Parallel Techniques in Irregular Codes: Cloth Simulation as Case of Study ⋆

Eladio Gutiérrez, Sergio Romero, Luis F. Romero, Oscar Plata and Emilio L. Zapata

Department of Computer Architecture,
University of Málaga,
29071 Málaga, SPAIN

Abstract

When parallelizing irregular applications on ccNUMA machines several issues should be taken into account in order to achieve high code performance. These factors include locality exploitation and parallelism, as well as careful use of memory resources (memory overhead). An important number of numerical simulation codes are clear examples of irregular applications. Frequently this kind of codes includes reduction operations in their core, so that an important fraction of the computational time is spent on such operations. Specifically, cloth simulation belongs to this class of applications, being a topic of increasing interest in diverse areas, like in the multimedia industry. Moreover, when real time simulation is the aim, its parallelization becomes an important option.

This paper discusses and compares different irregular reduction parallelization techniques on ccNUMA share memory machines. Broadly speaking, we may classify them into two groups: privatization–based and data partitioning–based methods. In this paper we describe a framework, based on data affinity, that permits to develop various algorithms inside the group of the data partitioning–based techniques. All these techniques and approaches are analyzed and adapted to the computational structure of a real, physically-based, cloth simulator.

Key words: Irregular reductions, cloth simulation, data locality, reduction parallelization, ccNUMA multiprocessor

⋆ This work was supported by Ministry of Education and Culture (CICYT), Spain, through grant TIC2003-06623

1 Introduction

In the last years, multimedia industry have been requiring better and faster tools for cloth simulation to be used in virtual humans, dynamic sceneries and computer games. These applications must perform realistic simulations and should run in real time for interactive purposes. For the former, a physically–based model is usually preferred, although it requires a high computational cost. To reach real time, optimization of each simulator stage is mandatory, including its parallelization.

Codes for this kind of applications spent a significant portion of its execution time in irregular reduction operations. These correspond to the computation of physical magnitudes, such forces acting over cloth particles. This process determines vector and matrix coefficients necessary to solve the differential equations that models the cloth behavior.

This work is focused on the efficient parallelization of reduction operations in cloth simulation codes in the context of ccNUMA share memory machines, where data locality existing in access patterns should be taken into account. We have developed a parallelization technique for reduction operations that is able to exploit the locality exhibited by memory accesses. This technique can be easily implemented in a compiler and automatically applied to irregular reduction codes. In this paper we use it to parallelize the reduction loops included in a cloth simulation code. Our method is compared with other well known techniques in terms of execution time and memory requirements.

The rest of the paper is organized as follows. Section 2 introduces a framework for parallelizing irregular reductions which exploits memory access locality. Section 3 describes our cloth simulator and also analyzes the characteristics of its reduction loops. Section 4 discusses the different parallelization techniques applicable to the reduction loops in the cloth simulator and their effects in performance. Section 5 provides an experimental comparison of the discussed techniques. Finally, section 6 concludes the paper.

2 Locality exploitation in parallel irregular reductions

Many common data organizations used in numerical applications involve irregular memory accesses, in which array elements are referenced by means of indirections. Reduction operations are often found in the context of irregular codes in scientific and numerical applications, representing an important class of irregular problems. Reduction operations are based on commutative and associative operators, like additions or multiplications.

When the reduction operator is applied to several entries of an array inside a loop, and the commutative and associative properties are preserved, the term

```
double R[RDim];
int  f1[N1][N2]...[NnLoops], f2[N1][N2]...[NnLoops], ...
     fnInd[N1][N2]...[NnLoops];

for (i1=0;i1<N1;i1++)
    for (i2=0;i2<N2;i2++)
      ...
          for (inLoops=0;inLoops<NnLoops;inLoops++) {
              R[f1[i1][i2]...[inLoops]] += ξ1();
              R[f2[i1][i2]...[inLoops]] += ξ2();
              ...
              R[fnInd[i1][i2]...[inLoops]] += ξnInd();
              }
```

Fig. 1. Nested loop with multiple irregular reductions

histogram reduction is used. An example piece of a histogram reduction is shown in Figure 1. This case corresponds to multiple reduction operations. Vector $R[\,]$ represents the reduction array (that could be multidimensional), which is updated (the operator + is used in this example) through the subscript arrays $f_1[\,]$, $f_2[\,]$, ... Functions $\xi_1()$, $\xi_2()$, ... represent the actual computation and they must not include references to the reduction array. In addition, the indirection arrays must not be modified inside the loop. In practice this functions share some common calculations and are usually computed before reductions in the loop body.

The recognition of the irregular reduction and which arrays work as reduction array(s) and which ones as subscript arrays may be accomplished in a compiler through the use of pattern-matching or idiom recognition techniques [18]. Analyzing memory references we observe that loop–carried data dependences may be present, due to the subscripted subscripts. Nevertheless, because of the associative and commutative properties satisfied by the reduction operator, the possible data dependences may be overcome by code/data transformations. In the last few years various code/data transformations that parallelize irregular reduction loops appeared in the literature (see subsection on related work later). Next sections will discuss a framework to develop efficient parallelization techniques for irregular reduction loops. This framework is focused on exploiting data locality in shared–memory multiprocessor platforms.

2.1 Locality and affinity

In order to optimize data locality through code/data transformations, we first need to characterize it. Without loss of generality, let us take the reduction loop shown in Figure 1 as a working example. We can distinguish two sources of data locality: Read locality associated with accesses to read-only and privatizable arrays, and write locality associated with accesses to the reduction arrays. In (cache-coherent) shared memory multiprocessors, writes usually have

a stronger impact on performance overhead than reads (writes must propagate and serialize through the memory hierarchy). We distinguish between two classes of write locality: Intra–iteration and inter–iteration. Intra–iteration locality corresponds to write locality inside the same loop iteration. Inter–iteration locality, on the other hand, is due to writes on the reduction arrays executed in different loop iterations.

When parallelizing the reduction code, the class of locality we can exploit depends on the granularity of the parallelization method. It is usual that the minimum amount of partitionable code is one full loop iteration. In such case, only inter–iteration locality can be exploited by code parallelization. If we want to also exploit intra–iteration locality, we must resort to data reorganizations [11] (basically the contents of the subscript arrays).

A simple method to exploit inter–iteration locality proceeds in two steps: First, we fix a data distribution of the reduction arrays on all threads that cooperate in the parallel computation. Second, reduction loop iterations are assigned to threads in such a way that the number of local writes (writes to owned reduction array elements) is maximized. Note that these iteration assignments not only exploit locality but also avoid the need of run-time dependence analysis, as iterations from different threads can be executed in parallel with no write conflicts.

In what follows we will describe a framework to define efficient locality–based loop iteration assignments. We introduce a procedure that allows us to classify loop iterations according to their inter–iteration locality characteristics. The main idea behind this procedure is the concept of data affinity between iterations as a measure of the locality in histogram reduction loops. Precisely, our idea of data affinity is based on data regions written by iterations for a given data distribution. In next paragraphs a more formal definition of the affinity concept is presented.

Starting from a histogram reduction nested loop like the one in Figure 1, we consider a distribution function $\Psi : \{R[0], R[1], ...R[RDim-1]\} \rightarrow P$ that assigns the entries of the reduction array `R[RDim]` to the threads whose set of identifiers is $P = \{1, 2, ...nThreads\}$. We assume a nested loop so as each iteration is given by an iteration index vector $\vec{\imath} = (i_1, i_2, ...i_{nLoops})$. The set of all possible values for the iteration index vector is the iteration space of the loop, denoted as $\mathcal{S}$.

For one particular iteration, $\vec{\imath} = (i_1, i_2, ...i_{nLoops})$, a set of reduction array entries are written. We call this set the ***write access set*** of iteration $\vec{\imath}$, that will be denoted as $Acc_{\vec{\imath}}(R)$. More formally, the write access set of iteration $\vec{\imath}$ is composed of all the indices m such that $R[m]$ is written in this iteration, that is, $Acc_{\vec{\imath}}(R) = \{m \in [0, RDim-1] \mid R[m] \text{ is written in iteration } \vec{\imath}\}$.

Applying the defined distribution function Ψ to the write access sets provides how the writings in a iteration are mapped onto threads. This mapping is the basis of our definition of affinity. We define that two iterations $\vec{\imath}$ and $\vec{\jmath}$ are ***write affine*** if they share the same write mapping onto threads, that is, if $\Psi(Acc_{\vec{\imath}}(R)) = \Psi(Acc_{\vec{\jmath}}(R))$. If two iterations are not write affine they have a partial sharing of their mapped access sets. In an extreme case they have no sharing. In this case we can assume they do not have any common locality properties. For this fact we use the term ***write dissimilar*** iterations. In a formal way, we define that two iterations $\vec{\imath}$ and $\vec{\jmath}$ are write dissimilar if $\Psi(Acc_{\vec{\imath}}(R)) \cap \Psi(Acc_{\vec{\jmath}}(R)) = \emptyset$, that is, their writings are mapped onto disjoint sets of threads.

2.2 Write affinity based parallelization

Using the write affinity property defined in the previous section we will derive an optimal method to parallelize histogram reduction loops. Given a data distribution function of the reduction array, a code transformation of the reduction loop will be defined so as some performance issues are optimized: parallelism and data locality are maximized, and computation replication, memory overhead, extra workload and synchronization overhead are minimized.

Since the write affinity relation can be seen as a binary relation between two iterations, we can easily prove that it satisfies the reflexive, symmetric and transitive laws. Thus this affinity relation establishes an equivalence relation in the reduction loop iteration space $\mathcal{S}$. This equivalence relation allows to classify iterations into **affinity equivalence classes**. Each affinity class contains all the iterations that are mutually write affine. As the affinity relation is based on the equality of write access sets, we can select a canonical representation of each class by means of a particular access set. For a subset Q of P we represent with $\mathcal{C}_Q$ the affinity equivalence class of all the iterations whose access set is Q. More formally, $\mathcal{C}_Q = \{\vec{\imath} \in \mathcal{S} \mid \Psi(Acc_{\vec{\imath}}(R)) = Q\}$. The affinity equivalence relation makes a partition of the iteration space $\mathcal{S}$ into affinity classes. The set of all the equivalence classes is called the ***affinity quotient set***, that will be denoted as $\mathcal{S}/\text{aff}$.

When using some locality-oriented data distribution function Ψ, for example a classical block distribution, it would be possible to exploit write inter–iteration locality by considering those iterations belonging to the same affinity equivalence class. From the parallelization viewpoint, we need to select data independent reduction iterations.

Some of the properties applicable to the loop iterations can be extended to the affinity classes, for example, the dissimilarity relation. In this way we say that two affinity classes $\mathcal{C}_Q$ and $\mathcal{C}_R$ are **dissimilar** if every pair of iterations, $\vec{\imath} \in \mathcal{C}_Q$ and $\vec{\jmath} \in \mathcal{C}_R$, are write dissimilar. It is easy to check whether two affinity classes

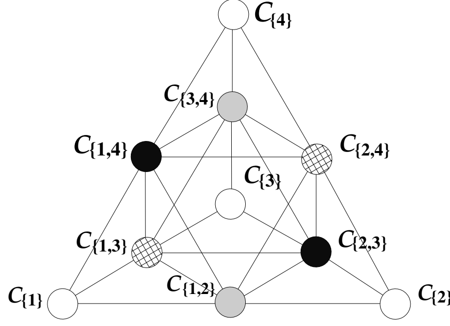

Fig. 2. Vertex coloring of the dissimilarity graph for a reduction loop with 2 indirections when 4 threads are considered

are dissimilar or not based on their canonical representation. Two classes, $\mathcal{C}_Q$ and $\mathcal{C}_R$, are dissimilar if and only if they are defined from disjoint access sets, that is, $Q \cap R = \emptyset$.

In a reduction loop the only true data dependences are caused by writes in the reduction array, thus two write dissimilar iterations are assured to be data independent. Hence iterations belonging to dissimilar equivalence classes can be executed fully in parallel, with no write conflicts. That means no parallelization overheads of any kind, like extra memory, synchronizations or computation replication. These are precisely the issues that we want to minimize in the parallelization of the reduction loop. In addition, if we can find large sets of dissimilar equivalence classes, we would have a lot of exploitable parallelism.

A representation of the dissimilarity property can be carried out by means of a ***dissimilarity graph***. The dissimilarity graph, $DG(\mathcal{S}/\text{aff}) = (N_{DG}, E_{DG})$, is an undirected graph whose vertices are the nonempty affinity classes, that is, $N_{DG} = \mathcal{S}/\text{aff}$. In this graph two vertices are connected by an edge if the corresponding classes are not dissimilar. An example of a dissimilarity graph is shown in Figure 2. In this case it has been considered four threads ($P = \{1, 2, 3, 4\}$), and two indirections inside the histogram reduction loop. As only 2 accesses to the reduction array is carried out per iteration, the classes $\mathcal{C}_Q$ are defined from subsets $Q \subset P$ with at most 2 elements. In this example, we consider that the complete set of affinity classes take part of the graph (but, in general, this is not the rule).

The dissimilarity graph relates potentially data dependent reduction iterations, for a given data distribution function. Non directly connected vertices in that graph corresponds to dissimilar equivalence classes, that contain data independent iterations. Therefore, if we want to maximize exploitable parallelism, we have to find the maximum number of non directly connected vertices in the dissimilarity graph. This can be done by applying a vertex coloring algorithm to the graph. Resulting from the coloring process sets of dissimilar classes are obtained. Iterations from the classes in each one of these sets can be executed fully in parallel because they write in non conflicting areas of the

```
for color∈coloring(DG(S/aff))
  #pragma parallel
  for C ∈ color
    Execute iterations ∈ C
  end
  #pragma barrier
end
```

(a)

```
omp_set_num_threads(nT);
for (ΔB=0; ΔB<nT; ΔB++)
  for (stg=1; stg<ΔB+1; stg++)
    #pragma omp parallel for
    for (B_min=stg;B_min<=nT;B_min+=ΔB)
    {Execute iterations∈(B_min+stg,ΔB)
    }
```

(b)

Fig. 3. (a) Parallel reduction based on affinity classes. (b) Execution phase of the DWA–LIP method

reduction array.

In the example of Figure 2 the maximum possible number of equivalence classes in the affinity quotient set was considered. Thus, the number of vertices in the dissimilarity graph is $\binom{4}{1}+\binom{4}{2} = 10$. These classes compound the affinity quotient set

$$\mathcal{S}/\text{aff} = \{\mathcal{C}_{\{1\}}, \mathcal{C}_{\{2\}}, \mathcal{C}_{\{3\}}, \mathcal{C}_{\{4\}}, \mathcal{C}_{\{1,2\}}, \mathcal{C}_{\{1,3\}}, \mathcal{C}_{\{1,4\}}, \mathcal{C}_{\{2,3\}}, \mathcal{C}_{\{2,4\}}, \mathcal{C}_{\{3,4\}}\}.$$

After applying the vertex-coloring algorithm to this graph we obtain the sets of classes than can be executed concurrently. Vertices with the same color in the graph are not directly connected. Therefore, sets of dissimilar equivalence classes can be obtained by grouping together all classes with the same color, that is:

$$\left\{ \{\mathcal{C}_{\{1\}}, \mathcal{C}_{\{2\}}, \mathcal{C}_{\{3\}}, \mathcal{C}_{\{4\}}\}, \{\mathcal{C}_{\{1,2\}}, \mathcal{C}_{\{3,4\}}\}, \{\mathcal{C}_{\{1,3\}}, \mathcal{C}_{\{2,4\}}\}, \{\mathcal{C}_{\{1,4\}}, \mathcal{C}_{\{3,2\}}\} \right\}.$$

We can schedule a parallel execution of the reduction loop following an inspector/executor scheme. An inspector builds the affinity classes, the corresponding dissimilarity graph and color it. After the inspection stage, computations are scheduled by the executor as shown in the pseudocode of Figure 3(a). Iterations in equivalence classes with the same color are executed in parallel, while a synchronization point is placed between execution of sets of classes with different colors.

2.3 Practical implementation

Although the general and theoretical approach described in the previous section could be used in parallelizing reduction loops, however some serious difficulties arise in practice. To maximize the available parallelism the minimum number of colors in the dissimilarity graph has to be found. This minimum number of colors is called the vertex-chromatic index of the graph, and it is known that a general algorithm to compute it is NP-hard. Nevertheless some simplifications can provide a near-optimal coloring with a polynomial com-

	1 indirection	2 indirections	3 indirections
2 threads	2 vertices / 1 color	3 vertices / 2 colors	3 vertices / 2 colors
4 threads	4 vertices / 1 color	10 vertices / 4 colors	14 vertices / 8 colors
8 threads	8 vertices / 1 color	36 vertices / 8 colors	92 vertices / 40 colors
16 threads	16 vertices / 1 color	136 vertices / 16 colors	696 vertices / 163 colors
20 threads	20 vertices / 1 color	210 vertices / 24 colors	1350 vertices / 233 colors

Table 1
Number of vertices and colors for the dissimilarity graph

plexity. In addition, to reduce the number of colors certain restrictions would be desirable, like maximizing the size of the equivalence classes with the same color, or considering conditions for workload balance. Such operations, however, would increase significantly the overhead of the inspection stage. Other difficulty is the fact that the number of possible non-empty affinity classes grows rapidly with the number of indirections in the reduction loop.

These difficulties are illustrated in Table 1. The number of vertices in the dissimilarity graph and its number of colors are shown for different valuess of threads and indirections. These results have been obtained by using a greedy coloring algorithm [6] with different initial vertex orders. As it is observed, the number of vertices of the graph grows fast with these two parameters. In fact, the resulting complexity considering the creation of the graph and its coloring using a low complexity heuristic like the greedy algorithm is $\mathcal{O}(nThreads^{2\,nInd})$. In practice this computational cost is very high, even more when a non optimal coloring has been used.

In order to make practical the implementation of the method in a compiler, the inspection phase must be lightened. This can be achieved by simplifying the equivalence class building process.

We have developed an approach called *Data Write Affinity with Loop Index Prefetching* (DWA–LIP) [7,8] that is based both on a block data distribution and on a restricted definition of the affinity relation. Instead of using a generic subset of threads, Q, to characterize an affinity equivalence class, $\mathcal{C}_Q$, DWA–LIP uses a pair of parameters $(B_{min}, \Delta B)$, being $B_{min} = min(Q)$ and $\Delta B = max(Q) - min(Q)$. The dissimilarity test with the new affinity relation is simpler since two iterations will be write dissimilar when their pairs $(B_{min}, \Delta B)$ do not correspond to overlapped areas of the reduction array.

The new simplified affinity relation allows the inspector to be lighter and makes possible an efficient schedule of the dissimilar classes during the exe-

cution phase. A pseudocode for this execution phase is shown in Figure 3(b). Synchronized stages based on the ΔB parameter are executed. It is observed that in each stage (loop iterated by `stg` in Figure 3(b)) a gang of dissimilar classes run in parallel. Although the method presents certain parallelism loss due to synchronizations, it does not introduce a large memory overhead, allows the inspector to be lighter and makes possible to exploit the inter–iteration locality [8].

Moreover some efficient optimizations [9] of DWA–LIP can be done in order to mitigate the parallelism loss without adding significant penalties. One of these optimizations consists of replicating the reduction arrays by a fixed ρ number of times, e.g. 2 or 4, less than the number of threads. This way we can guarantee that ρ threads may work in parallel, as they can write on different private memory spaces. This optimization, that allows to increase the exploitable parallelism with small memory overheads, is referred as *partial array expansion* [9].

2.4 Related work

Much research effort has been devoted to developing languages and compiler technologies for parallel computers. For instance, established standards, like High Performance Fortran (HPF) [4,13] and OpenMP [16], are defined as extensions of conventional languages, Fortran or C. Because numerical applications typically use irregular data structures, automatic parallelizers obtain suboptimal parallel programs. In order to increase the efficiency run-time techniques have been proposed, such as those based on the inspector-executor paradigm [17]. On other hand, reference locality is a well-known property that all programs exhibit to some extent. Assuming this property is the basis of common computer hardware design such as cache memory. The goal of such designs is to lessen the growing speed gap existing between the memory system and the processors. Locality exploitation has become a major issue to deal when obtaining a good efficiency in the execution of a code [11]. Indeed, diverse approaches to improve the locality exploitation have been explored in the literature [1,5,14,15] in the context of the parallelization of irregular codes.

Due to its importance, reduction operation parallelization has been a field where a hard effort has been done since first multiprocessors appeared. In the context of shared memory machines we can classify the existing reduction parallelization methods into two broad categories. The first one is based on the privatization of the reduction arrays and it is focused only in how to partition the iteration space among the cooperating threads. The second category is based on the partition and distribution of the reduction arrays among the threads. In this way the threads execute iterations according to the data partitioning.

Privatization–based methods [3] are the most popular methods to parallelize reduction loops especially in the automatic parallelizing world. Remember that a variable inside a loop is said to be privatizable when its use is preceded by a definition. So, iterations can be made independent making private copies of the variable. The commutative and associative properties of reductions allows to privatize reduction arrays in order to make a reduction loop parallelizable. This way, iterations become data independent (no write conflicts) allowing a free scheduling of iterations in the threads. Although several versions and optimizations of these methods were proposed [12,23], privatization-based techniques have important drawbacks, like a large extra memory requirement (reduction arrays must be replicated on all threads) and no exploitation of data locality.

Instead of distributing loop iterations, the other group of techniques uses distribution of the reduction arrays. This approach avoids the extra memory overhead discussed previously, and makes possible to take data locality into consideration. In these methods iterations are partitioned and assigned to the threads on the basis of a previously chosen data distribution for the reduction arrays. However, some specific technique must be used to solve data dependences due to write conflicts in the reduction arrays [8–10]. We will refer to these techniques as data partitioning–based methods, and in this group can be included the write affinity based parallelizations discussed previously.

A partitioning–based approach called LocalWrite [10,11] parallelizes reduction loops exploiting write locality, as in DWA–LIP. The LocalWrite inspector classifies iterations into two groups: local iterations and boundary iterations. An iteration is said to be local if it belongs to an affinity class with pair $(B_{min}, \Delta B)$ being $\Delta B = 0$. All reductions of local iterations in one particular affinity class are mapped onto the same thread. When $\Delta B \geq 1$ it is a boundary iteration. Boundary iterations carry out reductions over elements mapped onto different threads. The LocalWrite executor assigns local iterations to the threads that own the block written by them. However, this method applies a loop-splitting scheme to boundary iterations. Once boundary iterations are split they contains only one reduction and can be handled as local ones. Thus for these split iterations the computations are replicated, which implies an effective loss of parallelism. Note that the number of replications match with the number of indirect writings on the reduction array inside the loop.

3 Overview of the cloth simulation problem

Cloth simulation is an essential topic in computer animation of realistic virtual humans and dynamic sceneries. In a physically-based formulation, clothes are represented by discrete components each one numerically modeled by an ordinary differential equation (1), like

```
for (time=0; time<FinalTime; time+=timeStep()) {
    evaluateForces();
    collisionDetection();
    solveSystem();
}
```

Fig. 4. Iterative time–stepping algorithm

$$\frac{dv}{dt} = M^{-1} f(x, v), \qquad \text{where } v = \frac{dx}{dt} \tag{1}$$

This model uses a triangle mesh discretization. Force equations f contain non–linear components, that are linearized generating a linear system of algebraic equations where positions and velocities of the masses are the unknowns.

Most computations performed in a simulation process consist of loops iterating over data structures that represent the different physical elements involving such simulation (particles, triangles, forces,...). These data structures are closely related one to each other and the use of irregular meshes, needed to avoid anisotropy in the behavior of the simulated objects, gives raise to an irregular data access pattern.

Our research group has developed a non rigid material simulator based on the physical model introduced in the next section [20]. This application allows to simulate physical environments including both bulk and flat flexible objects. One of the most interesting capabilities is the realistic simulation of fabrics that can be mixed in postproduction with real scenes.

3.1 Physical model

In this section, the physical model describing the behavior of the non–rigid objects is presented as well as the associated computational model. To create animations, a time–stepping algorithm, shown in Figure 4 has been implemented. Every step is done in three phases: the `evaluateForces()` procedure creates the matrix and the vector of the equation system; the `collisionDetection()` procedure includes some constraints to the system; and the `solveSystem()` procedure is a conjugate gradient algorithm to solve it and updates the state of the system, which is made up of the position and velocity of each element.

Forces and constraints are evaluated on every discrete element (triangle). The particular forces considered are: stretch, shear, bend, gravity and air drag forces. Although explicit numerical integration methods provide accurate simulations [22], implicit methods overcome the performance of explicit ones, assuming a non visually perceptible lost of precision [2]. The implicit backward Euler method approximates (1) by $\Delta v = \Delta t \cdot M^{-1} \cdot f(x_i + \Delta x, v_i + \Delta v)$,

being $\Delta x = \Delta t(v_i + \Delta v)$. This is a non–linear system of equations which has been time–linearized by a first order Taylor expansion as follows,

$$f(x_i + \Delta x, v_i + \Delta v) = f_i + \left|\frac{\partial f}{\partial x}\right|_i \Delta x + \left|\frac{\partial f}{\partial v}\right|_i \Delta v. \qquad (2)$$

Function f represents the accumulation force vector, so every coefficient of f is the result of the sum of all different kind of forces in which the given particle is involved. This formulation gives a large system of algebraic linear equations with a sparse matrix,

$$\left(M - \Delta t^2 \frac{\partial f}{\partial x} - \Delta t \frac{\partial f}{\partial v}\right) \Delta v = \Delta t \left(f_i + \Delta t \frac{\partial f}{\partial x} v\right), \qquad (3)$$

where Δv is the unknown vector, from which the positions and velocities of the particles are computed. From here on, the system matrix will be referred as matrix `A` and the right hand side vector of the equation system will be referred as vector `b`. The building process of this equation system gives raise to several irregular reduction loops. All of them are similar to that presented in Figure 5, for every kind of force, following the same scheme shown in Figure 1.

3.2 Locality in cloth simulation

When dealing with cloth simulation, some locality features must be taken into account. In one hand, it is found a locality that is inherent to the physical problem itself. In the other hand, the code may exhibit a certain level of locality depending on the way the model is implemented. The later one is straightly related to the memory access pattern.

Given a discretization mesh in triangular finite elements of a piece of cloth, the locality inherent to the problem is clearly perceptible. To compute forces and their derivations for a given triangle only data related to its particle vertices are needed. Neighbour triangles computations share data, those related to common vertices. Moreover, most data not accessed by a given triangle is not used by a contiguous triangle. Determining the most suitable algorithm to order vertices (particles) and triangles (forces) allows to optimize the accesses to the memory hierarchy.

Data locality and computation locality referring to the different forces are the most influential aspects in the application performance. Determining an appropriate ordering of particles and forces, or finding an organization of the computations that optimizes the memory accesses, may be achieved by the analysis of the computation structures and the dependencies between the data and such computations.

```
for (cont=0;cont<numForces;cont++) {
   int i,j,k;
   double f[3][3], df[3][3][3][3];
   i=ListForce[cont][0]; /* indirections */
   j=ListForce[cont][1];
   k=ListForce[cont][2];
   f=computeForce(i,j,k);
   df=calcDerivsF(i,j,k);
   /* reductions over vector b and matrix A */
   b[i]+=f[0]; b[j]+=f[1]; b[k]+=f[2]; /* 3x1 accum. */
   /* 3x3 accum. */
   A[i][i]+=df[0][0]; A[i][j]+=df[0][1]; A[i][k]+=df[0][2];
   A[j][i]+=df[1][0]; A[j][j]+=df[1][1]; A[j][k]+=df[1][2];
   A[k][i]+=df[2][0]; A[k][j]+=df[2][1]; A[k][k]+=df[2][2];
}
```

Fig. 5. Force evaluation loop using implicit Euler integration method

In the core of the force evaluation stage (`evaluateForces()`), loops like the one presented in Figure 5 are found, which is an example of shear or stretch force. Every coefficient `f[i]` is a 3D vector, while `df[i][j]` is a 3×3 dense matrix. In each iteration, particles involved are determined, the acting force and its derivations are computed and then three 3D coefficients in the force vector `b` and nine 3×3 coefficients in the system matrix `A` are updated. When dealing with bend forces, there are four particles of two adjacent triangles, so the number of 3×3 matrix coefficients in this kind of forces are sixteen and four 3D coefficients for vector `b`.

3.2.1 Reordering strategies

In the code shown in Figure 5, the procedure `computeForce()` contains no references to the reduction arrays `b` and `A`. The subscript array `ListForce` depends on the loop index `i`, appearing in both sides in the assignment sentence, through the variables `i`, `j`, `k`. Such a pattern is precisely a *histogram* reduction. While entries in `ListForce` are visited in order, the entries in vector `b` and coefficients in matrix `A` are updated following an irregular pattern.

In the literature several reordering methods can be found [1,5,11,14,15]. As the discretization mesh is a *well–formed graph*, the use of combinatorial reordering techniques, that require a high computational load, do not add any additional benefit with respect to geometrical reordering methods in this kind of problems. For this reason, geometrical methods have been chosen. First, particles are sorted following a domain decomposition method, improving the spatial locality in the access to the entries in vector `b` (see Figure 6 (b)), and also in matrix `A`; later, list of forces are redistributed according with the previous particle ordering [19]. In this stage, temporal locality is improved, as successive iterations in the loop will use data accessed in previous iterations,

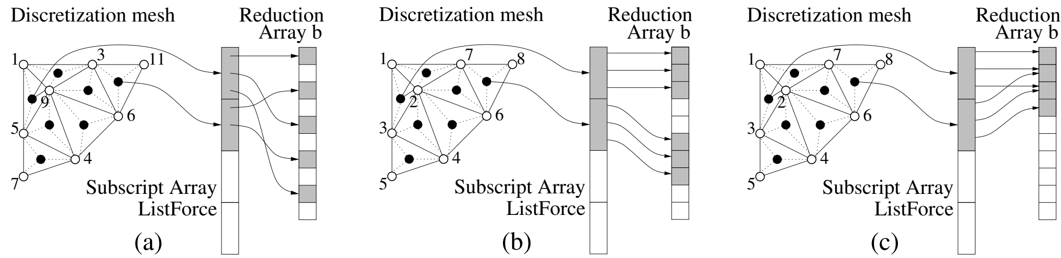

Fig. 6. Original data distribution (a); reordering particles (b), and triangles (c)

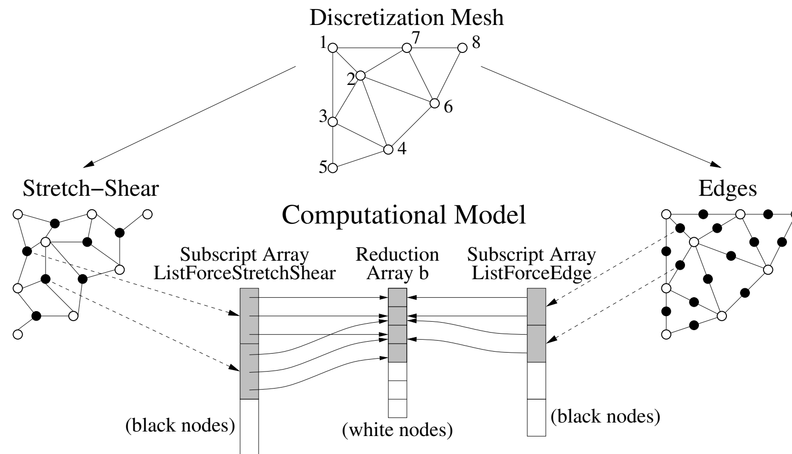

Fig. 7. Two different force loops updating the same vector `b`

as shown in Figure 6 (c).

The ordering strategy for the particles must benefit the latter reordering of the different force loops, as shown in Figure 7, which represent the way of system matrix and vector coefficients are updated. The upper graph is an example of the discretization mesh of a piece of cloth. The lateral graphs represent two sample forces, the left one is a force acting over triangles and the right one is a force acting over edges. In these graphs black nodes represent an elementary force while white nodes represent particles of the discretization. Edges between a black and a white node mean that a given force acts over this given particle.

Black nodes in the left graph links to three white nodes as triangle related forces, like stretch or shear, act over the three particles of the triangle vertices. The lower part of the Figure 7 represents how these forces are stored in data structures (`ListForce`), one array for every kind of force and one entry in each array for every elementary force. Vector `b` is an array with an entry for every particle. How force–particle relationship are coded is also shown: each array entry of elementary forces points to the positions in the vector `b` array of the involved particles.

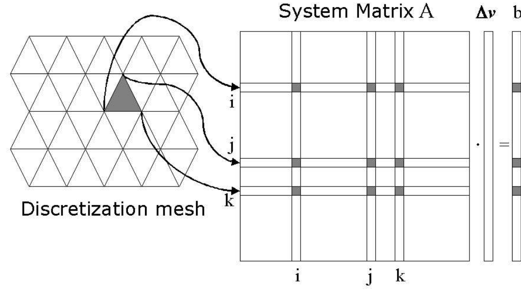

Fig. 8. Accesses to the system matrix

3.2.2 Memory accesses

Although for clarity figures 6 and 7 refer to accesses only to vector `b`, in this kind of loop, most of the memory accesses correspond to the updating of the matrix `A`. The system matrix is sparse and is stored in a compressed format. This implies another level of indirection when updating a coefficient, first locating its row and then its column. A coefficient is a 3×3 dense matrix, these nine real variables are stored in consecutive memory addresses. When a given $a_{i,j}$ coefficient of the system matrix `A` is computed, due to a force involving particles i and j, then the coefficients $a_{i,i}$, $a_{j,j}$ and $a_{j,i}$ are also computed, as well as the coefficients b_i and b_j of the *rhs* vector `b`. In the case that it refers to a triangle *(i,j,k)* related force then coefficients $a_{i,k}$, $a_{j,k}$, $a_{k,i}$, $a_{k,j}$, $a_{k,k}$ and b_k are also computed and updated, as shown in Figure 8. Moreover, in the case of a bending force, where four particles are involved, it implies 16 coefficients in four different rows of `A`. This fact means accessing coefficients in the same matrix row and also other coefficients located in different matrix rows. In order to improve access locality, it is convenient that all non–zero coefficients being located as close to the diagonal as possible. The previously described reordering strategies are particularly used to carry this transformation out.

4 Parallel irregular techniques in cloth simulation

In this section different choices to parallelize irregular reduction loops in our cloth simulation are analyzed. The discussion is focused on the parallelization of force computation stages where reduction operations are carried out in a similar way to that shown in Figure 5. The main goal is determining which parallelization strategy is more appropriate for our application in the context of shared memory NUMA machines.

Three important issues must be considered for each method in order to achieve the best execution time. The first one is the memory scalability of the technique. As commented previously, privatization–based methods exhibit a bad memory scalability because they need to replicate the reduction arrays in each thread. This fact has two negative impacts. On the one hand, writings in the reduction arrays are disseminated among their private copies and so access

locality is penalized. On the other hand, increasing the memory needs can make the application to exceed the memory limits of the machine, even more if we are dealing with large data sets.

The second issue is the ability of the technique to exploit locality. As we have discussed, codes may show some inherent locality that may influence the behavior of the memory hierarchy. Privatization–based methods do not consider at all the locality of access patterns. This fact may introduce a negative impact in performance. However, locality is taken into account by data partitioning–based methods, like DWA–LIP, that can benefit from the exploitation of inter–loop locality.

The third one is the penalty due to the parallelism loss caused by the method itself. This fact affects mainly to data partitioning–based methods, like DWA–LIP or LocalWrite. DWA–LIP suffer from parallelism loss because it executes gangs of dissimilar classes that are write conflict free. However one gang may have less members than threads. On the other hand, LocalWrite replicates computations associated to boundary iterations, which means an effective parallelism loss, too.

Next, being more specific, it is described how the reduction parallelization techniques has been implemented on our cloth simulator. We have implemented some representative methods of each class (privatization–based and partition–based techniques) and also some modifications that improve aspects of the methods.

4.1 Privatization–based methods

The well–known technique called *array expansion* [12,23] is one of the most representative techniques in this group. Array expansion makes copies (privatization) of the reduction arrays by adding an extra dimension. This new dimension is used by each thread to specify its own working private copy of the reduction arrays. Threads work on their own private copies during the execution of the assigned set of reduction loop iterations. A final reduction stage combines private reduced copies over the global reduction array. The transformed code has a similar structure to the original one, and no inspector stage is needed. Also, several loops sharing reduction arrays may use the same expanded arrays. However, they suffer from a low memory scalability due to the private copies. Other important drawback is the high computational cost associated with the final global reduction stage. The large size of the reduction arrays (matrix `A` and vector `b`) makes this final reduction stage to spend a considerable execution time. Figure 9 (a) sketchs how array expansion proceeds.

When the access pattern to the reduction arrays exhibits a high degree of locality only a small fraction of the replicas is accessed by each thread. This fact permits to avoid the replication of unused reduction array regions. This

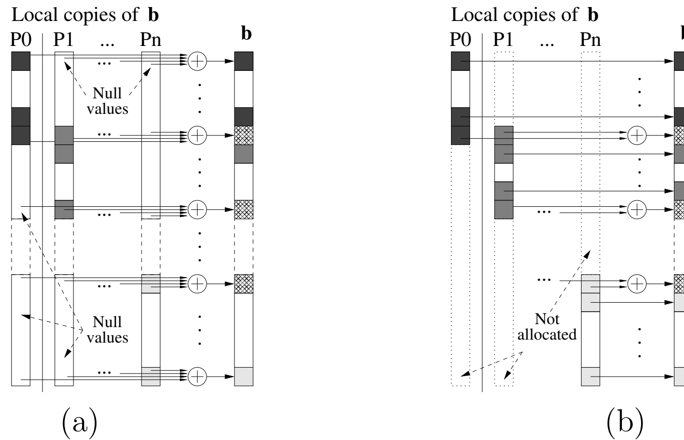

Fig. 9. Array expansion (a) and pseudo–expansion (b) methods

```
#pragma omp parallel
#pragma omp for
  for (i=0;i<numParticles;i++)
    for (id=0;id<numthreads;id++)
      b[i]=b[i]+privb[id][i];
```

(a)

```
#pragma omp parallel
  for (id=0;id<numthreads;id++)
#pragma omp for
    for (i=0;i<numParticles;i++)
      b[i]=b[i]+privb[id][i];
```

(b)

Fig. 10. Final reduction stage: (a) non optimized and (b) optimized versions

way the extra amount of memory needed by the expansion can be reduced. Nevertheless, an inspection stage to delimit boundaries of accessed regions is necessary. We will refer to this technique as *pseudo–expansion*, which is in fact a special case of *selective privatization* [23]. In Figure 9 (b) the behavior of pseudo–expansion is shown. Note that it performs better than the original array expansion as long as a good ordering of writings in the array is present.

Regarding the second drawback, the final global reduction stage, Figure 10 (a) sketches a piece of code showing how this stage is typically performed (OpenMP notation is used to phrase parallel sections). Basically, each thread is in charge of reducing a block of all private copies of the reduction arrays. In each reduction step, `id`, `i`-th entries of all private copies (`privb[]`) are reduced into the `i`-th entry of the global array `b[]`. This process is executed in sequence for all entries in the block (outermost loop).

However, it is possible to exploit more access locality by reorganizing this final stage as shown in Figure 10 (b), where both loops where interchanged. As before, all threads are assigned the same block of the private copies to be reduced. However, now these blocks are reduced in a different order. The thread starts in the block of the first private copy and reduces all entries, one by one, on the global reduction array (innermost loop). After that, the process is repeated with the same block of the second private copy. Global reduction is finished whenever all private copies are exhausted. With this strategy, the

final result is exactly the same as in the original schedule (Figure 10 (a)).

Note that this second scheduling runs sequentially over all entries of a block before going to the next one. This way spatial locality at block-level is exploited, in contrast to the first scheduling. However, some temporal locality due to updating entries in the global reduction array may be lost, although its impact in performance is lower. The net effect is that, tipically, the locality exploitation improves with the second scheme, though all this depends on the size of the blocks.

When using array expansion, any implementation of the final stage may be used. However, for pseudo-expansion it is more appropriate to use the second strategy, as no complete replicas exist. In addition, as this second version updates the global reduction array block by block, in its implementation we may use machine-optimized BLAS-like routines, which may improve performance of this stage very significantly.

4.2 Data partitioning–based methods

These techniques are based on the partitioning of the reduction arrays among threads. As reduction arrays are not replicated, these methods present a better memory scalability than privatization–based techniques, while they are able to exploit data access locality. Concerning the memory requeriments, these methods only need additional data structures for the iteration distribution. This requirement depends on the loop size, and it does not increase with the number of threads. However, an inspection stage is needed to perform the iteration distribution. The parallel work load distribution may be sensitive to the data partitioning and some load imbalance may appear. Nevertheless, in practice a good work load balance is usually achieved using block distributions.

We have applied to our cloth simulator some practical implementations of the write affinity based parallelization technique introduced in section 2, in particular DWA–LIP and LocalWrite. As these methods may exhibit a parallelism loss when the intra–loop locality is very low, the *partially expanded DWA–LIP* [9] has been also considered.

5 Experimental results

The following experimental results have been obtained from the parallelization of the force computation loop of our cloth simulator. Input data is obtained from the discretization of a piece of cloth yielding a total amount of 218272 nodes and 434829 triangles. The size of the *rhs* vector is also the number of nodes. The irregular reduction loop selected for the experimental study contains three indirections to the *rhs* vector and nine indirections to the system matrix.

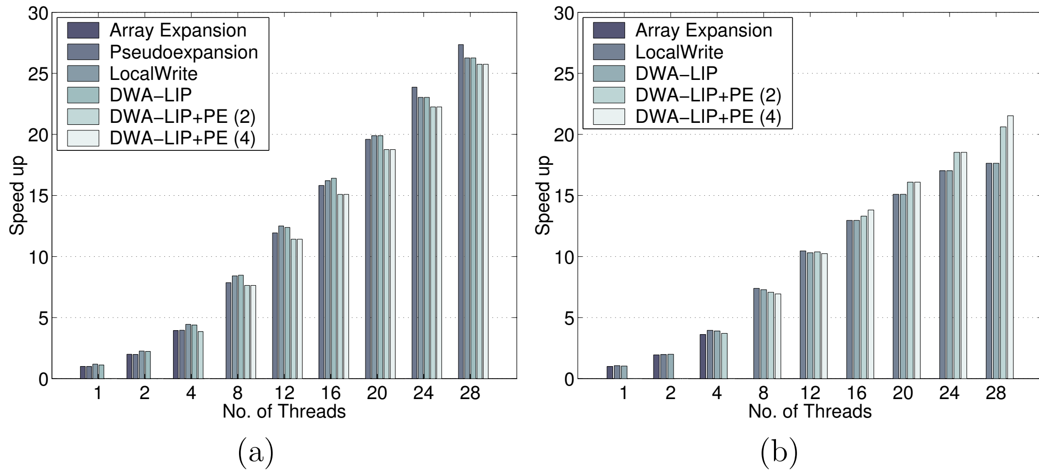

Fig. 11. Reduction loop speed–up for the sorted (a) and the non–sorted inputs (b)

The experiments have been conducted on a SGI Origin2000 multiprocessor, with 400-MHz R12000 processors (8 MB L2 cache) and 12 GB main memory. Codes were parallelized using OpenMP [16] and compiled using the SGI MIPSpro C compiler (with optimization level `O2`).

In order to analyze the impact of the input data sets reordering, we have tested two different sortings of the mesh elements. In the first one, the original order provided by the mesh generator was kept. In the second case, a geometrical reordering algorithm has been applied over the indirection vectors.

The results corresponding to the parallelization of the reduction loop using a geometrical reordering (sorted) for the input data are shown in Figures 11(a) and 12(a). As it can be observed, all methods exhibit similar speed–up. Note that results for the expansion technique for more than 4 processors cannot be obtained because of the large amount of memory requirement exceeding the machine limits. However, pseudo–expansion technique, requiring a limited auxiliary memory that do not depend on the number of threads, has been successfully used thanks to the good input data ordering. The implementation of pseudo–expansion used in these experiments does not include an inspector phase, as just replicating always and only neighbouring blocks is enough for our case study. In any case, data partitioning–based methods show a good efficiency and lower extra memory overhead. Term PE in the plots corresponds to the partially expanded version of DWA–LIP. The number after PE is the degree of partial replication (number of reduction array copies).

Figures 11(b) and 12(b) summarizes the experimental results obtained for an input data set to which no reordering strategy has been applied (non-sorted). These results are significantly different from those corresponding to sorted input data, as a lower speed–up is achieved. A first remarkable fact is that the pseudo–expansion method becomes equivalent to the array expansion, since writes are uniformly scattered among the private copies of the reduction ar-

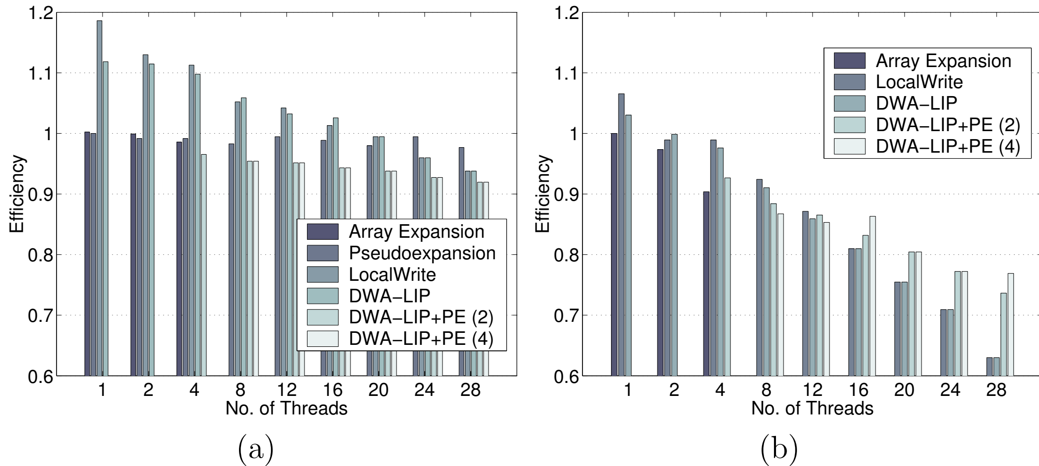

Fig. 12. Reduction loop efficiency for the sorted (a) and the non–sorted inputs (b)

rays. In this case, pseudo-expansion needs as much extra memory as array expansion. For this reason pseudo–expansion results are not displayed in Figures 11(b) and 12(b). On the other hand, data partitioning–based methods are negatively influenced by the low intra–loop locality of the non–sorted data. This low locality implies an important loss of parallelism in those methods (increase in syncrhonizations for DWA-LIP and increase in computation replications in LocalWrite). This loss of parallelism is mitigated by using partial array expansion in DWA-LIP, outperforming both, pure DWA-LIP and LocalWrite. In comparison with privatization–based methods, data partitioning–based ones present a better behavior in these cases, because they take advantage of inter–loop locality. In addition, they are not limited by the memory scalability.

Concerning memory requirements, in Figure 13 the considered methods are compared. As stated previously, privatization–based methods replicate completely or partially the reduction arrays, that are very large for our case study. In the case of the data partitioning–based methods, the auxiliary memory is used to store the data structures associated to the iteration distribution (inspector phase). In general, the required memory for these data structures is considerably smaller than for the replication of the reduction arrays as it does not grow linearly with the number of threads.

6 Conclusions

Many numerical simulation applications, like our cloth simulator, present histogram reduction computations in their core. When parallelizing these reduction loops on architectures with a complex memory hierarchy, a key goal must be locality exploitation. A framework to design efficient locality–based parallel reduction methods has been presented in this paper, based on the data affinity property.

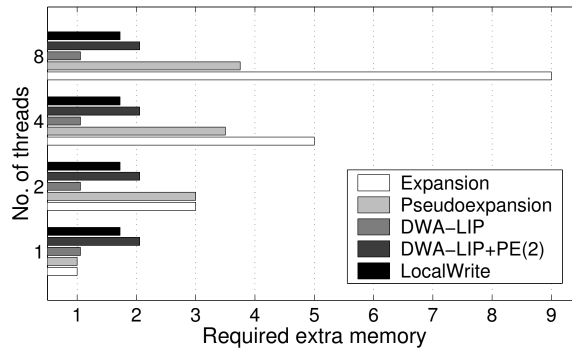

Fig. 13. Relative extra memory required by different parallelization methods

We have use this framework to select various known methods for reduction parallelization. These methods were adapted to our non–rigid material simulator and implemented, in order to compare important performance issues, like parallel execution time, locality exploitation and memory overhead. These experiments allow us to extract relevant information about what parallelization techniques are best suited to optimize numerical applications with a computational structure similar to our cloth code.

Specifically, on using extrinsic input data reorderings, privatization–based methods perform slightly better. Nevertheless their low memory scalability penalizes their execution when using a large number of threads. However, when input data is not reordered before computation, data partitioning–based methods exhibit a relatively higher efficiency.

References

[1] I. Al-Furaih, S. Ranka. Memory Hierarchy Management for Iterative Graph Structures. *International Parallel Processing Symp.*, 1998.

[2] D. Baraff and A. Witkin. Large Steps in Cloth Simulation. *ACM SIGGRAPH 98 Conf.*, 1998.

[3] W. Blume, R. Doallo, R. Eigenmann, *et al.* Parallel Programming with Polaris. *IEEE Computer*, 29(12):78–82, 1996.

[4] R. Das, R. von Hanxleden, K. Kennedy, C. Koelbel, J. Saltz. Compiler Analysis for Irregular Problems in Fortran D. *Int'l. Work. on Languages and Compilers for Parallel Computing*, 1992.

[5] C. Ding, K. Kennedy. Improving Cache Performance in Dynamic Applications Through Data and Computation Reorganization at Run Time. *ACM SIGPLAN Conf. on Programming Language Design and Implementation*, 1999.

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

[7] E. Gutiérrez, O. Plata and E.L. Zapata. On Automatic Parallelization of Irregular Reductions on Scalable Shared Memory Systems. *Lecture Notes in Computer Science*, 1685:422–429, 1999.

[8] E. Gutiérrez, O. Plata and E.L. Zapata. A Compiler Method for the Parallel Execution of Irregular Reductions in Scalable Shared Memory Multiprocessors. *ACM Int'l. Conf. on Supercomputing*, 2000.

[9] 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.

[10] H. Han and C-W. Tseng. Efficient Compiler and Run-Time Support for Parallel Irregular Reductions. *Parallel Computing*, 26(13–14):1861–1887, 2000.

[11] H. Han and C-W. Tseng. Improving Locality for Adaptive Irregular Scientific Codes. *Int'l. Work. on Languages and Compilers for Parallel Computing*, 2000.

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

[13] P. Mehrotra , J.V. Rosendale, H. Zima. High Performance Fortran: History, status and future. *Parallel Computing*, 71(3–4):325–354, 1998.

[14] J.M. Mellor-Crummey, D.B. Whalley, K. Kennedy. Improving Memory Hierarchy Performance for Irregular Applications. *ACM Int'l. Conf. on Supercomputing*, 1999.

[15] N. Mitchell, L. Carter, J. Ferrante. Localizing Non-Affine Array References. *Int'l Conf. on Parallel Architectures and Compilation Techniques*, 1999.

[16] OpenMP Architecture Review Board. OpenMP C and C++ Application Program Interface. March, 2002.

[17] R. Ponnusamy, J. Saltz, A. Choudhary, S. Hwang, and G. Fox. Runtime Support and Compilation Methods for User-Specified Data Distributions. *IEEE Trans. on Parallel and Distributed Systems*, 6(8):815–831, 1995.

[18] W.M. Pottenger and R. Eigenmann. Idiom Recognition in the Polaris Parallelizing Compiler. *ACM Int'l Conf. on Supercomputing*, 1995.

[19] S. Romero, L.F. Romero, and E.L. Zapata. Approaching Real-Time Cloth Simulation Using Paralelism. *16th IMACS World Congress 2000 on Scientific Computation, Applied Mathematics and Simulation*, 2000.

[20] S. Romero, L.F. Romero and E.L. Zapata. Fast Cloth Simulation Using Parallel Computers. *Lecture Notes in Computer Science*, 1900:491–499, 2000.

[21] K. Scloegel, G. Karypis and V. Kumar. Graph Partitioning for High Performance Scientific Simulations. *CRPC Parallel Computing Handbook.* Morgan Kaufmann, 2000.

[22] P. Volino, M. Courchesne, N. Thalmann. Versatile and Efficient Techniques for Simulating Cloth and Other Deformable Objects. *Computer Graphics*, 29:137–144, 1995.

[23] H. Yu and L. Rauchwerger. Adaptive Reduction Parallelization Techniques. *14th ACM Int'l. Conf. on Supercomputing*, 2000.