Parallelization Issues of a Code for Physically-based Simulation of Fabrics *

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

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

Abstract

The simulation of fabrics, clothes, and flexible materials is an essential topic in computer animation of realistic virtual humans and dynamic sceneries. New emerging technologies, as interactive digital TV and multimedia products, make necessary the development of powerful tools to perform real-time simulations. Parallelism is one of such tools. When analyzing computationally fabric simulations we found these codes belonging to the complex class 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. In fabric simulators these operations appear when evaluating forces, giving rise to the equation system to be solved. For this reason, this paper discusses only this phase of the simulation.

This paper analyzes and evaluates different irregular reduction parallelization techniques on ccNUMA shared memory machines, applied to a real, physically-based, fabric simulator we have developed. Several issues are taken into account in order to achieve high code performance, as exploitation of data access locality and parallelism, as well as careful use of memory resources (memory overhead). In this paper we use concept of data affinity to develop various efficient algorithms for reduction parallelization exploiting data locality.

Keywords: Fabric and cloth simulation, data locality, irregular reductions, parallelization techniques, ccNUMA multiprocessors
PACS: 07.05.Tp, 02.60.Cb

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

1 Introduction

Fabric and flexible material simulation is an essential topic in computer animation of realistic virtual humans, dynamic sceneries and computer games, among others. New emerging technologies, as interactive digital TV and multimedia products, make necessary the development of powerful tools to perform (near) real-time simulations. To reach real time each simulator stage should be optimized and executed on a high-performance platform, that frequently includes multiple processors. In such case, optimization implies parallelization as an effective tool for real-time execution.

Codes resulting from fabric simulators using a physical model are typically included in the class of irregular applications, as computations are organized using complex data structures, and data access patterns are unknown until runtime. This fact poses great difficulties to the optimization of such codes, and in particular to their parallelization.

In addition to their irregular nature, codes for this kind of applications commonly spent a significant portion of its execution time in reduction operations. These are accumulative operations based on commutative and associative operators (described in detail in Section 3.1). In our simulation codes, these operations correspond to the computation of physical magnitudes, such forces acting over fabric particles. This process determines vectors and matrix coefficients necessary to solve the differential equations that models the fabric behavior.

This paper discusses the parallelization of a physically-based fabric simulator that we have developed, focusing on those procedures that carry out irregular computations, particularly, irregular reductions. We have chosen a ccNUMA shared memory architecture as the target platform. Several issues should be taken into account in order to achieve high code performance. These factors include, in addition to parallelism, exploitation of data locality as well as careful use of memory resources (memory overhead).

We analyze the computational structure of the selected procedures of the simulator and, consequently, adapt existing irregular reduction parallelization techniques in order to obtain the maximum efficiency from the code. Among the analyzed techniques some of our own proposals are included. These proposals are derived from the concept of data write affinity that we have developed to design efficient irregular reduction parallelization methods exploiting data locality. All the studied techniques were implemented and experimentally tested, in order to obtain comparative data about efficiency and overheads.

The rest of the paper is organized as follows. Section 2 introduces the

physical foundations of our fabric simulator and also analyzes the characteristics of its force computation loops. Section 3 describes the irregular computational structure of the force loops, focusing on locality and reduction operations. Section 4 discusses the different parallelization techniques applicable to the reduction loops in the fabric simulator and their effects in performance. Also we introduce the write affinity concept for paralleliz-ing irregular reductions which exploits memory access locality. Section 5 provides an experimental evaluation of the discussed techniques. Finally, section 6 concludes the paper.

2 Overview of the fabric simulation problem

In a physical approach, fabrics and other non–rigid objects are usually represented by interacting discrete components (finite elements, springs–masses, patches) each one numerically modeled by an ordinary differential equation, as

$$\ddot{x} = M^{-1} f(x, \dot{x}), \qquad \frac{d}{dt}\begin{pmatrix} x \\ v \end{pmatrix} = \begin{pmatrix} v \\ M^{-1} f(x, v) \end{pmatrix}. \tag{1}$$

In most physically–based formulations, equations contain non–linear components, that are linearized generating a linear system of algebraic equations where positions and velocities of the masses are the unknowns. Such positions and velocities are the state of the system. Although explicit numerical integration methods provide accurate simulations [13], the work of Baraff et al. [1] demonstrates that implicit methods overcome the performance of explicit ones, assuming a non visually perceptible loss of precision. In the composition of virtual sceneries, appearance, rather than accuracy, is required. So, backward Euler implicit method has been used.

When simulating flat objects like fabrics using discrete elements, a domain decomposition method for surfaces is required. The result of the decomposition of a surface is a mesh that can be regular or irregular depending on the nature of the model. The use of regular discretization produces a non-realistic symmetrical behavior and their axes of symmetry are aligned with the discretization ones. To have control over the anisotropy in the behavior of the simulated object the use of an irregular triangular mesh is required. The use of irregular meshes (Figure 1) give raise to an irregular data access pattern. This requires an analysis of the data organization to improve the locality, thus exploiting the memory hierarchy.

Most computations performed in a simulation process consist in loops iterating over data structures that represent the different physical elements

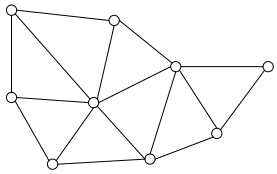


Figure 1: Example of a discretization mesh

involving such simulation (particles, triangles, forces,...). These data structures are closely related one to each other. The data distribution to obtain the best locality in a given loop may be unappropriated for other loops. Otherwise, the optimal parallelization of a given loop involves a distribution of data structures that can be against of the data distribution required in the parallelization of other loops.

Both computations and data distributions must guarantee a good locality in different levels. They must be organized in such a way that they exploit efficiently the local memory hierarchy of each processor and reduce the access cost to remote memories.

Based on these principles our research group has developed an efficient non rigid material simulator [12]. 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.

2.1 Physical model

In this section, the physical model describing the behavior of non–rigid objects is presented as well as the associated computational model. To create animations, a time–stepping algorithm has been implemented. Every step is mainly performed in three phases: computation of forces, determination of interactions and resolution of the system. The iterative algorithm is shown in Figure 2.

The `updateSystem()` procedure computes the new state of the system, which is made up of the position and velocity of each element, calculated from the previous one; `evaluateForces()` procedure creates the matrix and the vector of the equation system; `collisionDetection()` procedure includes some constraints to the system and `solveSystem()` procedure is a conjugate gradient algorithm to solve it.

```
do {
   evaluateForces();
   collisionDetection();
   solveSystem();
   updateSystem();
   time = time + timeStep();
} while(time < FinalTime)
```

Figure 2: Fabric simulation iterative algorithm

Regardless of the chosen time stepping algorithm, it would be necessary a system matrix building stage (procedure `evaluateForces()`) as long as an implicit time integration method is used. In the rest of the paper an efficient implementation of this stage is discussed, being other aspects like the stepping algorithm or the integration techniques out of the aim of this work.

2.2 Evaluating forces

Given a triangular discretization mesh, the mass of the object is shared out among the triangle vertices. The mass value of a vertex is the result of the sum of a third of the area of every triangle it is involved multiplied by the density of the material. Positions and velocities of these vertices represent the state of the system. Forces and constraints are evaluated on every discrete element in order to compute the equation coefficients for the second Newton's law. These equation coefficients depend on the state of the system, because the position of the masses determine the geometry of every triangle and the velocities determine how triangles will change. The backward Euler method approximates equation (1) by means of equation (2), being $\Delta x = \Delta t(v_i + \Delta v)$.

$$\Delta v = \Delta t \cdot M^{-1} \cdot f(x_i + \Delta x, v_i + \Delta v). \tag{2}$$

This is a non–linear system of equations which has been time–linearized by a first order Taylor expansion, obtaining the next expression,

$$f_{i+1} = f(x_{i+1}, v_{i+1}) = f_i + \left|\frac{\partial f}{\partial x}\right|_i \Delta x + \left|\frac{\partial f}{\partial v}\right|_i \Delta v. \tag{3}$$

Function f represents the accumulation force vector, so f_i is the total amount of force exerted over the particle i, and it is the result of the sum of all particular forces in which the given particle is involved.

The particular forces considered are internal: stretch, shear and bend; and external: gravity and air drag forces. In particular, internal forces are due to the internal potential energy (E). Such potential energy is produced from the deformations of the discrete components with respect to an initial rest state. Stretch energy is proportional to the variation of the dimensions of a discrete element (triangle) measured in two orthogonal axes, shear energy is proportional to the angle of the given axes, which are orthogonal in the rest state, and finally, bend energy is proportional to the angle between two neighbor triangles, initially in the same plane.

There is an analytical expression for each one of the internal potential energies. From these expressions, internal forces are analytically derived, as $f = -\partial E/\partial x$, as well as their partial derivations, appearing in equation (3). Then the code for its numerical evaluation is straightforwardly obtained. Forces derived from these energies try to recover the deformed object to its initial undeformed state, so forces are exerted over the masses or triangle vertices. These forces, proportional to a given deformation parameter, are spring–like forces. At this point the model presents a contrived gummy behavior. To improve the model, in parallel with the internal forces, some damping forces have been introduced. This way each one of the aforementioned internal forces has associated its own damping force, resulting in a realistic fabric behavior.

All above gives a large system of algebraic linear equations with a sparse matrix, as

$$\left(M - \Delta t^2 \frac{\partial f}{\partial x} - \Delta t \frac{\partial f}{\partial v}\right) \Delta v = \Delta t \left(f + \Delta t \frac{\partial f}{\partial x} v\right), \tag{4}$$

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`. A coefficient in vector `b` is by itself a 3D vector with x, y and z coefficients. Likewise a coefficient in matrix `A` is a 3×3 dense matrix because it is the Jacobian of a 3D vector. Whenever 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 the force involves a triangle *(i,j,k)* then coefficients $a_{i,k}$, $a_{j,k}$, $a_{k,i}$, $a_{k,j}$, $a_{k,k}$ and b_k are also computed

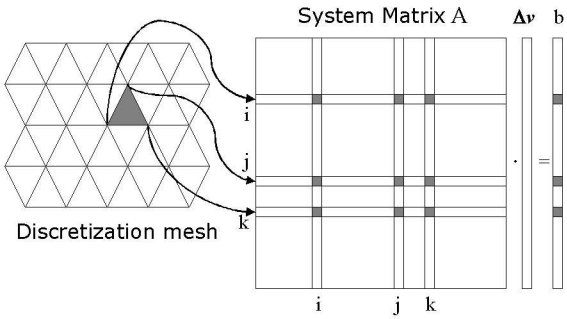


Figure 3: Accesses to the system matrix

and updated, as shown in Figure 3. Moreover, in the case of bending forces in which two adjacent triangles are involved, it implies 16 coefficients in four different rows of `A`, and four coefficients in vector `b`.

The building process of this equation system gives raise to several loops similar to that presented in Figure 4. For an internal force like stretch or shear, variable `cont` enumerates the triangles and variables `i`, `j` and `k` are indices to the triangle vertices. With the positions and velocities of the related particles, procedure `computeForce` updates the right hand side of the equation (4) through the variable `f` and the procedure `calcDerivsF` updates the related coefficients of the system matrix through the variable `df`.

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 the force vector `b` and the system matrix `A` are updated. In this loop, three 3D coefficients in vector `b` and nine 3×3 coefficients in matrix `A` are updated. These nine real variables are stored in a chunk of consecutive memory addresses, but the nine groups are scattered in three different matrix lines. In other words, locality is not assured. The total amount of data updated is 90 floating-point numbers per iteration, nine in the vector and 81 in the matrix. 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 coefficients for vector `b`, involving 156 floating-point numbers per iteration.

```
int i,j,k;
double f[3][3], df[3][3][3][3];

for (cont=0; cont<numForces; cont=cont+1)
{
   i = ListForce[cont][0]; /* related particles */
   j = ListForce[cont][1];
   k = ListForce[cont][2];
   f = computeForce(i,j,k);
   df = calcDerivsF(i,j,k);
   b[i] = b[i] + f[0]; /* rhs vector b */
   b[j] = b[j] + f[1]; /* 3D accumulations per line */
   b[k] = b[k] + f[2];
   A[i][i] = A[i][i] + df[0][0]; /* system matrix A */
   A[i][j] = A[i][j] + df[0][1]; /* 3x3 accum. per line */
   A[i][k] = A[i][k] + df[0][2];
   A[j][i] = A[j][i] + df[1][0];
   A[j][j] = A[j][j] + df[1][1];
   A[j][k] = A[j][k] + df[1][2];
   A[k][i] = A[k][i] + df[2][0];
   A[k][j] = A[k][j] + df[2][1];
   A[k][k] = A[k][k] + df[2][2];
}
```

Figure 4: Force evaluation loop using implicit Euler integration method

3 Irregular properties of the simulator code

The aforementioned force computation procedure belongs to the class of codes known as irregular. This property is found commonly in many data organizations used in numerical applications derived from models based on irregular domain discretization methods. Irregular codes are characterized by the way in which memory access patterns are carried out. Such accesses to data are not performed directly but by means of indirections. Approaching the optimization, like parallelization, of irregular codes involves two main difficulties.

On the one hand, irregular memory accesses are unknown until the program is executed. The reason for this fact is that indirection references are typically either computed during execution or read externally from a

file. In our simulator, these indirections correspond to the description of the discretization mesh which only depends on the particular fabric to be simulated. Such indirect accesses appearing in the force computation loops are a consequence of the accumulation of different magnitudes. Although, in general, indirections may prevent code optimizations, this accumulation operation allows us, precisely, to deal with irregular indirect accesses as it is described in the next subsection.

On the other hand, irregular codes may exhibit low memory access locality due to the use of indirections. The indirect access patterns are determined by both the discretization mesh topology and the refinement process used when this mesh is generated. Thus, a bad arrangement in indirections given by the discretization yields memory accesses with poor locality. This lack of locality may have a negative effect on performance, specially if a shared memory NUMA architecture is used as the target machine. Nevertheless, due to the inherent locality associated to the physical problem, it is possible to exploit to some extent such locality following certain strategies. All these aspects are discussed in subsequent subsections.

3.1 Irregular reductions

As observed in Figure 4 irregular memory accesses appearing in the force computation loop are associated to accumulation operations, and the same indirection is present both on the left hand side and on the right hand side of assignments. Such scheme of accumulation operations belongs to an important class of operations named reductions. By definition, a reduction operation is 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 histogram reduction is used. A simplified histogram reduction loop is shown in Figure 5. 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 $\mathtt{s}_1$, $\mathtt{s}_2$, ... Note that entries in the indirection (subscript) arrays are visited in order, while entries in the reduction array are written following an irregular pattern. Functions ξ_1, ξ_2, ... represent the actual computation and they must not include references to the reduction array. In practice these functions share some common calculations which are usually computed before the reduction sentences inside the loop body. Additionally indirection arrays must keep unchanged during the execution of the entire

```
double R[RDim];
int  s_1[N], s_2[N], ... s_r[N];

for (i=0; i<N; i=i+1)
{
     R[s_1[i]] = R[s_1[i]] + ξ_1();
     R[s_2[i]] = R[s_2[i]] + ξ_2();
        ...
     R[s_r[i]] = R[s_r[i]] + ξ_r();
}
```

Figure 5: A generic loop with multiple irregular (histogram) reductions

reduction loop.

Returning to the force computation code (Figure 4) two reduction arrays are found, matrix `A` and vector `b`, being used the addition as the reduction operator. `ListForce` acts as an indirection (subscript) array, which is accessed directly by the loop index `cont`. The above two reduction arrays are accessed indirectly by using `ListForce`.

Analyzing memory references in the above loop we observe that loop–carried data dependences may be present, due to the use of a subscript array (`ListForce`), that prevents its parallelization. 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, as will be discussed in following sections. First, however, locality properties of our loop code are analyzed.

3.2 Locality properties of the fabric simulator

In order to achieve high performance from the execution of the fabric simulator, some locality properties must be taken into account. On one the hand, the simulator exhibits locality that is inherent to the physical problem itself. On the other hand, the code of the simulator may exhibit a certain level of locality depending on the way the model is implemented. The latter one is straightly related to the memory access pattern.

Given a discretization mesh in triangular finite elements of a piece of

fabric, the locality inherent to the physical problem is clearly shown. To compute forces and their derivations for a given triangle only data related to its particle vertices is needed. Neighbor 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 memory accesses, may be achieved by the analysis of the computation structures and the dependencies between the data and such computations.

With the aim of obtaining a good cache memory performance, accesses to data stored in cache must be maximized (*temporal locality*), while all data laying in the same cache line must be also accessed (*spatial locality*). A study of the intrinsic locality of the problem allows to establish a mechanism to increase the data locality.

Without loss of generality, let us take the reduction loop shown in Figure 5 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 than reads (writes must propagate and serialize through the memory hierarchy). We distinguish between two types 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 to 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 a 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 [8] (basically the contents of the subscript arrays).

As it is inferred from Figure 3 several coefficients of different rows are accessed in each loop iteration. Thus improving intra–iteration locality will imply that all non–zero coefficients being located as close to the diagonal as possible. The narrower diagonal band we get the better intra–iteration locality we could exploit.

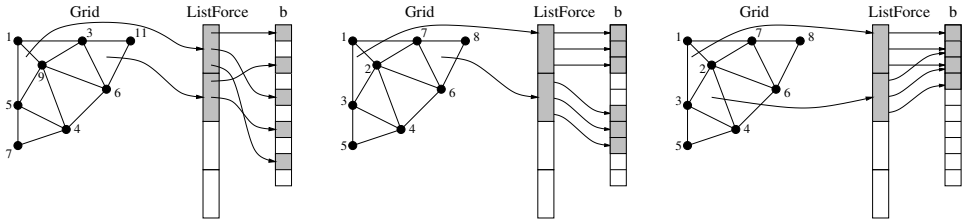


Figure 6: Original data distribution (a), reordering particles (b), and reordering triangles (c)

A way to achieve this goal could be the use of an extrinsic reordering method. First, particles are sorted following a domain decomposition method, improving the spatial locality when accessing entries of vector `b` (see Figure 6 (a) and (b)), and also of matrix `A`. Later, list of forces are redistributed according with the previous particle ordering [11]. In this stage temporal locality is improved, as successive iterations in the loop will use data accessed in previous ones, 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 represents the way of system matrix and vector coefficients are updated. The upper graph is an example of the discretization mesh of a piece of fabric. 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 link 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 represents how these forces are stored in data structures (`ListForce`), one array for every kind of force and one element in each array for every elementary force. Vector `b` is an array with an entry for every particle. How force–particle relationships are coded is also shown. Each array entry of elementary forces points to the positions in the vector `b` of the involved particles.

Although for clarity reasons Figures 6 and 7 include accesses only to vector `b`, however, in this loop, most of the memory accesses correspond to the updating of the matrix `A`. The system matrix is sparse and it is stored in a compressed format. This implies another level of indirection when updating a coefficient, first locating its row and then its column.

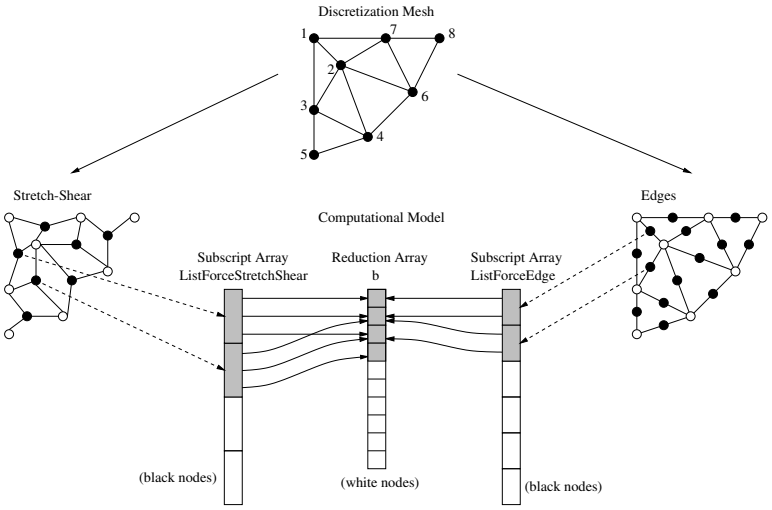


Figure 7: Two different force loops updating the same vector `b`

4 Irregular reduction parallelization

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 reduction parallelization methods, found in the literature, into two broad categories. The first one is based on the privatization of the reduction arrays and it is focused only on 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 how data was assigned.

In this section different choices to parallelize irregular reduction loops in our fabric 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 4. 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. Privatization–based methods exhibit a bad memory scalability because they need to replicate the reduction arrays in each thread. This way, the extra memory requirement grows up with the number of threads.

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 partitioning–based methods 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 partitioning–based methods. It is usual in such methods, in order to exploit data locality, to either introduce some extra synchronization points inside the parallel code, or to replicate computations in some portions of the parallel code. Both situations imply certain amount of parallelism loss.

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

4.1 Privatization–based methods

Privatization–based methods [2] are the most popular methods to parallelize reduction loops, specially 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 [9, 15], 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.

The well–known technique called *array expansion* [9, 15] is one of the most representative techniques in this group. Array expansion makes copies

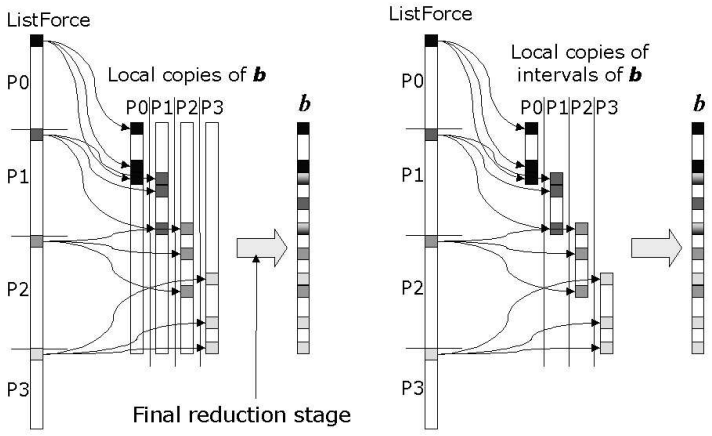
(privatization) of the reduction arrays by adding an extra dimension. This new dimension is used by each thread to specify its own working private copy of the reduction arrays. Threads work on their own private copies during the execution of the assigned set of reduction loop iterations. A final reduction 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`) in our fabric simulator make this final reduction stage to spend a considerable execution time. Figure 8 (a) sketches 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 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* [15]. In Figure 8 (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 on the reduction array is present.

Regarding the second drawback, the final global reduction stage, Figure 9 (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`, the `i`-th entry for 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). Looking into this process from the locality point of view, a total of `numthreads` consecutive writes are executed in the same entry of `b`, so we have a high temporal locality in accessing `b`. However, no locality is exploited while reading `privb`, which is distributed among threads.

It is possible to exploit access locality in a different way by reorganizing this final stage as shown in Figure 9 (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



(a) (b)

Figure 8: Array expansion (a) and pseudo–expansion (b) methods

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 9 (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 is lost. Loosing temporal write locality may be a performance problem in ccNUMA multiprocessors, as it may increase the invalidation rate for duplicated pages. The net effect is that, typically, performance is slightly better in the second scheme, though this fact 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 improve performance of this stage very significantly.

```
#pragma omp parallel
#pragma omp pfor
   for (i=0; i<numParticles; i=i+1)
      for (id=0; id<numthreads; id=id+1)
         b[i] = b[i] + privb[id][i];
```

(a)

```
#pragma omp parallel
   for (id=0; id<numthreads; id=id+1)
#pragma omp pfor
      for (i=0; i<numParticles; i=i+1)
         b[i] = b[i] + privb[id][i];
```

(b)

Figure 9: Two version for the final reduction stage

4.2 Partitioning–based methods

Instead of distributing loop iterations, other group of techniques use partitioning of the reduction arrays among threads. As reduction arrays are not replicated, these methods present a better memory scalability than privatization–based techniques, as well as they are able to exploit data access locality. Concerning the memory requirements, 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.

In what follows we will introduce the idea of data write affinity that allows us to develop efficient locality–based reduction parallelizations, as well as to analyze the properties of existing methods in this group.

4.2.1 Parallelization based on write affinity

In general, a partitioning–based parallelization method will proceed in two steps: First, we fix a 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 must be done in a write conflict–free way and one of our aims is to be able to exploit the existing locality.

In what follows we will introduce an approach that allows us to classify loop iterations according to their inter–iteration locality characteristics. The main idea behind this approach is the concept of data affinity between iterations as a measure of the locality in histogram reduction loops. This affinity definition is based on data regions written by iterations for a given data distribution.

Starting from a histogram reduction loop, like the one in Figure 5, we consider a distribution function $\Psi : \{0, 1, ...RDim-1\} \rightarrow P$ that assigns the entries of a reduction array `R[RDim]` to the threads whose set of identifiers is $P = \{1, 2, ...nThreads\}$. For a particular iteration i, we define its *write access set*, denoted as $Acc_i(R)$, as the set of all the indices m such that $R[m]$ is written in that iteration i.

We say that two iterations i and j are *write affine* if they share the same write mapping onto threads, that is, if $\Psi(Acc_i(R)) = \Psi(Acc_j(R))$. Similarly, we use the term *write dissimilar* whenever two iterations write to different sections of the reduction array, that is, if $\Psi(Acc_i(R)) \cap \Psi(Acc_j(R)) = \emptyset$. Note that write dissimilar iterations can be executed fully parallel because they write in non conflicting areas of the reduction array.

We have developed an approach for write affinity-based parallelization of reduction loops, called *Data Write Affinity with Loop Index Prefetching* (DWA–LIP) [4, 5]. DWA–LIP is based both on a block data distribution and on a specific definition of the affinity relation. Each iteration is associated with a pair of parameters $(B_i^{min}, \Delta B_i)$, being $B_i^{min} = min(\Psi(Acc_i(R)))$ and $\Delta B_i = max(\Psi(Acc_i(R))) - min(\Psi(Acc_i(R)))$. Two iterations with the same pair are write affine. The dissimilarity test, in this case, is simple since two iterations will be write dissimilar when their pairs $(B_i^{min}, \Delta B_i)$ do not correspond to overlapped areas of the reduction array.

The aforementioned affinity relation allows the inspector to be light and makes possible an efficient schedule of the execution of dissimilar iterations during the execution phase. A pseudocode for this execution phase is shown in Figure 10. Synchronized stages based on the ΔB_i parameter are exe-

```
for (ΔB_i=0; ΔB_i<nThreads; ΔB_i=ΔB_i+1)
{
   for (stage=1; stage < ΔB_i+1; stage = stage + 1)
   {
      #pragma parallel for
      for (B_i^min=stage; B_i^min<=nThreads; B_i^min = B_i^min + ΔB_i)
      {
           Execute iterations with pair (B_i^min + stage, ΔB_i)
      }
      #pragma barrier
   }
}
```

Figure 10: Execution phase of the DWA–LIP method

cuted. It is observed that in each stage a gang of dissimilar iterations 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 light and makes possible to exploit the inter–iteration locality [5]. Moreover some efficient optimizations [6] of DWA–LIP can be done in order to reduce the parallelism loss without adding significant penalties.

Another partitioning–based approach found in the literature is called LocalWrite [7]. It parallelizes reduction loops exploiting write locality, as in DWA–LIP. However, this method is based on applying loop-splitting to those iterations belonging to affinity classes with pairs $(B_i^{min}, \Delta B_i)$, $\Delta B_i \geq 1$. 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.

As these methods may exhibit a parallelism loss when the intra–loop locality is very low, an optimized version of DWA–LIP has also been considered. This optimization consists of replicating partially the reduction array by a fixed number of times, e.g. 2 or 4, less than the number of threads. This partial expansion allows to mitigate the parallelism loss with no excessive memory expenses. This technique will be referred to as *partially expanded* DWA–LIP [6].

5 Experimental evaluation

The following experimental results have been obtained from the parallelization of the force computation loop of the discussed fabric simulator. This loop corresponds to the code shown in Figure 4, which is the body of the `evaluateForces()` procedure (see Figure 2). In such a reduction loop the system matrix `A` and the right hand side vector `b` of the equation system to solve are built. Input data is obtained from the discretization of a piece of fabric yielding a total amount of 218272 nodes, 653100 edges and 434829 triangles. The system matrix is a symmetric squared sparse matrix whose size is in the order of the number of nodes with a sparsity factor of 0.015%. The size of the *rhs* vector is also the number of nodes. The irregular reduction loop contains three indirections to the *rhs* vector and nine indirections to the system matrix.

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 a shared memory model, with the aid of OpenMP [10] directives for synchronization purposes, and compiled using the SGI MIPSpro C compiler (with optimization level `O2`). A Block-Jacobi preconditioner has been used for the CG solver, and it has been parallelized by using the owner's rule for the data distribution of the particles. Synchronization directives has been carefully placed in the iterative part of CG to minimize the overhead due to load imbalance [16].

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 (see section 3.2).

Reduction parallelization methods of both categories, based on privatization and partitioning, have been implemented. In the first category array expansion and pseudoexpansion techniques have been chosen. In both cases the optimized code shown in Figure 9(b) for the final reduction stage has been used, because it provides a better cache behavior, as it is shown in table 1. From the other category, DWA–LIP and LocalWrite techniques have been selected. Additionally, partially expanded DWA–LIP have been tested for the non sorted input data set case, since it exhibits low locality.

The results corresponding to the parallelization of the reduction loop using an extrinsic reordering for the input data set are shown in Figure 11. Both, speed-up and efficiency, are calculated taken the array expanded code executed in one thread as the reference. So, all methods are compared

Cache miss rate		
#Threads	(a)	(b)
1	8,33%	8,33%
2	6,25%	8,33%
4	34,38%	8,33%
8	33,85%	8,33%
16	33,59%	8,33%
32	33,46%	8,33%

Table 1: Cache behavior simulation for the final reduction stage codes shown in Figures 9(a) and (b)

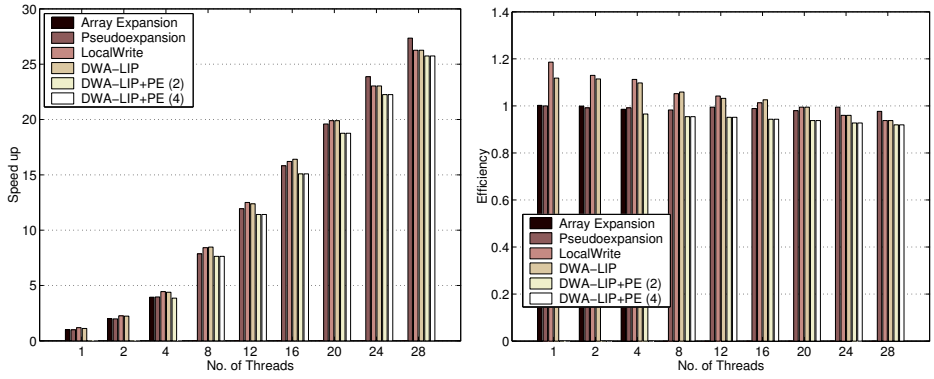


Figure 11: Reduction loop parallelization speed–up and efficiency for the sorted input data set

against array expansion (which is the classical technique to parallelize reduction loops). 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, the pseudo–expansion technique, requiring a limited extra 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 neighboring blocks is enough for our case study. In any case, 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).

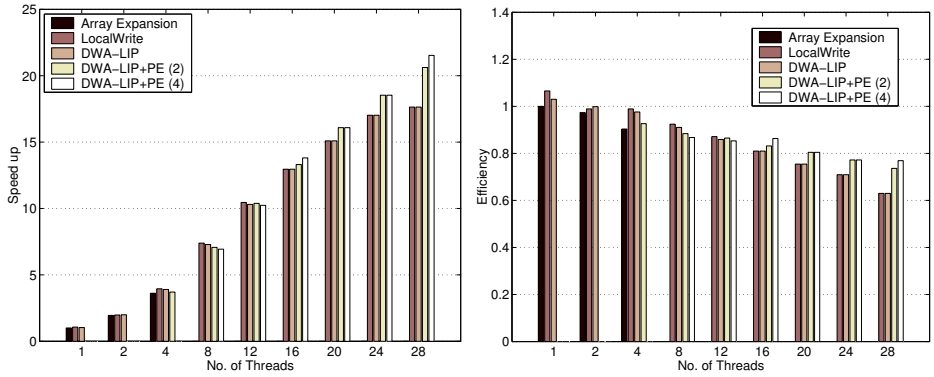


Figure 12: Reduction loop parallelization speed–up and efficiency for the non–sorted input data set

Figure 12 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 in Figure 11, 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 scattered among the private copies of the reduction arrays. That implies that array expansion does not need more extra memory than pseudo–expansion. For this reason pseudo–expansion results are not displayed in Figure 12. On the other hand, partitioning–based methods are negatively influenced by the low intra–loop locality of the non–sorted data. In comparison with privatization–based methods, 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 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.

In order to illustrate the simulator capabilities the output for two experiments is shown in Figure 14. At the top of the figure the three frames match with three simulation time–steps of a piece of fabric falling down over a table until it reaches the minimum energy state. The other three frames

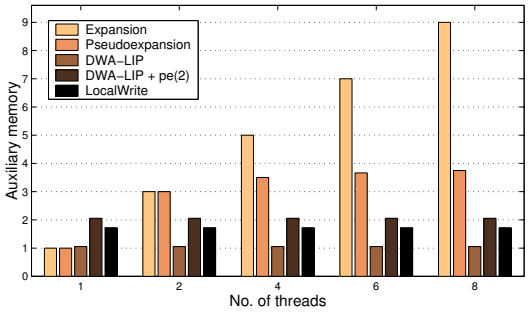


Figure 13: Required memory for different parallelization methods (normalized to the sequential code)

display the simulation of the behavior of a hanged fabric interacting with a cylinder.

The model for both fabrics is composed of 2602 particles and 4982 triangles. A total of 76 time-steps of 33 milliseconds each were simulated (that is a simulation of around 2,5 seconds). For the first cloth (top part of Figure 14) the total execution time in one processor of the aforementioned SGI Origin2000 machine is 42,07 seconds, which is split it up into 4,56 seconds for the `evaluateForces()` procedure, 7,01 seconds for the `collisionDetection()` procedure, and 30,5 seconds for the `solveSystem()` procedure. For the second cloth, on the other hand, the total execution time is 43,3 seconds (4,42 seconds for `evaluateForces()`, 3,88 seconds for `collisionDetection()`, and 30,0 seconds for `solveSystem()`). As an example of parallel execution, the first fabric takes 6,33 seconds using 12 processors.

6 Conclusions

Among the available techniques to optimize applications that requires (near) real-time, parallelism is a good candidate. Fabric simulation is one of these cases, as interactive and multimedia products (dynamic virtual sceneries, computer games, ...) requires tight time constraints.

In the computational core of our fabric simulator irregular reductions are found, where a significant portion of the complete execution time is spent. This situation leads us to search efficient solutions to parallelize this kind of operations.

In this paper we have adapted some of the solutions found in the literature for irregular reduction parallelization, including our own proposals.

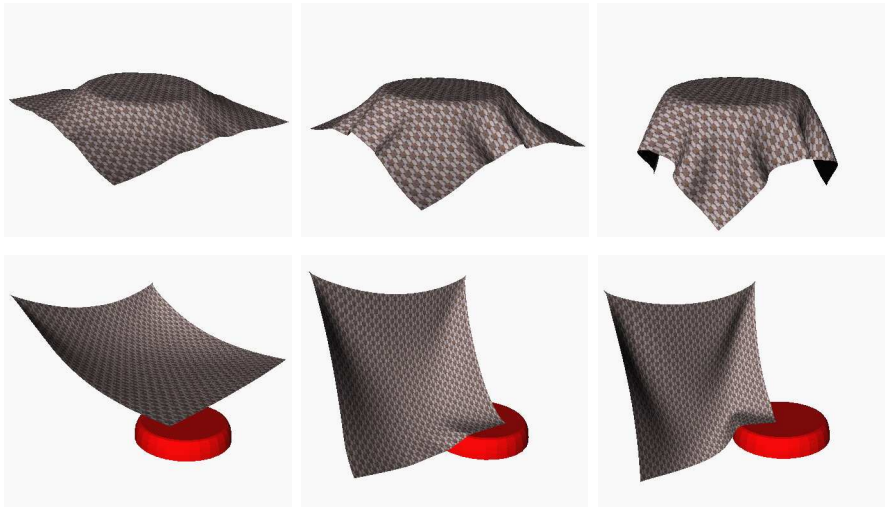


Figure 14: Tablecloth and hanged fabric simulations

In order to get maximum performance, issues such as data locality, memory overhead and load balancing, in addition to parallelism, were taken into account. Our proposals were designed using a framework based on the data affinity property, allowing to exploit locality features found in irregular memory access patterns.

We have carried out experiments to compare the analyzed solutions and get actual performance data. In such experiments we included extrinsic data reordering as an additional issue of analysis. Our experiments conclude that data locality is a critical issue to obtain the needed efficiency, and must be exploited either externally (by a reordering algorithm) or at runtime by the reduction parallelization method. Also, methods in the last option have the additional advantage of a good memory scalability.

References

[1] D. Baraff and A. Witkin. Large steps in cloth simulation. In Michael Cohen, editor, *SIGGRAPH 98 Conference Proceedings*, Annual Conference Series, pages 43–54. ACM SIGGRAPH, Addison Wesley, July 1998. ISBN 0-89791-999-8.

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

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

[4] E. Gutiérrez, O. Plata and E.L. Zapata. On Automatic Parallelization of Irregular Reductions on Scalable Shared Memory Systems. Lecture Notes in Computer Science (LNCS) no. 1685, Springer-Verlag, Berlin, 1999, pp. 422–429.

[5] 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 (ICS'2000)*, pp. 78–87, 2000.

[6] E. Gutiérrez, O. Plata and E.L. Zapata. Improving Parallel Irregular Reductions Using Partial Array Expansion. *IEEE/ACM Int'l. Conf. for High Performance Computing and Communications (SC'2001)*, 2001.

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

[8] H. Han and C-W. Tseng. Improving locality for adaptive irregular scientif codes. *Int'l. Workshop on Languages and Compilers for Parallel Computing (LCPC'2000)*, pp. 173–188, 2000.

[9] 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)*, 1998.

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

[11] S. Romero, L.F. Romero, and E.L. Zapata. Approaching real-time cloth simulation using parallelism. In *Proceedings of 16th IMACS World Congress 2000 on Scientific Computation, Applied Mathematics and Simulation*, 2000.

[12] S. Romero, L.F. Romero, and E.L. Zapata. Fast cloth simulation using parallel computers. In *Lecture Notes in Computer Science, vol. 1900, pp.491–499*, 2000.

[13] P. Volino, M. Courchesne, N. Thalmann. Versatile and efficient techniques for simulating cloth and other deformable objects. Computer Graphics, 29 (Annual Conference Series):137–144, 1995.

[14] Kirk Scloegel, George Karypis, and Vipin Kumar. *CRPC Parallel Computing Handbook*, chapter Graph partitioning for high Performance Scientific Simulations. Morgan Kaufmann, 2000.

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

[16] E.M. Ortigosa, L.F. Romero and J.I. Ramos. Parallel scheduling of the PCG method for banded matrices arising from FDM/FEM. *Journal of Parallel and Distributed Computing*, 2003. 63:1243–1256.