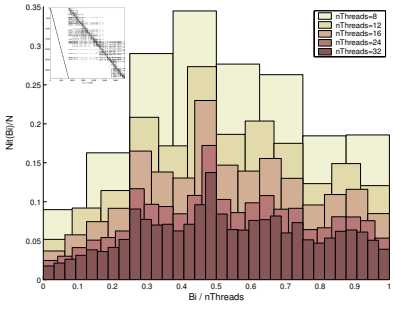
This section is devoted to present a quantitative analysis of the different issues that have influence on the performance of affinity based parallel reduction

```
INTEGER init(1:nThreads,0:nThreads−1)
INTEGER count(1:nThreads,0:nThreads−1)
INTEGER next(1:fDim)
REAL A_e(1:ADim,ρ)

C$OMP PARALLEL

A_e=init()

do ΔB=0,nThreads−1
   if (ΔB .le .((nThreads−1)/2)) then
      Δ^EXP=floor(ΔB/ρ)+1
      is_end=Δ^EXP
      is_step=1
   else
      Δ^EXP=1
      is_end=nThreads−ΔB
      is_step=ρ
   endif
   do is=1, is_end , is_step
      if (ΔB .le .((nThreads−1)/2)) then
         Bi_end=nThreads−ΔB
      else
         Bi_end=min(is+ρ−1,nThreads−ΔB)
      end
      C$OMP PDO
      do Bi=is, Bi_end , Δ^EXP
         ir=mod((Bi−is)/Δ^EXP,ρ)+1
         i=init(Bi,ΔB)
         do ii=1,count(Bi,ΔB)
            Compute ξ1, ξ2
            A_e(f1(i), ir)=A_e(f1(i), ir)⊕ξ1
            A_e(f2(i), ir)=A_e(f2(i), ir)⊕ξ2
            i=next(i)
         enddo
      enddo
      C$OMP END DO
   enddo
enddo

A=final_reduction(A_e)

C$OMP END PARALLEL
```

Fig. 8. Partially expanded SynchWrite computation phase for a 2-indirection reduction loop (omitted the initialization of expanded array (A_e) and final reduction)



(a)



(b)

```
REAL A_e(k_p:k_q, nThreads)
INTEGER init(1:nThreads,0:nThreads−1)
INTEGER count(1:nThreads,0:nThreads−1)
INTEGER next(1:fDim)

C$OMP PARALLEL
do ΔB=0,nThreads−1
   do is=1, ΔB+1
      C$OMP PDO
      do Bi=is, nThreads−ΔB, ΔB+1
         i=init(Bi,ΔB)
         do ii=1,count(Bi,ΔB)
            Compute ξ1, ξ2
            if (f1(i).le.k_q).and.(f1(i).gt.k_p) then
               A_e(f1(i),Bi)=A_e(f1(i),Bi) ⊕ ξ1
            else
               A(f1(i))=A(f1(i)) ⊕ ξ2
            endif
            if (f2(i).le.k_q).and.(f2(i).gt.k_p) then
               A_e(f2(i),Bi)=A_e(f2(i),Bi) ⊕ ξ1
            else
               A(f2(i))=A(f2(i)) ⊕ ξ2
            endif
            i=next(i)
         enddo
      enddo
      C$OMP END DO
   enddo
enddo
C$OMP END DO
C$OMP END PARALLEL
```

(c)

Fig. 9. High contention regions: (a) Simple example; (b) Histogram function for the access pattern of the sparse matrix *av41092*; (c) Locally expanded SynchWrite computation phase for a 2-indirection loop (omitted the expanded array A_e initialization and final reduction stages)

solutions. Essentially, performance is determined by the ability of the solution to exploit locality with the minimum possible effective parallelism loss.

6.1 Performance analysis of the computation phase

Expression 1 can be adapted to the practical implementations introduced in section 4. In the case of LocalWrite (DWA-LIP implementation in Fig. 5(b)), the parallel execution time can be expressed as:

$$\begin{aligned} T^{PAR} =& T_{iter}^{LOCAL} \max_{1\leq p\leq nThreads} \left\{ Card(\mathcal{C}_{\{p\}}^{LOCAL}) \right\} + \\ &+ T_{iter}^{BOUND} \sum_{idx=1}^{nInd} \max_{1\leq p\leq nThreads} \left\{ Card(\mathcal{C}_{\{p\}}^{BOUND_{idx}}) \right\}. \end{aligned} \tag{2}$$

This expression contains two terms, the first corresponding to the local iteration space (affinity classes $\mathcal{C}_Q^{LOCAL}$), and the second corresponding to the boundary iteration spaces (affinity classes $\mathcal{C}_Q^{BOUND_{idx}}$ for the idx-th indirection). Remember that for all spaces the classes are of the form $\mathcal{C}_Q$ with $Card(Q)$=1. Although the mean time per iteration of local (T_{iter}^{LOCAL}) and boundary iterations (T_{iter}^{BOUND}) may be in general different, in practice, they are all similar because most of the computational payload is common for all kind of iterations. This fact was considered in expression 2 only for the boundary iterations. This expression shows how computation replication has a negative influence on the parallel time, which increases when the number of boundary iterations or the number of indirections are larger.

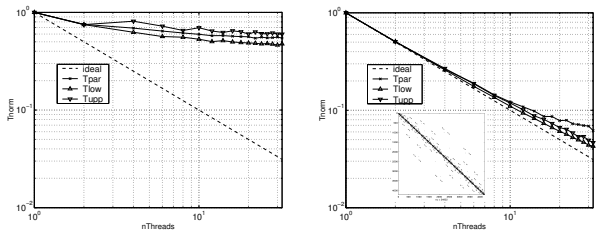
For SynchWrite, (code in Fig. 5(c)), the parallel execution time can be expressed in terms of the cardinality of affinity classes $\mathcal{C}_{(B_{min},\Delta B)}$, as follows:

$$T^{PAR} = T_{iter}^{SEQ} \sum_{\Delta B=0}^{nThreads-1} \sum_{i_g=1}^{\Delta B+1} \max_{\substack{B_{min}\leq nThreads-\Delta B \\ B_{min}=i_g+k(\Delta B+1), \\ \text{with } k\in\mathbb{N}}} \left\{ Card(\mathcal{C}_{(B_{min},\Delta B)}) \right\}. \tag{3}$$

Under conditions of perfect workload balancing, iterations are uniformly distributed across classes with the same value of ΔB.

Calling $N_{\Delta B} = \sum_{B_{min}=1}^{nThreads-\Delta B} Card(\mathcal{C}_{(B_{min},\Delta B)})$ the total number of iterations of all classes $\mathcal{C}_{(B_{min},\Delta B)}$ we can obtain from expression 3 the following bounds:

$$\begin{aligned} T_{iter}^{SEQ} \left(\sum_{\Delta B=0}^{nThreads-1} \frac{N_{\Delta B}}{\left\lceil \frac{nThreads-\Delta B}{\Delta B+1} \right\rceil} \right) \leq T^{PAR} \\ T^{PAR} \leq T_{iter}^{SEQ} \left(\sum_{\Delta B=0}^{nThreads-1} \frac{N_{\Delta B}}{\left\lceil \frac{nThreads-\Delta B-min(\Delta B+1, nThreads-\Delta B)+1}{\Delta B+1} \right\rceil} \right). \end{aligned} \tag{4}$$



(a) (b)

Fig. 10. Basic SynchWrite performance analysis: (a) dense pattern, (b) sparse pattern

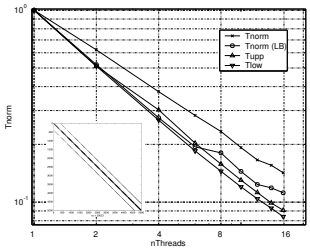


Fig. 11. Workload balanced SynchWrite performance evaluation for the access pattern of the sparse matrix *sherman3*

In expression 4 we can observe more clearly a possible parallelism loss. The denominators in the two bounds decrease when ΔB grows and, therefore, the parallelism decreases. Fig. 10 shows the normalized times corresponding to the three terms of the two inequalities in expression 4, that is, T^{PAR} and the two bounds (Tupp: upper bound, Tlow: lower bound), for two different input patterns. The dense pattern corresponds to a 2-indirection reduction loop where all possible pairs are traversed, like in an all-to-all N-body problem. The sparse pattern, on the other hand, is shown in the inset of the plot 10(b). In this case, due to the high intra–iteration locality, the parallelism loss is much lower. Observe that for a high number of threads some load imbalance occurs because exact values of T^{PAR} go out of the bounds.

In the special case of a 1-indirection reduction loop, all affinity classes are of the form $\mathcal{C}_{(B_{min},0)}$, and consequently all of them can be executed in parallel. So, there is no parallelism loss and the parallel execution time can be expressed as $T^{PAR} \approx T^{SEQ}_{iter} \frac{Nit_{\Delta B=0}}{nThreads}$.

In a similar way to expression 1 the memory hierarchy effect can also be taken into account in expression 3. As iterations in a class $\mathcal{C}_{(B_{min},\Delta B)}$ write into a larger reduction array area as ΔB grows, the expected intra-iteration locality decreases correspondingly. This effect can be included in expression 3 by introducing a latency factor $\lambda(\Delta B)$, that increases the effective time of iterations with larger ΔB:

$$T^{PAR} = T^{SEQ}_{iter} \sum_{\Delta B=0}^{nThreads-1} \sum_{i_s=1}^{\Delta B+1} \max_{\substack{B_{min} \leq nThreads-\Delta B \\ B_{min}=i_s+k(\Delta B+1), \\ \text{with } k \in \mathbb{N}}} \left\{ Card(\mathcal{C}_{(B_{min},\Delta B)}) \cdot \lambda(\Delta B) \right\}. \tag{5}$$

Following a similar approach we proceed to analyze the performance of the three improved SynchWrite solutions, introduced in section 5.

Workload balancing approach. From the balanced SynchWrite code shown in Fig. 7 we can state an expression analogous to 3 but considering balanced groups of classes instead of affinity classes and the parameter $\Delta^{LB}_{\Delta B}$ instead of ΔB:

$$T^{PAR} = T^{SEQ}_{iter} \sum_{\Delta B=0}^{nBlocks-1} \sum_{i_g=1}^{\Delta^{LB}_{\Delta B}+1} \max_{\substack{p \le nThreads, \\ p=i_g+k\left(\Delta^{LB}_{\Delta B}+1\right), \\ \text{with } k\in\mathbb{N}}} \left\{ Card(\mathcal{C}^{LB}_{(p,\Delta B)}) \right\}. \tag{6}$$

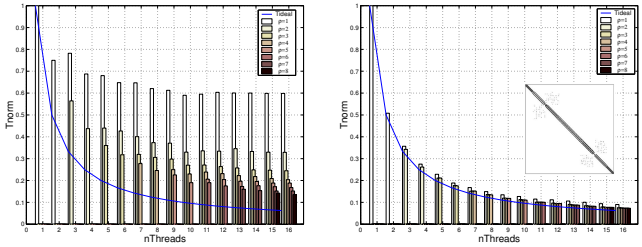
This expression has been evaluated in Fig. 11 for the pattern of the sparse matrix *sherman3* from the *Harwell-Boeing* collection [5], shown in the inset. The normalized parallel time has been plotted for the basic SynchWrite method (Tpar) together with bounds in expression 4 (Tupp, Tlow) and the workload balancing approach (Tpar(LB)). Times have been calculated for $nBlocks = 8\, nThreads$. Note that the load balancing improvement makes the parallel time to be closer to the bounds (that means better balance).

Partial expansion approach. From the code of the partially expanded SynchWrite (Fig. 8), the parallel execution time can be expressed as (without considering the initialization and global reduction):

$$T^{PAR} = T^{SEQ}_{iter} \left(\sum_{\Delta B=0}^{\left\lceil \frac{nThreads-1}{2} \right\rceil} \sum_{i_g=1}^{\Delta^{EXP}} \max_{\substack{B_{min}\le nThreads-\Delta B \\ B_{min}=i_g+k\ \Delta^{EXP}, \\ k\in\mathbb{N}}} \left\{ Card(\mathcal{C}_{(B_{min},\Delta B)}) \right\} + \right.$$
$$\left. + \sum_{\Delta B=\left\lceil \frac{nThreads-1}{2} \right\rceil+1}^{nThreads-1} \sum_{\substack{i_g=1+n\rho,\ n\in\mathbb{N} \\ i_g\le nThreads-\Delta B}} \max_{\substack{B_{min}\le nThreads-\Delta B \\ B_{min}=i_g+k, \\ k\in[1,\rho]\subset\mathbb{N}}} \left\{ Card(\mathcal{C}_{(B_{min},\Delta B)}) \right\} \right), \tag{7}$$

that, after further analysis and under perfect load balancing, it can be bounded as shown next:

$$\left(\sum_{\Delta B=0}^{\left\lceil \frac{nThreads-1}{2} \right\rceil} \frac{N_{\Delta B}}{\left\lceil \frac{nThreads-\Delta B}{\Delta^{EXP}} \right\rceil} + \sum_{\Delta B=\left\lceil \frac{nThreads-1}{2} \right\rceil+1}^{nThreads-1} \frac{N_{\Delta B}}{\min(\rho, nThreads-\Delta B)} \right) \le$$
$$\le \frac{T^{PAR}}{T^{SEQ}_{iter}} \le \left(\sum_{\Delta B=0}^{\left\lceil \frac{nThreads-1}{2} \right\rceil} \frac{N_{\Delta B}}{\left\lceil \frac{nThreads-\Delta B-\Delta^{EXP}+1}{\Delta^{EXP}} \right\rceil} + \sum_{\Delta B=\left\lceil \frac{nThreads-1}{2} \right\rceil+1}^{nThreads-1} N_{\Delta B} \right). \tag{8}$$



(a) (b)

Fig. 12. Partially expanded SynchWrite performance evaluation: (a) dense pattern, (b) *fidapm11* sparse matrix pattern

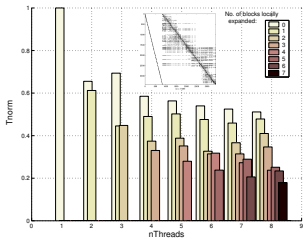


Fig. 13. Locally expanded SynchWrite performance evaluation for the sparse matrix *av41092* pattern

Expression 7 consists of two terms. The first one is associated with the classes $\mathcal{C}_{(B_{min},\Delta B)}$, with $\Delta B \leq \frac{nThreads-1}{2}$, that are executed concurrently in gangs of the form $\mathcal{C}_{(s,\Delta B)}$, $\mathcal{C}_{(s+\Delta^{EXP},\Delta B)}$, $\mathcal{C}_{(s+2\Delta^{EXP},\Delta B)}$,... The second term corresponds to the remaining classes, that are executed concurrently in gangs of the form $\mathcal{C}_{(s,\Delta B)}$, $\mathcal{C}_{(s+1,\Delta B)}$, $\mathcal{C}_{(s+2,\Delta B)}$,... As parameter $\Delta^{EXP} = \left\lfloor \frac{\Delta B}{\rho} + 1 \right\rfloor$ decreases when the partial expansion index, ρ, increases, denominators in the above inequalities are smaller and, hence, more parallelism is exploited. The first term in both bounds in expression 8 shows that the parallel execution time does not decrease for values of ρ greater than a certain threshold. This is the case when $\Delta^{EXP} = 1$ that leads to $\rho = \frac{nThreads-1}{2}$. Making ρ greater than this value increases the amount of extra memory but with no benefit of greater parallelism. This value was called ρ_{sat} in section 5.2. For access patterns exhibiting high intra–iteration locality, the maximum parallelism of the method could be reached with a value of ρ below ρ_{sat}. For this kind of patterns, partially expanded SynchWrite could perform as well as or better than full array expansion but with a much lower memory overhead.

Fig. 12 shows the evaluation of expression 7 for two different memory access patterns, 16 threads and a 2-indirection reduction loop. The normalized parallel time of the computation phase has been represented for several values of ρ. The plot (a) in Fig. 12 displays the evaluation of the above equation for an all-to-all dense memory access pattern. Plot (b) shows the results for the pattern defined by the sparse matrix *fidapm11* [25]. For the dense pattern case all affinity classes have the same cardinality, causing a poor parallelism exploitation for the basic SynchWrite. For this reason, as ρ increases the parallelism loss is reduced, obtaining a much faster parallel code, as noted in the Fig. 12(a). Nevertheless, in the case of the sparse pattern, better intra–iteration locality exists. Hence a larger amount of parallelism is exploited by using a smaller partial expansion index, as observed in Fig. 12 (b). In fact, for a value as small as $\rho = 5$ ($\rho_{sat} = 8$), parallel expanded SynchWrite reaches maximum parallelism.

Local expansion approach. The performance analysis of the locally expanded SynchWrite, (section 5.3), is analogous to that of the basic version. In this way, the parallel time can be evaluated as in expression 3 adding some small overhead due to the initialization and final reduction stages of local private buffers. Nevertheless the time of the parallel computation phase is expected to be smaller because local expansion allows to avoid write conflicts for a certain set of iterations. This fact leads to the promotion of these iterations from their native affinity classes towards classes $\mathcal{C}_{(B_{min},\Delta B)}$ with lower ΔB. Consequently the parallel execution time will decrease according to expression 3. For instance, Fig. 13 displays the evaluation of the normalized parallel time for the access pattern of sparse matrix *av41092* [7]. We observe a decrease in the execution time when more blocks are locality replicated.

6.2 Inspector overhead and memory requirements

Other relevant aspects of the performance analysis are the overhead caused by the inspection phase and the memory requirements for auxiliary data structures. Both aspects are discussed in this subsection.

The number of times that the inspection phase is executed and the computational cost of each execution determine the inspector overhead. The inspector has to be executed every time that the indirection arrays are modified. Typically, in real codes this happens either once before the reduction loop (for instance, a static mesh) or each certain number of reduction loop executions (for instance, in dynamic codes). We denote $\eta_{c/i}$ the ratio between the time of one computation phase iteration and one inspection phase iteration, and η_{reuse} the number of times that the inspection phase is executed per each whole reduction loop execution. High values of $\eta_{c/i}$ and η_{reuse} involve low computational weight of the inspector in the global performance. Both LocalWrite and SynchWrite solutions have similar inspector overheads. Basically, for both methods, the inspector traverses all the iterations, reading only the indirection entries and determining the prefetching data structures. No payload computation is carried out, so $\eta_{c/i}$ is mainly given by the reading time of the subscripted subscripts. The time overhead of the inspector can be considered in the total time according to the expression: $T_{\text{total}} = T_{\text{computation phase}} \left(1 + \frac{1}{\eta_{c/i}\,\eta_{\text{reuse}}}\right)$. Hence, we can use the factor $\left(1 + \frac{1}{\eta_{c/i}\,\eta_{\text{reuse}}}\right)$ as a measure of the inspector overhead. Note that, for affinity based solutions, the inspection phase can be fully parallelized by privatization.

Improvements discussed in section 5 can involve a small additional computational cost to the inspector. In the case of the workload balancing approach this cost is due to the building of the balanced groups, and it is very low because only matrix `count` needs to be examined (its size is $\mathcal{O}(nBlocks^2)$). In the case of the partial expansion approach, inspection phase remains unchanged and, so, the additional cost is null. Finally, in the case of the local

	Privatization	Inspector Data Structure
Array Expansion	$\mathcal{O}(ADim \cdot nThreads)$	—
Local Write (DWA-LIP)	—	$\mathcal{O}(nInd \cdot N + 2 \cdot nInd \cdot nThreads) \approx \mathcal{O}(nInd \cdot N)$
SynchWrite	—	$\mathcal{O}(N + 2 \cdot nThreads^2) \approx \mathcal{O}(N)$
Load Balanced SynchWrite	—	$\mathcal{O}(N + 2 \cdot nBlocks^2 + nBlocks \cdot nThreads)$
Partially Expanded SynchWrite	$\mathcal{O}(\rho \cdot ADim)$	$\mathcal{O}(N + 2 \cdot nThreads^2) \approx \mathcal{O}(N)$
Locally Expanded SynchWrite	$\mathcal{O}(nPeaks \cdot ADim)$	$\mathcal{O}(N + 2 \cdot nThreads^2) \approx \mathcal{O}(N)$

* *ADim*: size of the reduction array, *nInd*: no. of indirections, *N*: no. of iterations, *nThreads*: no. of threads, *nBlocks*: no. of subblocks in load balanced SynchWrite, ρ: partial expansion index, *nPeaks*: no. of peaks considered in histogram function for local expansion.

Table 2
Memory overhead: Components due to privatization of reduction arrays and inspection data structures

expansion approach, after the computation of matrix count, the inspector must locate high contention regions and recompute the matrix count according to this information.

Regarding memory requirements it is usual to take as reference the privatization based parallelizations. Note that an important disadvantage of such techniques is the high requirements of memory. Methods like Array Expansion replicate the reduction arrays in each thread. Consequently the memory overhead is of the order of $\mathcal{O}(nThreads \cdot ADim)$. Although some optimizations has been proposed [18,26], in the worst case they spend as much storage space as the full privatization.

Table 2 summarizes the memory overhead of different affinity based solutions. The main source of memory overhead comes from the inspector data structures, although certain degree of privatization has been added for some of the improved solutions. For LocalWrite, implemented as DWA-LIP, the first term in the memory overhead expression corresponds to arrays next for local and boundary iterations (as many as possible indirections). The second term corresponds to the size of matrices count and init for the worst case where all the iterations are boundaries. Typically, we can assume that $N \gg nThreads$ that leads to the approximation shown in the table. The SynchWrite memory requirements are potentially lower because the boundary iterations are not split and thus only one prefetching array is needed. In this case only an array *next* is needed. Improved versions of SynchWrite increase lightly these basic memory requirements. In the case of the balanced version, the array *next* keeps unmodified, but the arrays *init*, *count* are larger and the new array $blockSz$ is introduced. The ratio $nBlocks/nThreads$ represents the discretization granularity of balancing. Increasing $nBlocks$ a better balancing may result but at expense of an increase in memory overhead. Hence, a trade-off between time and space should be found. For the partially expanded SynchWrite the additional memory is due to the private copies of the reduction arrays. Unlike full array expansion the additional memory can be bound by selecting ρ. A similar additional memory is required for the locally expanded SynchWrite.

	#Edges (2 indirections)		#Faces (3 & 4 indirections)	
Connectivity:	8	18	8	18
#Nodes: 800K	7038K	13900K	740K	1463K
1000K	8836K	14490K	892K	1766K

Table 3
Features of the meshes tested for the Euler code.

In this case, this extra memory depends on the number of blocks expanded, which are given by the number of peaks in the histogram distribution function.

7 Results

We have experimentally evaluated the discussed solutions for locality-based parallelization of irregular reduction loops. In this evaluation we compared such solutions with other methods and the theoretical results derived from our model. The target computing platform was a SGI Origin3000 ccNUMA multiprocessor, with 400-MHz R12000 processors (8 MB L2 cache) and 2 GB main memory per node. All parallel codes were implemented in Fortran 77 with OpenMP directives, and compiled using the SGI MIPSpro.

A first batch of tests has considered only the basic versions of the techniques, and has used as benchmark the kernel of the EULER code [8]. In this kernel, several reduction loops appear inside an outer time-step loop. The reductions are carried out on three-dimensional arrays. One of them traverses the edges of an input mesh resulting in a two-indirection reduction loop. Other two loops visit the faces of the input mesh involving three and four indirections, respectively. In order to analyze the effect of different locality sources several input meshes have been tested. The intra–iteration locality has been taken into account by generating meshes with different connectivity (ratio between edges and nodes). Two values have been chosen for this parameter: 8 and 18. Note that the number of indirections may have also influence on the intra–iteration locality, as a bigger number of indirections will decrease the probability of issuing nearby references inside the same iteration. The effect of inter–iteration locality is captured by reordering the edges and faces in the input mesh. With this purpose, two versions of the mesh have been generated. A lower inter–iteration locality version is synthesized by applying a coloring algorithm to the edges and faces. A higher inter–iteration locality version is obtained by sorting lexicographically the list of edges and faces. All these variety of tested meshes are summarized in table 3. In table 4 the sequential execution times for the different tested meshes are shown. Observe how the performance depends on the locality features of the input data, taking more time those executions whose input exhibits lower intra– or inter–iteration locality.

In Figs. 14, 15 and 16 speedups for the parallel computation phase of different reduction techniques, number of indirections and input data sets are shown.

	Sorted version						Colored version					
# Indirections:	2		3		4		2		3		4	
Connectivity:	8	18	8	18	8	18	8	18	8	18	8	18
Mesh size: 800Knodes	37.80	76.45	9.40	17.25	7.95	16.10	157.60	308.40	16.75	36.05	11.95	28.35
1000Knodes	53.10	86.20	11.25	21.15	10.40	19.50	216.75	342.40	18.45	68.15	13.95	58.20

Table 4
Sequential execution time (sec.) for the Euler reduction loops (20 timesteps)

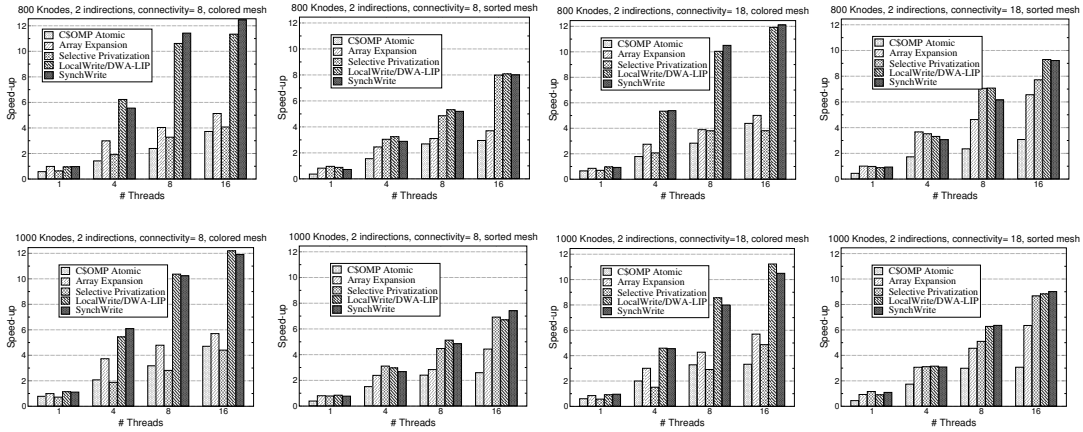


Fig. 14. Speedup for a 2-indirection parallel reduction loop in the EULER code

LocalWrite (implemented as DWA-LIP), SynchWrite as well as three other techniques (critical sections, Array Expansion and Selective Privatization), that are used as references, have been tested. Critical sections have been implemented using the OpenMP clause C$OMP Atomic. Whereas Array Expansion performs a full replication of the reduction arrays, Selective Privatization tries to save memory by replicating only those entries of the reduction arrays written by several threads. Experiments in Figs. 14,15 and 16 show the ability of LocalWrite and SynchWrite to exploit data locality. For this reason these techniques work better for those cases where the inter–iteration locality is low (colored meshes). On the other hand, a low intra–iteration locality has a negative effect over all the techniques (high connectivity, high number of indirections). Moreover, a high number of indirections reduces the performance of LocalWrite due to a higher replication of computations. Also we can observe that Selective Privatization outperforms Array Expansion for cases with high locality (good reordering of input data and low connectivity or number of indirections) because there is a low interference in accessing privatized portions of the reduction arrays.

Table 5 summarizes the speedup values predicted by the analytical model for locality-based methods. Such values are close to the speedups observed experimentally, mainly for 2 and 3-indirection loops. In the case of the 4-indirection loop the high cost of the synchronization makes the model to behave a little worse. Even so, the measured values follow relatively the theoretical trends. Also we observe that the computation replication causes the efficiency of LocalWrite to decrease for a high number of indirections with regards to SynchWrite.

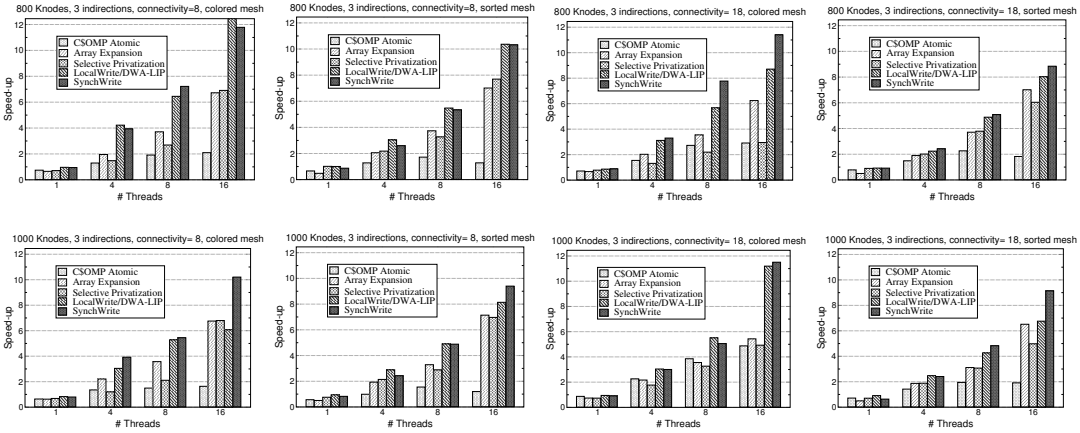


Fig. 15. Speedup for a 3-indirection parallel reduction loop in the EULER code

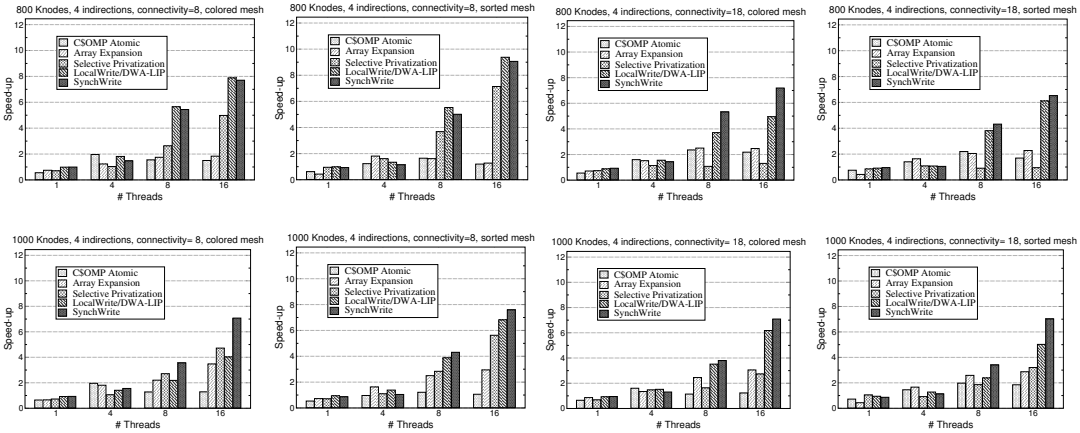


Fig. 16. Speedup for a 4-indirection parallel reduction loop in the EULER code

Table 6 shows the inspector overhead for LocalWrite and SynchWrite. Both implementations exhibit a similar inspector overhead as it depends basically on the number of times the inspector is executed and the reduction loop count (data structures are similar). The EULER code is static as the indirections remain unchanged during the whole computation. Hence, the inspector is executed once, so as its cost is spread over the total number of iterations (table 6 considers 20 time-steps). Although the inspector overhead grows with the number of indirections, note that for the colored mesh the executor is relatively slower and consequently the ratio $\eta_{\mathrm{c/i}}$ decreases. Remember that neither critical section technique nor Array Expansion need an inspection phase. Nevertheless the Selective Privatization inspector overhead can be between two and three times larger than LocalWrite and SynchWrite inspectors.

The rest of the section is devoted to evaluate the improved versions of the locality-based solutions. First the partially expanded SynchWrite is considered. For its evaluation, only the 2-indirection reduction loops of the EULER code have been taken into account, and an input mesh with low intra–iteration locality has been built by means of a randomized renumbering of its nodes. The input mesh was of 1000 Knodes with a connectivity of 8. Likewise both

	LocalWrite (as DWA-LIP)						SynchWrite					
# Indirections:	2		3		4		2		3		4	
Connectivity:	8	18	8	18	8	18	8	18	8	18	8	18
Mesh size: 800Knodes	14.6	14.0	11.1	10.4	6.7	6.2	14.6	14.0	11.9	11.5	9.8	9.5
1000Knodes	14.6	13.8	10.9	10.2	6.2	6.2	14.6	13.8	11.7	11.3	9.0	9.2

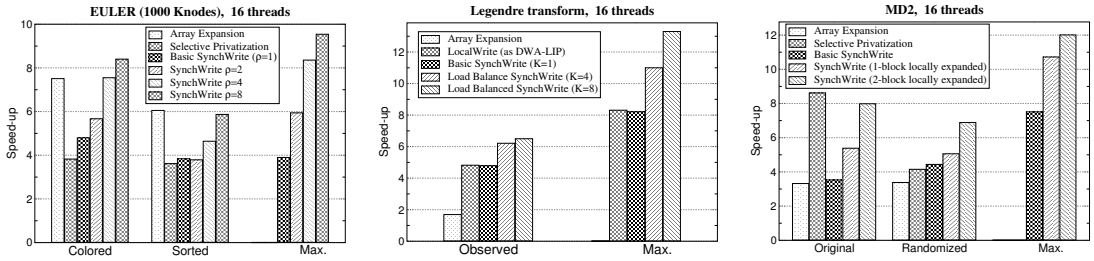
Table 5
Theoretical speedup for the Euler reduction loops according to section 6.1 considering 16 threads

	Sorted version						Colored version					
# Indirections:	2		3		4		2		3		4	
Connectivity:	8	18	8	18	8	18	8	18	8	18	8	18
Mesh size: 800Knodes	0.8%	0.9%	1.4%	1.3%	2.4%	2.3%	0.3%	0.4%	0.5%	0.4%	1.8%	1.9%
1000Knodes	0.8%	0.9%	1.5%	1.5%	2.2%	2.5%	0.3%	0.4%	0.5%	0.3%	2.1%	1.1%

Table 6
Inspector overhead factor $\frac{1}{\eta_{c/i}\eta_{reuse}}$ for the Euler reduction loops (20 time-steps)

versions with different inter–iteration locality (colored and sorted) have been used. In Fig. 17(a) the speedup of the computation phase is shown for the basic and partially expanded SynchWrite and compared to Array Expansion and Selective Privatization. Also maximum theoretical values obtained from the expressions in section 6.1 are displayed. For the colored input mesh, the basic SynchWrite shows a parallelism loss due to the low intra–iteration locality. Array Expansion takes advantage of this situation. However, a partial expansion of the reduction arrays can help SynchWrite to achieve a similar performance as Array Expansion but with a lower memory overhead. The reason is that the parallelism loss is mitigated but it takes advantage of the low inter–iteration locality of the colored mesh. This way, for $\rho = 8$ the partially expanded SynchWrite performance exceeds that of Array Expansion (with half of the memory overhead of Array Expansion). For the sorted mesh case, there is still a parallelism loss caused by the low intra–iteration locality but partially expanded SynchWrite no longer outperforms Array Expansion due to the higher inter–iteration locality. Nevertheless, for large number of threads, partially expanded SynchWrite reaches similar speedup as Array Expansion with only half of extra memory (16 threads, $\rho = 8$). In the case of Selective Privatization the low intra–iteration locality causes a replication of a high number of reduction array entries because they are accessed by several threads, even more for the colored mesh. So, Selective Privatization barely save extra memory compared to the full privatization. We observe that the achieved speedups for LocalWrite and SynchWrite get closer to the maximum expected, particularly for the colored mesh, for which a higher speedup is obtained compared to the sequential execution.

Secondly, the workload balancing problem is evaluated. With this aim, a code with this problem was chosen. It is a kernel for the Legendre transform used in numerical weather prediction [22]. In this code, irregular reductions are car-



(a) (b) (c)

Fig. 17. (a) Speedup for a 2-indirection reduction loop of the EULER code with a low intra–iteration locality (partially expanded SynchWrite and other techniques) (b) Speedup for the balanced SynchWrite and other techniques for the Legendre transform (c) Speedup of locally expanded SynchWrite and other techniques for the MD2 simulation code

ried out inside a multiply nested loop, where the innermost loop bounds are determined by subscripted subscripts, resulting in a potential workload imbalance. Fig. 17(b) shows the experimental results compared to the maximum theoretical achievable. We observe that the computational imbalance affects negatively Array Expansion because it partitions the outermost loop. This imbalance effect can be fixed in load balanced SynchWrite. For this improved solution, the granularity of the subblock partitioning ($nBlocks$) determines the effectiveness of the balancing procedure, given by $K = nBlocks/nThreads$. Note that as K is higher a better balancing is obtained for SynchWrite and consequently performance is better. The balancing procedure obtains a maximum improvement with a granularity of $K = 8$, although with $K = 4$ the maximum is very close. In this case, the divergence between the theoretical and observed values falls in an operative reason: it was necessary to flatten the nested loop in order to apply the locality based transformations (LocalWrite and SynchWrite). This loop flattening involves, in practice, an execution overhead.

Finally, with the aim of evaluating the local expansion approach we have selected a 2D short-range molecular dynamics simulation code [21] (MD2). A high contention region has been artificially introduced in the particle domain composed of 640K particles. This application simulates an ensemble of particles subject to a Lennard-Jones short-range potential. The use of a neighbour list results in an 2-indirection reduction nested loop. Two versions of the neighbour list were used, a sorted one (high inter–iteration locality) and a randomized one (low inter–iteration locality). Fig. 17(c) shows the speedup of the computation phase for locally expanded SynchWrite compared to other techniques and the maximum provided by our model. The high inter–iteration locality of the sorted neighbour list favors Selective Privatization both in performance and memory overhead. Basic LocalWrite (DWA–LIP) and SynchWrite suffers from load imbalance caused by the high contention regions. However, a local expansion of such regions improves the performance

of SynchWrite. For the randomized neighbour list, on the other hand, Selective Privatization performs worse because the number of conflicting regions increases drastically. However, in this case, LocalWrite and SynchWrite are able to exploit the inter–iteration locality maintaining their performance. Moreover, locally expanded SynchWrite can improve the efficiency with a lower memory overhead than full expansion. Both the overheads due to the initialization and final reduction stage of the expanded blocks and the conditional processing of them cause the observed speedup not to reach the values predicted by the model.

In all these experiments, the overhead of the prefetching phase ($\frac{1}{\eta_{c/i}\eta_{reuse}}$) for the locality based methods was not significant: less than 5% for the experiments in Fig. 17(a) (static, 20 timesteps) , about 1% for the experiments in Fig. 17(b) and below 1% for the experiments in Fig. 17(c) (dynamic, update each 10 time-steps).

8 Conclusions

In this paper, a general framework for the parallelization of irregular reductions is introduced in the context of shared memory multiprocessors. This framework is based on the definition of equivalence classes that considers write affinity. This affinity reveals the two sources of memory reference locality that exist in this kind of codes using indirections: intra–iteration and inter–iteration localities. Let us mention that the locality exploitation takes on special significance in the contemporary architectures due to the increasing gap between processor and memory latencies.

Although the framework is not directly usable in practice, it is a good starting point for the analysis of locality-based solutions. Likewise, existing techniques can be included and classified on the basis of this framework, as well as their performance limitations can be found. On the other hand, the framework can be useful to propose solutions to mitigate such potential limitations.

From this viewpoint, we have developed a formal performance model for the locality-based techniques, both existing ones and others proposed by the authors. The analysis is not only limited to the basic techniques but also it has been extended to some improved versions. The model allows to determine all aspects that have an influence, positive or negative, on the performance of the solutions. It also permits to establish the working conditions for which the performance is diminished, in order to apply the corresponding improved solution.

Finally, all discussed locality-based techniques have been implemented and evaluated experimentally, being compared to the theoretical results derived from our model. In addition, they were compared to other methods that were

used as reference. These experiments allow to verify the ability of the solutions to exploit locality as well as to overcome the performance problems through the improved versions. They have also allowed to validate the analytical model.

References

[1] I. Al-Furaih and S. Ranka, "Memory hierarchy management for iterative graph structures" *Int'l Parallel Processing Symposium*, Orlando, FL, Mar. 1998.

[2] W. Blume, R. Doallo , R. Eigenmann, J. Grout , J. Hoeflinger , T. Lawrence , J. Lee , D. Padua, Y. Paek, B. Pottenger, L. Rauchwerger and P. Tu, "Parallel programming with Polaris", *IEEE Computer*, 29(12):78–82, 1996.

[3] S. Chaudhry, P. Caprioli, S. Yip and M. Tremblay, "High-performance throughput computing", *IEEE Micro*, 25(3):32–45, 2005.

[4] C. Ding and K. Kennedy, "Improving cache performance in dynamic applications through data and computation reorganization at run time", *ACM SIGPLAN Conference on Programming Language Design and Implementation*, Atlanta, GA, May 1999.

[5] I.S. Duff, R.G. Grimes, and J.G. Lewis, "Users' guide for the Harwell-Boeing sparse matrix collection", *Technical Report TR/PA/92/86*, CERFACS, France, Oct. 1992.

[6] P. Feautrier, "Array expansion", *2nd International Conference on Supercomputing*, St. Malo, France, Jun. 1988.

[7] T. Davis, "University of Florida sparse matrix collection", *NA Digest*, 97(23), Jun. 1997.

[8] I. Foster, R. Schreiber and P. Havlak, "HPF-2, Scope of activities and motivating applications", *Technical Report CRPC-TR94492*, Rice University, Nov. 1994.

[9] A. Gibbons, *Algorithmic Graph Theory*, Cambridge University Press, 1999.

[10] E. Gutiérrez, O. Plata and E.L. Zapata, "An automatic parallelization of irregular reductions on scalable shared memory multiprocessors", *5th International Euro-Par Conference*, Tolouse, France, Sep. 1999.

[11] 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 International Conference on Supercomputing*, Santa Fe, NM, May 2000.

[12] E. Gutiérrez, O. Plata and E.L. Zapata, "Data-partitioning based parallel irregular reductions", *Concurrency and Computation: Practice and Experience*, 16(2–3):155-172, 2004.

[13] M. Hall, J. Anderson, S. Amarasinghe, B. Murphy, S.W.Liao, E. Bugnion, and M.S. Lam, "Maximizing multiprocessor performance with the SUIF compiler", *IEEE Computer*, 29(12):84–89, 1996.

[14] H. Han and C.W. Tseng, "Efficient compiler and run-time support for parallel irregular reductions", *Parallel Computing*, 26(13–14):1861–1887, 2000.

[15] H. Han and C.W. Tseng, "A comparison of parallelization techniques for irregular reductions", *15th IEEE International Parallel and Distributed Processing Symposium*, San Francisco, CA, Apr. 2001.

[16] H. Han and C.W. Tseng, "Exploiting Locality for Irregular Scientific Codes", *IEEE Transactions on Parallel and Distributed Systems*, 17(7):606–618, 2006.

[17] R. Jin, G. Yang and G. Agrawal, "Shared memory parallelization of data mining algorithms: Techniques, programming interface, and performance ", *IEEE Transactions on Knowledge and Data Engineering*, 17(1):71–89, 2005.

[18] 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.

[19] D.J. Mavriplis, R. Das, J. Saltz and R.E. Vermeland, "Implementation of a parallel unstructured Euler solver on shared and distributed memory architectures" *The Journal of Supercomputing*, 8(4):329–344, 1995.

[20] J.M. Mellor-Crummey, D.B. Whalley and K. Kennedy, "Improving memory hierarchy performance for irregular applications", *13th ACM International Conference on Supercomputing*, Rhodes, Greece, Jun. 1999.

[21] J. Morales and S. Toxvaerd, "The cell-neighbour table method in molecular dynamics simulations", *Computer Physics Communications*, 71:71–76, 1992.

[22] N. Mukherjee and J.R. Gurd, "A comparative analysis of four parallelisation schemes", *13th ACM International Conference on Supercomputing*, Rhodes, Greece, Jun. 1999.

[23] OpenMP Architecture Review Board, "OpenMP API Version 2.5", URL: `www.openmp.org`, May 2005.

[24] N.M. Strout, L. Carter and J. Ferrante, "Compile-time composition of run-time data and iteration reorderings", *ACM SIGPLAN Conference on Programming Language Design and Implementation*, San Diego, CA, Jun. 2003.

[25] Y. Saad, "Sparskit: A basic tool kit for sparse matrix computations", *Technical report*, University of Minnesota, MN, 1994.

[26] H. Yu and L. Rauchwerger, "An adaptive algorithm selection framework for reduction parallelization", *IEEE Transactions on Parallel and Distributed Systems*, 17(10):1084–1096, 2006.

[27] G.M. Zoppetti, G. Agrawal and R. Kumar, "Compiler and runtime support for irregular reductions on a multithreaded architecture", *IEEE International Parallel and Distributed Processing Symposium"*, Fort Lauderdale, FL, Apr. 2002.