Improving Parallel Irregular Reductions Using Partial Array Expansion

Eladio Gutiérrez Oscar Plata Emilio L. Zapata

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

{eladio,oscar,ezapata}@ac.uma.es

ABSTRACT

Much effort has been devoted recently to efficiently parallelize irregular reductions. In this paper, parallelizing techniques for these computations are analyzed in terms of three performance aspects: parallelism, data locality and memory overhead. These aspects have a strong influence in the overall performance and scalability of the parallel code. We will discuss how the parallelization techniques usually try to optimize some of these aspects, while missing the other(s). We will show that by combining complementary techniques we can improve the overall performance/scalability of the parallel irregular reduction, obtaining an effective solution for large problems on large machines. Specifically, a combination of array expansion and a locality-oriented method (DWA-LIP), named partial array expansion, is introduced. An implementation of the proposed method is discussed, showing that the transformation that the compiler must apply to the irregular reduction code is not excessively complex. Finally, the method is analyzed and experimentally evaluated.

1. INTRODUCTION

Many scientific/engineering applications are based on complex data structures that introduce irregular memory access patterns. In general, automatic parallelizers obtain sub–optimal parallel codes from those applications, as traditional data dependence analysis and optimization techniques are precluded. Run–time techniques have been proposed in the literature to support the parallelization of irregular codes, like those based on the inspector-executor paradigm [10], or the speculative execution of loops in parallel [11].

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

SC2001, November 2001, Denver, Colorado, USA


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

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

Figure 1: A loop with multiple reductions

In this paper, we analyze the techniques proposed in the literature to parallelize irregular reductions in terms of three performance aspects: parallelism, data locality and memory overhead. These aspects have a strong influence in the overall performance and scalability of the parallel reduction code. We will discuss how the parallelization techniques usually try to optimize one or two of the above-mentioned aspects, missing the other(s). That is, the parallel code is usually not optimal in terms of both performance and scalability. In fact, we may distinguish two categories of methods, *parallelism-oriented techniques* and *locality-oriented techniques*, being the former typically less scalable than the latter. We will show that we can combine in a natural way techniques from both categories, with the aim of improving the overall performance/scalability of the parallel irregular reduction. Specifically, a combination of array expansion and a locality-oriented method results in a technique *partial array expansion* where the overall scalability is improved (memory overhead is reduced) while data locality and parallelism are highly exploited.

2. PARALLELISM, DATA LOCALITY AND MEMORY OVERHEAD

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

2.1 Reduction Parallelization Techniques

Different specific solutions to parallelize irregular reductions in a shared memory model have been proposed in the literature. We may classify them into two categories: *parallelism-oriented techniques* (POT) and *locality-oriented techniques* (LOT). There is also the simple solution based on critical sections, where the reduction loop is executed fully parallel by just enclosing the accesses to the reduction array in a critical section.

The POT category includes two important methods. One method (*replicated buffer*) replicates private copies of the reduction array on all threads. Other method (*array expansion*) expands the reduction array by the number of parallel threads. These techniques transforms the reduction loop into a fully parallel one (parallelism-oriented). However, they have scalability problems for large data sets, due to the full privatization of the reduction array on all threads

Methods in the LOT category avoid the privatization of the reduction array by the introduction of an inspector whose net effect is the reordering of the reduction loop iterations (through the reordering of the subscript arrays). In addition, this reordering is also used to exploit data (reference) locality.

We will highlight here two important methods in LOT category. One method was termed LOCALWRITE [5, 6, 7], based on the *owner–computes rule*, where each thread owns and updates a portion of the reduction array (block partitioning). Note, however, that, in order to fulfill the computes rule, those iterations that updates more than one block of the reduction array must be replicated across the owner threads. This computation replication introduces a performance penalty.

An alternative method that avoids computation replication is DWA–LIP (Data Write Affinity with Loop Index Prefetching) [2, 3, 4]. Consider that the blocks of the reduction array are indexed by the natural numbers. The inspector (named *loop-index prefetching* phase, or LIP) now sorts all the iterations of the reduction loop into sets characterized by the pair $(B_{min}, \Delta B)$, where B_{min} (B_{max}) is the minimum (maximum) index of all blocks touched by the iterations in that set, and ΔB is the difference $B_{max} - B_{min}$. The execution phase (or computation phase) of the method is organized as a sequence of non-conflicting (parallel) stages. In the first stage, all sets of iterations of the form $(B_{min}, 0)$ are executed in parallel because they are all data flow independent (optimal utilization of the threads). The second stage is split into two sub-stages. In the first one, all sets $(B_{min}, 1)$ with an odd value of B_{min} are executed fully parallel, followed by the second sub-stage where the rest of sets are executed in parallel. A similar scheme is followed in the subsequent phases, until all iterations are exhausted (see execution flow in Fig. 2 (a)). A direct translation into code of this execution flow is shown in Fig. 2 (b) (arrays `init()`, `count()` and `next()` implement the iteration sets built by the inspection phase).

2.2 Performance Aspects

Methods in the POT and LOT categories have, in some sense, complementary performance characteristics. Methods in the first class exhibit optimal parallelism exploitation (the reduction loop is fully parallel), but no data locality is taken into account and lack scalability (memory overhead is proportional to the number of threads). Methods in the second class, however, exploit data locality and exhibit usually much lower memory overhead, and it is not dependent on the number of threads (the inspector may need some extra buffering to store subscript re-orderings, independently on the number of threads). However, either the method introduces some computation replication or is organized in a number of synchronized phases. In any case, this fact represents loss of parallelism.

Fig. 3 shows a space representation of the three mentioned performance aspects, parallelism, data (reference) locality and memory overhead, for two representative methods (one POT, array expansion, and one LOT, DWA–LIP). The parallelism axis shows that array expansion exploits always full parallelism out of the reduction loop, while DWA–LIP depends on the *intra-loop locality*, that is, the locality to index the reduction array `A()` by all the subscript arrays `f1()`, ..., `fn()` in a particular loop iteration. If different blocks are touched in the same iteration some sets $(B_{min}, \Delta B)$, with $\Delta B > 0$, are not empty, exhibiting the method a sub-optimal parallelism. The reference locality axis shows how much data locality the method exploits. Array expansion shows no locality exploitation at all, depending completely on how well ordered is the input data. DWA–LIP, however, reorders the reduction loop iterations and sorts them out into sets. Thus, *inter-loop locality* is optimized. It can be say that DWA–LIP trades locality for parallelism.

The memory overhead axis shows that, in general, array expansion needs much more extra memory than DWA–LIP, specially for a high number of threads. The first method expands `A()` by the number of threads, while the second method only needs to duplicate one of the subscript arrays (independently on the number of threads) and to introduce two small additional arrays to index and count the iteration sets (see the implementation in [3]). Note, also, that a subscription array is a one-dimensional array storing integers, while the reduction array, despite usually shorter, however is a one-, two- or three-dimensional array storing doubles.

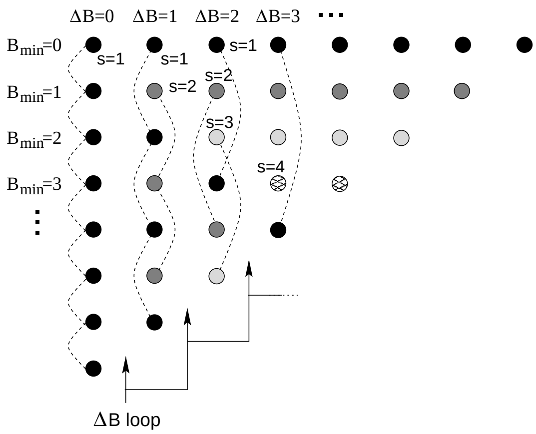
The first two aspects, together with the machine architecture (memory hierarchy), determines the overall performance of the parallel code (in terms of the execution time), as shown in the bottom axis in Fig. 3. If the intra-loop locality is low, DWA–LIP has lower performance than array expansion because the improvement in the inter-loop locality does not reduce the execution time enough to compensate the parallelism loss. On the other hand, in the opposite case (which is usually the normal case for real-world applications), DWA–LIP performs better than array expansion, as parallelism exploitation is almost optimal and additionally inter-loop locality is taken into account.

In general, LOTs exploit partially the reference locality (only the inter-loop locality). This way, these methods usually performs sub-optimally if the input data is not well ordered (enumerated) at the intra-loop level. For instance, if the input data corresponds to a finite element mesh, the intra-loop locality is associated with the mesh elements (arcs, triangles, or whatever). In these cases, array expansion usually performs better, but it is not a scalable solution.

To obtain the best from both classes of parallelization techniques, in this paper we propose a combination of DWA–LIP and array expansion, named *DWA–LIP with partial array expansion*, or simpler, *partial array expansion* (PAE). Specifically, we propose a new method based on DWA–LIP but introducing partial replication of the reduction array. We will show that the new method performs as well as array expansion in situations with low intra-loop locality but needing a much lower extra memory (improving scalability).

3. PARTIAL ARRAY EXPANSION

Different solutions has been proposed recently to reduce the high memory overhead of array expansion. The *reduction table* method [8] assigns a private buffer to each thread of a fixed size (lower than the size of the reduction array). Then, each thread works on its private buffer indexed by using a fast hash formula. When the hash table is full, any new operation will work directly on the global reduction array within a critical section. Other method is *selective privatization* [13], where the replication include only those elements referenced by various threads. It first determine (inspector phase) which are those elements and then allocate for them private



(a)

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

(b)

Figure 2: (a) Model of the execution phase of DWA–LIP, and its application to the reduction loop in Fig. 1

storage space. Each thread, then, works on its private buffer when updating conflicting elements, while it works on the global reduction array otherwise. This execution behavior implies a replication of each subscript array in order to store the new indexing scheme. Some sort of combination of the above both techniques has been also proposed in the literature [13].

All these solutions only stress on the memory overhead problem of pure array expansion. Here we will propose a different solution that implicates all three space axes in Fig. 3. That is, the solution will try to maximize reference locality and parallelism while reducing at most memory overhead.

The PAE method is obtained from DWA–LIP by replicating the reduction array a fixed number of times, always less than the number of threads. The number of copies of the reduction array will be the *partial expansion factor* (ρ). This replication increases the parallelism exploitable by DWA–LIP, as, for a particular ΔB value (that is, a column in Fig. 2 (a)), conflicting iteration sets may now be non-conflicting because they have the possibility of updating different private copies of the reduction array. In other words, as ρ private copies of the reduction array are available, there is always the opportunity of having, at least, ρ threads working in parallel.

The hard problem here is how to schedule the iteration sets so as we can benefit from this parallelism most of the time. Returning to Fig. 2 (a), we observe that for each column of nodes, the number of conflicting super-sets (represented in the figure by linked nodes of the same color) is equal to $\Delta B + 1$. If the reduction array is replicated ρ times, then, for each column, ρ super-sets stop being conflicting, as each one may work on a different private copy. Taking in mind this fact, we can prove that PAE shares the same execution model than DWA–LIP but considering that the number of conflicting super-sets in each column is now $\Delta^{exp} = \left\lfloor \frac{\Delta B}{\rho} + 1 \right\rfloor$.

There are different possibilities to assign private copies of the reduction array to super-sets of iterations sets. A simple one, that results into a compact code, consists in assigning cyclically each super-set to each private buffer, from top to bottom in the corresponding column. This simple execution model has a limitation. Considering a specific column in Fig. 2 (a), the average number of non-conflicting iteration sets (nodes) is

$$\frac{nThreads - \Delta B}{\Delta^{exp}}, \qquad (1)$$

assuming that the number of threads ($nThreads$) is equal to the number of nodes in the leftmost column (that is, the reduction array was block partitioned assigning one block per thread). The above number (1) is decreasing when ΔB increases. In particular, when

$$\Delta B > \frac{nThreads - 1}{2}, \qquad (2)$$

then (1) is less than ρ. That means, if we continue with the same execution model, we go under the minimum exploitable parallelism for all columns verifying (2). To solve this situation, for all iterations sets with ΔB verifying (2) we change the execution model as follows. In each column, the iteration sets are grouped into supersets of, at most, ρ elements. All set in each super-set can be executed in parallel, working on different private arrays.

Fig. 4 depicts the new execution model for PAE (for $\rho = 2$). For the four leftmost columns, the execution model is similar to DWA–LIP (but considering Δ^{exp}) because (1) is greater than ρ. In general, comparing this execution flow with that in Fig. 2 (a), we note a significant increase in parallelism.

A code implementing the execution model for the discussed partially expanded DWA–LIP is shown in Fig. 5. At the beginning of the code appears an initialization stage of the partially expanded reduction array A_e(). At the end, the code computes the final global reduction from the local copies. In the middle, the computation phase follows a similar structure of DWA–LIP, presented in Fig. 2 (b). If expression (1) is not less than ρ then s and B_{min}

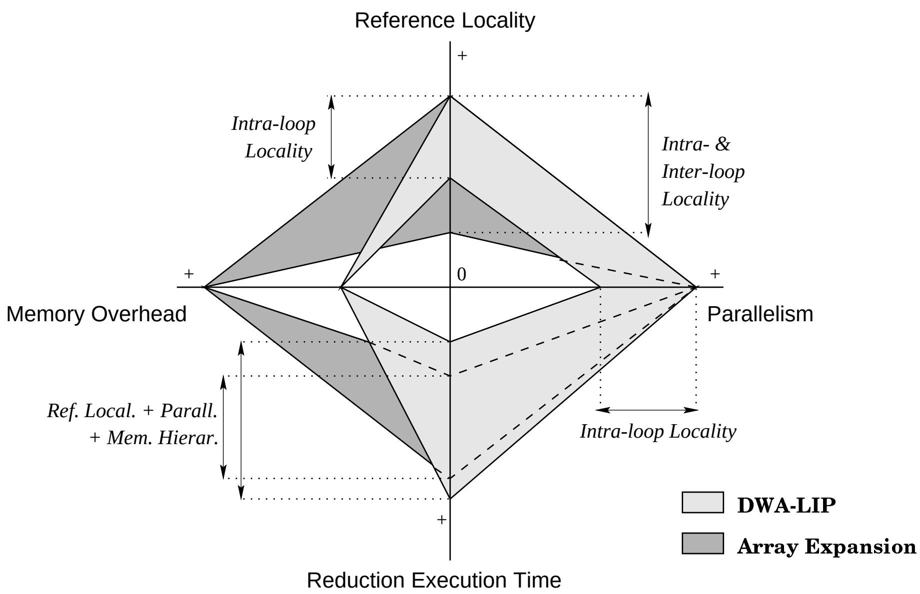


Figure 3: Space representation of parallelism, reference locality and memory overhead for array expansion and DWA–LIP

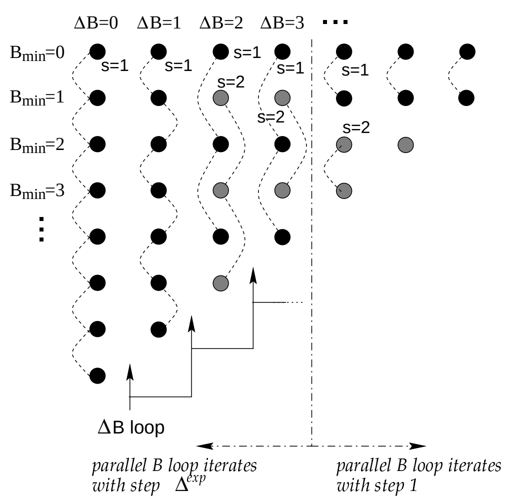


Figure 4: Model of the execution phase of PAE

loops behavior in the same way than in Fig. 2 (b) but using Δ^{exp} instead of $\Delta B + 1$. Otherwise, B_{min} loop iterates one-by-one with a total length not greater than ρ. Finally, it should be noted that the inspection phase (LIP) is exactly the same than in DWA–LIP.

4. PERFORMANCE ANALYSIS AND EVALUATION

4.1 Algorithm Analysis and Evaluation

The performance of the PAE method is determined by both, the index prefetching phase (inspector) and the computation phase (executor) overheads. We characterize the computational cost of the prefetching phase by means of two parameters: the *reuse factor*, η_{reuse} and the *computation/prefetching ratio*, $\eta_{c/p}$. The reuse factor measures the number of times that the computation phase is executed per each prefetching phase. We only need to re-execute the prefetching phase when the subscripts arrays in the reduction loop change. The updating frequency of the inspection is not usually very high in real codes. The computation/prefetching ratio is a measure of the time spent in one execution of the computation phase relative to one execution of the index prefetching phase. Although both, prefetching and computation phases, run over the same iteration space, one execution of the computation phase has usually a much greater cost than the same for the prefetching phase.

From the above parameters, the total parallel execution time of the reduction loop can be stated as, $T = T_{\text{computation}} + T_{\text{prefetching}} = T_{\text{computation}}\left(1+\frac{1}{\eta_{c/p}\,\eta_{\text{reuse}}}\right)$. Note that if η_{reuse},$\eta_{c/p}$ have a high value for a problem, the fraction of time spent in the prefetching phase will be no significant in relation to the computation phase, as $\left(1+\frac{1}{\eta_{c/p}\,\eta_{\text{reuse}}}\right)$ will be close to the unit. In addition, as the prefetching phase can be parallelized, the relative overhead is not dependent on the number of threads.

We can identify three overhead sources in the computation phase: synchronization between iteration sets that are executed in parallel, *copy-in* and *copy-out* stages associated with the partially expanded array (initialization and global reduction), and the parallelism loss associated to iteration sets executed in parallel on a number of threads less than total. The number of synchronization operations in the computation phase code (see Fig. 5) is given by the total number of executions of the B_{min} loop, which is $O(nThreads^2)$, and it corresponds to the parallel execution of non-conflicting iteration sets. In most real codes this amount is small in comparison with the number of iterations of the parallelized loop. Except for a very large number of threads we should not worry about this source of overhead. The initialization of the private copies of the reduction array and the final global reduction, have both a complexity of $O(Adim\,\rho)$. As these operations can be done fully in parallel, we can reduce its workload to $O(\frac{Adim\,\rho}{nThreads})$. That is, in contrast to array expansion, this overhead diminishes as the number of threads increases. Additionally, it is also bounded by the partial expansion factor ρ.

Not considering, then, the above two overhead sources, the parallel execution time of the computation phase may be written as follows,

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

c$omp parallel
c$omp do
do j = 1,ρ
   do i = 1, ADim
      A_e(i, j)=0
   enddo
enddo
c$omp end do
do ΔB = 0, nThreads−1
   if (ΔB .le . (( nThreads−1)/2)) then
      rD = floor (ΔB/ρ+1)
      s_end = rD
      s_step = 1
      B_step = rD
   else
      rD=1
      s_end = nThreads−ΔB
      s_step = ρ
      B_step = 1
   endif
   do s = 1, s_end, s_step
     if (ΔB .le . (( nThreads−1)/2)) then
        B_end = nThreads−ΔB
     else
        B_end = min(s+ρ−1,nThreads−ΔB)
     end
     c$omp do
     do B_min = s, B_end, B_step
        ir = mod((B_min−s)/rD,Rho)+1
        i = init (B_min,ΔB)
        do ii = 1, count(B_min,ΔB)
           Calculate ξ1,ξ2
           A_e(f1(i ), ir ) = A_e(f1(i ), ir )⊕ξ1
           A_e(f2(i ), ir ) = A_e(f2(i ), ir )⊕ξ2
                              ...
           A_e(fn(i ), ir ) = A_e(fn(i ), ir )⊕ξ2
           i = next(i )
        enddo
     enddo
     c$omp end do
   enddo
enddo
c$omp do
do i = 1, ADim
   do j = 1,ρ
      A(i ) = A(i)⊕A_e(i , j )
   enddo
enddo
c$omp end do
c$omp end parallel
```

Figure 5: Parallel loop with multiple reductions using PAE

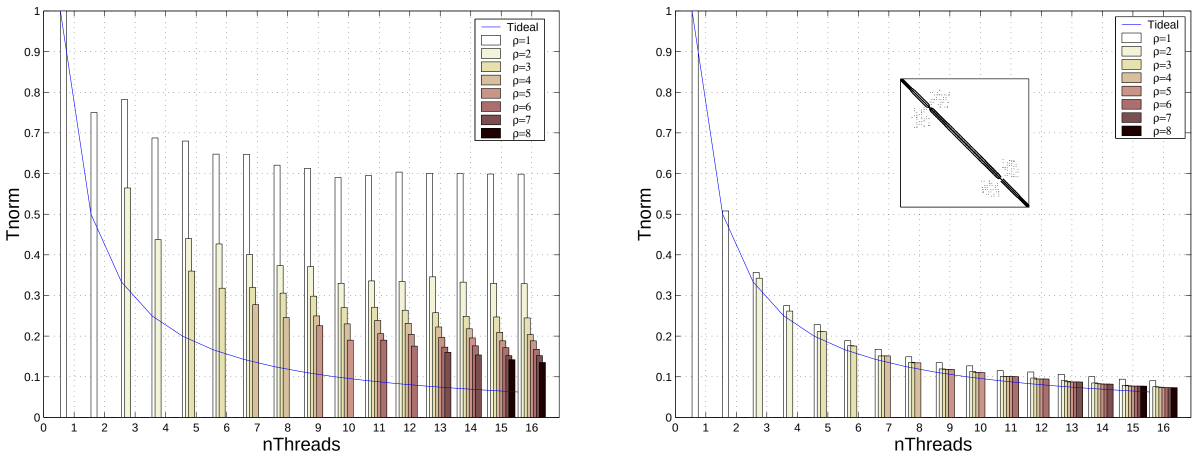
$$T^{PAR} = Tit^{SEQ} \cdot \left(\sum_{\Delta B=0}^{\lceil \frac{nThreads-1}{2} \rceil} \sum_{s=1}^{\Delta^{exp}} \left(\max_{\substack{B_{min} \leq nThreads-\Delta B \\ B_{min}=s+k\ \Delta^{exp}, \\ k\in\mathbb{N}}} \left\{ Nit(B_{min},\Delta B) \right\} \right) + \right.$$
$$\left. + \sum_{\Delta B=\lceil \frac{nThreads-1}{2} \rceil +1}^{nThreads-1} \sum_{\substack{s=1+n\rho,\ n\in\mathbb{N} \\ s\leq nThreads-\Delta B}} \left(\max_{\substack{B_{min} \leq nThreads-\Delta B \\ B_{min}=s+k, \\ k\in[1,\rho]\subset\mathbb{N}}} \left\{ Nit(B_{min},\Delta B) \right\} \right) \right) \tag{3}$$

The parallel execution time, T^{PAR}, has been expressed as a function of the sequential iteration time, Tit^{SEQ}, and the number of iterations in set $(B_{min},\Delta B)$, denoted by $Nit(B_{min},\Delta B)$. Observe that the coefficient $count(B_{min},\Delta B)$ of the DWA–LIP data structure must be equal to $Nit(B_{min},\Delta B)$ after the prefetching phase. According to the code in Fig. 5, the parallel execution time is determined by the maximum number of iterations of each set inside the parallel B_{min} loop.

Fig. 6 shows the evaluation of Eq. 3 for two different memory access patterns. We have represented the normalized parallel time ($Tnorm = \frac{T^{PAR}}{N\,Tit^{SEQ}}$, where N is the number of reduction loop iterations) of the computation phase for several values of the partial expansion factor ρ. The plot (a) in Fig. 6 displays the evaluation of the above equation for a dense memory access pattern, in which the reduction array is referenced by pairs $(n1,n2)$, where $n1$ and $n2$ go over all possible values. This is, for instance, the pattern for an all-to-all N-body problem. In part (b), on the other hand, we shows the results for a sparse pattern, specifically, the sparse matrix *fidapm11* belonging to the set FIDAP of SPARKSKIT collection [12]. The matrix size is 22294x22294 with 617874 non-zero entries (its shape is also shown in the plot).

For the dense pattern the number of iterations of the reduction loop is distributed equally among the entries $count(B_{min},\Delta B)$ of the DWA-LIP data structure. Thus, the exploited parallelism is very poor when the pure DWA–LIP method is used. However, greater values of ρ, reduces significantly this parallelism loss, obtaining a much faster parallel code. In the case of the sparse pattern, better reference locality exists. Therefore we can exploit the maximum parallelism available by using a smaller partial expansion factor (much smaller than full expansion). In Fig. 6 (b) we can see that most of the non-zero entries in the sparse matrix are placed near the diagonal. In real irregular and sparse codes this is a common situation.

Further analysis of Eq. 3 reveals that the maximum exploitable parallelism is achieved for $\rho = \frac{nThreads-1}{2}$ (denoted by ρ_{sat}). Consequently, although an extra amount of memory is used, making ρ greater than this value, the parallelism will not increase. For access patterns exhibiting high reference 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 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 $Nit(B_{min},\Delta B) = 0$ 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.



(a) (b)

Figure 6: Parallel load evaluation of PAE: (a) dense pattern, (b) *fidapm11* sparse matrix pattern

4.2 Experimental Results

The PAE method has been experimentally tested on the EULER code, from the motivating application suite of HPF-2 [1]. 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 code includes a single loop with two subscripted reductions on one array with three dimensions, which is placed inside an outer time-step loop. As an static problem, the inspector phase is computed only once.

We have parallelized EULER using PAE, DWA–LIP (PAE with $\rho = 1$), array expansion (optimized Polaris implementation), selective privatization, LOCALWRITE and a direct parallelization via the ATOMIC OpenMP clause (critical section). The experiments have been conducted on a SGI Origin2000 multiprocessor, with 250-MHz R10000 processors (4 MB L2 cache) and 12 GB main memory. Codes were parallelized using OpenMP and compiled using the SGI MIPSpro Fortran77 compiler (with optimization level `O2`).

In a previous work [3], we have compared pure DWA–LIP with array expansion for this same EULER code using the original input mesh. This input data presents a high intra-loop reference locality, resulting in a better behavior in performance for DWA–LIP against array expansion. In this section, we change the input data with the aim of reducing the intra-loop reference locality. This way, a much lower performance is expected for pure DWA–LIP method.

The parallel EULER kernel has been tested using a 1161K nodes mesh with a connectivity of 8 (ratio between edges and nodes). Two versions of the mesh has been generated. One of them is obtained after applying a coloring algorithm to the edges, and placing edges of the same color consecutively in the indirection array. For this version we expect a low inter-loop locality in access to the reduction array between different iterations. In the other version the list of edges has been sorted, and the locality between iterations is expected to be higher. This fact becomes evident in serial reduction executions, which is 74.2 sec. for the colored version and 34.7 sec. for the sorted version.

The tested mesh has low intra-loop locality as shown in table 1, where the percentage of the total iterations in the set $(B_{min}, \Delta B)$ is given for different values of the number of threads and ΔB. Note that there is a significant fraction of iterations with a high ΔB value. Therefore the performance of the pure DWA-LIP method will be low due to the parallelism loss in iteration sets with a high ΔB.

ΔB / Threads	1163 knodes				
	0	1	2	3	Remainder
1	100%				
2	90.7%	9.3%			
4	86.0%	6.8%	5.0%	2.1%	
8	83.5%	4.7%	3.0%	2.9%	5.9%
16	80.9%	4.8%	1.6%	1.5%	11.2%

Table 1: Percentage of the total number of reduction iterations for different values of ΔB

In Fig. 7 we have plotted the parallel execution times and the speedup (referenced to the sequential execution time) for the reduction loops of analyzed methods. Observe that the pure DWA-LIP method, that is, when ρ is equal to one, has a lower performance than array expansion technique. The reason is the parallelism loss due to iteration sets with a high ΔB parameter.

For all cases we observe a poor behaviour of the parallelization using ATOMIC OpenMP clause due to the high time consumption of the implementation of barriers on the target machine.

As the intra-loop locality of the tested mesh is low, selective privatization method will replicate a high number of elements of the reduction array, due to the fact that these elements will be modified by several threads, even more for the colored mesh. This is the reason for the low performance of this method.

LOCALWRITE suffers from parallelism loss due to the computation replication in boundary iterations. In the tested code the reduction loop have two indirection arrays, thus we will loose half of parallelism in the boundary iterations. As the tested data has low intra-loop locality, the number of these iterations is relatively high.

The colored mesh shows a low inter-loop reference locality in the accesses to the reduction array of consecutive iterations. Thus we expect a worse behavior of array expansion in this case. We observe that for more than 8 processors we can reach a better execution time with partially expanded DWA-LIP using $\rho = 4$. For 16 threads and $\rho = 8$, the DWA-LIP based parallelization outperforms array expansion.

The access inter-loop locality is better for the sorted version of

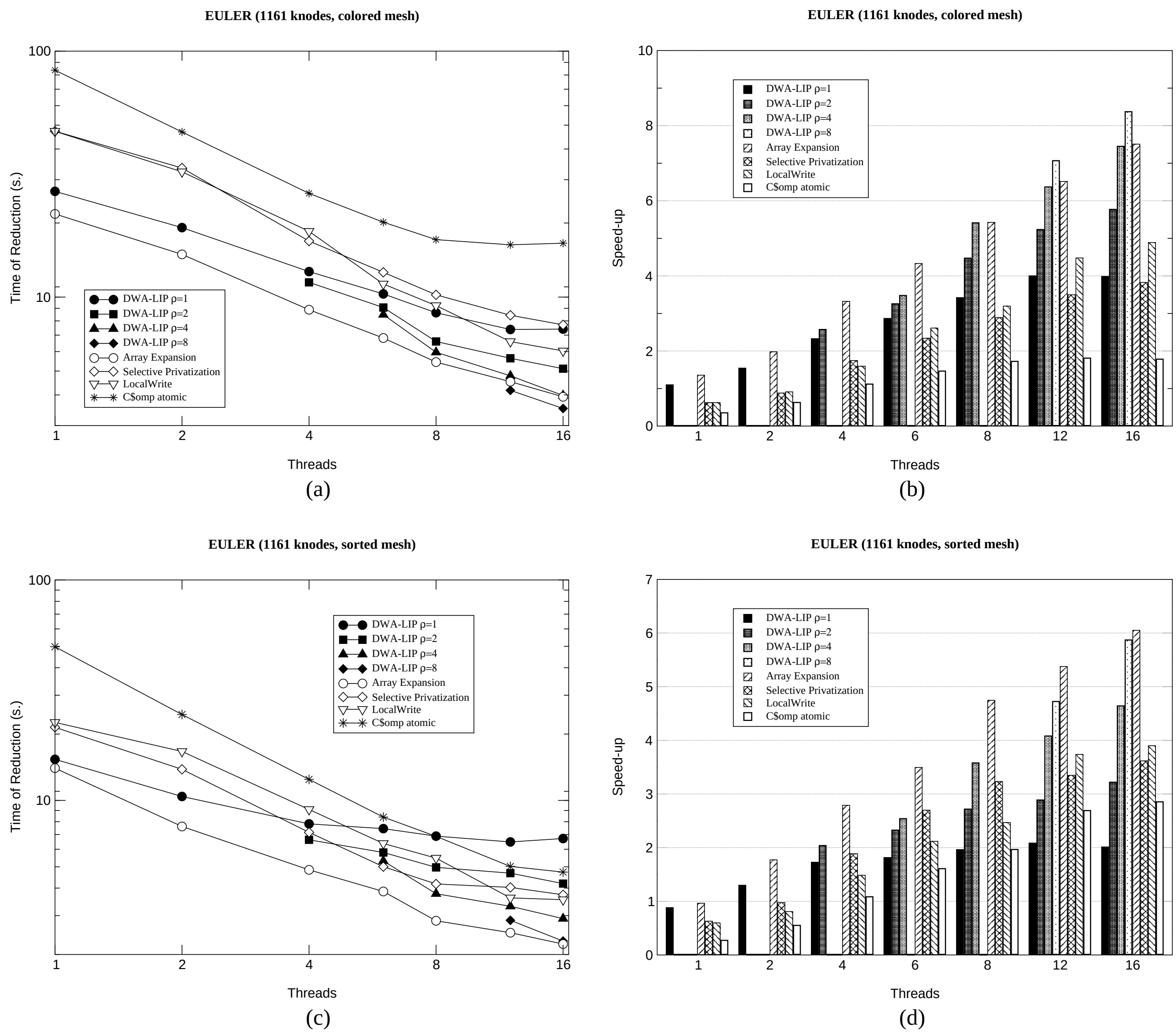


Figure 7: Parallel execution times and speedups for the parallel EULER code using DWA-LIP, PAE, array expansion, selective privatization, LOCALWRITE and critical section methods (colored (a,b) and sorted (c,d) meshes)

the mesh. In this case, DWA-LIP does not benefit from this increment of locality, because, for this mesh, the main limitation is the parallelism loss caused by the low intra-loop locality. Nevertheless we observe that for a given number of threads the parallel execution time 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 $\rho = 8$.

The inspection time for methods using an inspector stage is 3.2s, 2.5s and 5s., for PAE, LOCALWRITE, and selective privatization. For PAE and LOCALWRITE this overhead is not very significant. In particular, the overhead inspector factor $\left(1+\frac{1}{\eta_{c/p}\,\eta_{\text{reuse}}}\right)^{-1}$ is 0.95 (PAE), 0.96 (LOCALWRITE) and 0.93 (selective privatization) for the colored version of the mesh, and 0.91 (PAE), 0.93 (LOCALWRITE), and 0.87 (selective privatization) for the sorted version.

The extra memory needed by both methods is another important overhead factor. Fig. 8 shows the memory overhead of the different methods regarding only the replication of reduction array elements and including the inspector data structures. This overhead is plotted taken the total size of the reduction arrays in the sequential code as unit. We observe that array expansion has the maximum memory overhead, growing linearly with the number of threads. The number of reduction array elements replicated by selective privatization is high because of the low intra-loop locality of the tested pattern. It is even higher than in array expansion when considering inspector data, because selective privatization needs a copy of the indirection arrays to translate subscripts. It can be seen that the PAE method allows to reduce the memory requirements compared to other methods, obtaining at the same time similar or better performance. Considering, for instance, 16 threads in parallel, in our experiments the PAE method with $\rho = \frac{nThreads}{2}$ exhibits similar/better speedup than array expansion.

5. CONCLUSIONS

There is a much interest in the compiler literature to parallelize efficiently irregular reductions, as these operations appear frequently

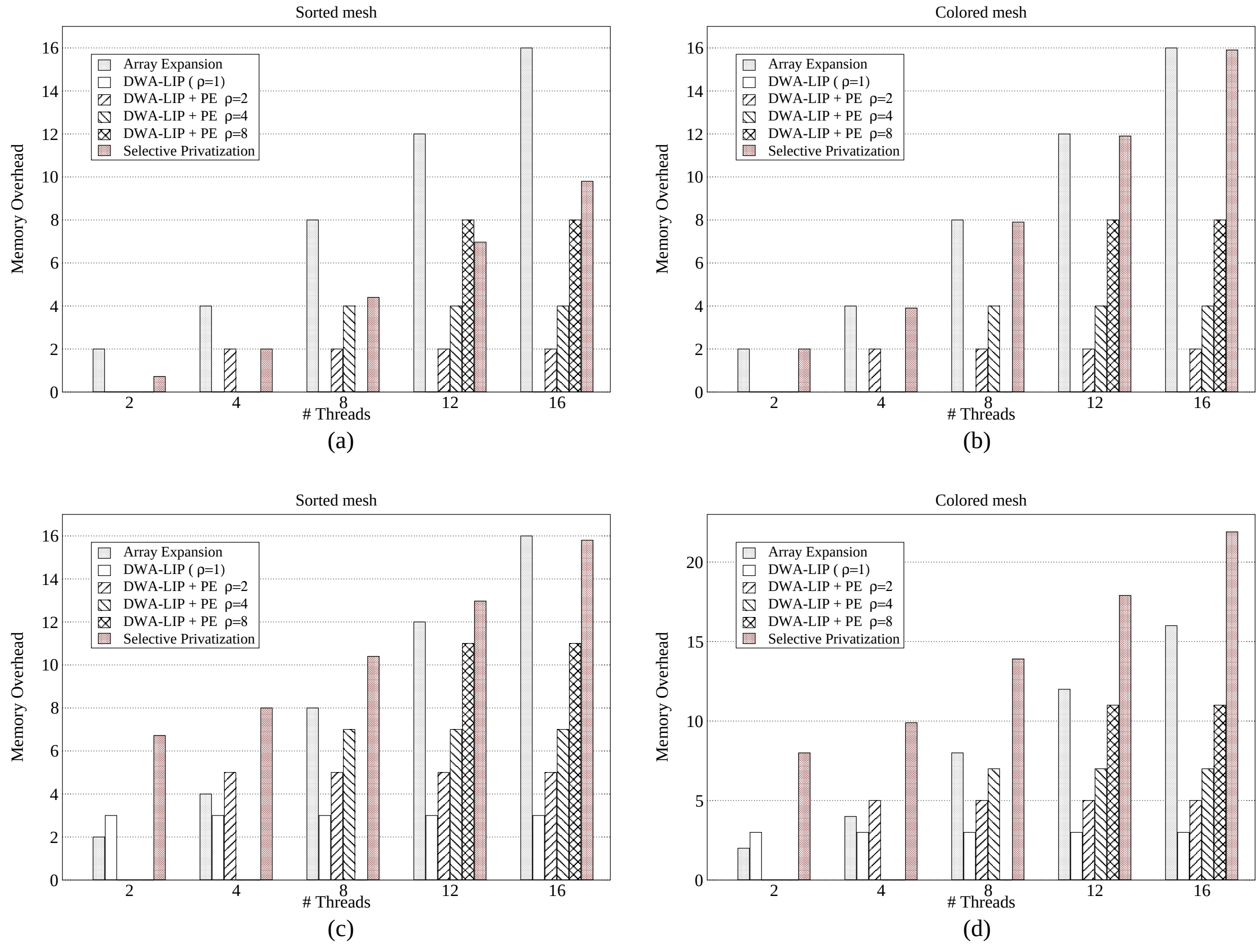


Figure 8: Memory overhead regarding only the replication of array reduction elements (a,b) and including inspector data structure (c,d)

in the core of many numerical applications. Many parallelization techniques have been proposed elsewhere, that can be classified into two groups. The first one, basically try to exploit maximum parallelism, paying low attention to data locality and memory overhead. In the second group, the situation is the opposite. Data locality is exploited, and extra memory is saved, but losing opportunities to exploit parallelism.

In this paper we have proposed a new method that combines both worlds, parallelism-oriented and locality-oriented methods, trying to exploit reference locality and limit memory overhead and, at the same time, taking advantage of maximum parallelism. This new method derives directly from the combination of a locality-oriented method, DWA–LIP, and array expansion.

We have analyzed quantitatively and experimentally our method and proved that, for real-world applications, it is easy to perform as well as or better than array expansion and other parallel reduction methods but using a much lower extra memory.

6. REFERENCES

[1] I. Foster, R. Schreiber and P. Havlak, *HPF-2, Scope of Activities and Motivating Applications, Technical Report CRPC-TR94492*, Rice University, November 1994.

[2] E. Gutiérrez, O. Plata and E.L. Zapata, *An Automatic Parallelization of Irregular Reductions on Scalable Shared Memory Multiprocessors*, **5th Int'l. Euro-Par Conf. (EuroPar'99)**, Tolouse, France, pp. 422–429, August-September 1999.

[3] E. Gutiérrez, O. Plata and E.L. Zapata, *A Compiler Method for the Parallel Execution of Irregular Reductions in Scalable Shared Memory Multiprocessors*, **14th ACM Int'l Conf. on Supercomputing (ICS'2000)**, Santa Fe, NM, pp. 78–87, May 2000.

[4] E. Gutiérrez, R. Asenjo, O. Plata and E.L. Zapata, *Automatic Parallelization of Irregular Applications* **J. Parallel Computing**, 26(13–14):1709–1738, December 2000.

[5] H. Han and C.-W. Tseng, *Improving Compiler and Run-Time Support for Irregular Reductions*, **11th Workshop on Languages and Compilers for Parallel Computing (LCPC'98)**, Chapel Hill, NC, August 1998.

[6] H. Han and C.-W. Tseng, *Efficient Compiler and Run–Time Support for Parallel Irregular Reductions*, **J. Parallel Computing**, 26(13–14): 1709-1738, December 2000.

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

[8] 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 (LCR'98), Pittsburgh, PA, May 1998.

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

[10] R. Ponnusamy, J. Saltz, A. Choudhary, S. Hwang and G. Fox, *Runtime Support and Compilation Methods for User-Specified Data Distributions*, *IEEE Trans. on Parallel and Distributed Systems*, vol. 6, no. 8, pp. 815–831, June 1995.

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

[12] Y. Saad, *Sparskit: A Basic Tool Kit for Sparse Matrix Computations, Version 2*, **Technical report, Computer Science Department**, University of Minnesota, MN, June 1994.

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