Efficient Parallel Implementations for Surface Subdivision

E. J. Padrón¹ M. Amor¹ M. Bóo² R. Doallo¹

¹Dept. Electronic and Systems, University of A Coruña, E–15071 A Coruña, Spain
²Dept. Electronics and Computer Eng., University of Santiago de Compostela, E–15782 Santiago de Compostela, Spain

Abstract
Achieving an efficient surface subdivision is an important issue today in computer graphics, geometric modeling, and scientific visualization. In this paper we present two parallel versions of the Modified Butterfly algorithm. Both versions are based on a coarse-grain approach, that is, the original mesh is subdivided into small groups and each processor performs the triangles subdivision for a set of groups of the mesh. First approach sorts the groups in decreasing order of number of triangles per group, and then the sorted groups are cyclically distributed on the processors in order to achieve a good load distribution. In the second parallel version the processors can dynamically balance the work load by passing groups from heavier loaded processors to lighter ones, achieving in that way a better load balance. Finally, we evaluate the algorithms on two different systems: a SGI Origin 2000 and a Sun cluster. Good performances in terms of speedup have been obtained using both static and dynamic parallel implementations.

1. Introduction
Smooth curves and surfaces must be generated to satisfy a growing expectation of realism in many computer graphics applications: geometric modeling, scientific visualization, and virtual reality, among others. Since computer graphics hardware accelerates the display of triangle meshes, subdivision algorithms for obtaining detailed meshes from coarse meshes are frequently employed. This way, that feature (surface subdivision) has been recently introduced both in DirectX 8 and OpenGL (N-Patches and PN Triangles, respectively). As an example, TRUFORM ⁶, a new technology developed by ATI Technologies, implements the surface subdivision in the graphics chip using Bezier surfaces ¹⁰. This approach is based on the tessellation of each triangle by means of the local information of each edge. Smoother surfaces can be obtained if the neighbor information is employed for the subdivision procedure ³ ⁴. Recursive subdivision schemes ⁹ are well suited for this purpose because they are easy to implement and computationally efficient. A recursive subdivision algorithm generates a smooth mesh from a coarse mesh as the limit of a sequence of successive refinements with a fixed set of subdivision rules. In general, these subdivision techniques can be categorized into two distinct classes: approximating and interpolating schemes.

Approximating schemes for arbitrary topology meshes are typically modifications of spline based schemes. This way, the vertices of the original mesh do not belong to the final computed mesh. A representative approximating scheme for triangle meshes is the Loop algorithm ⁵. On the other hand, for interpolating schemes the vertices of the original mesh are also points of the final surfaces. The most well–known interpolation-based subdivision scheme is the Butterfly algorithm proposed by Dyn et al. ⁴. This algorithm exhibits degeneracies when it is applied to a topologically irregular grid. This problem was overcome with the Modified Butterfly algorithm proposed by Zorin et al. ¹² ¹¹.

The Loop algorithm achieves high quality surfaces and fast convergence speed, while the Modified Butterfly algorithm can be considerably simplified, and many calculations can be performed in place. Moreover, interpolating schemes match the original data exactly, and are crucial for fast multisolution and wavelet techniques. From the computational point of view both subdivision algorithms, Loop and Modified Butterfly, are based on the same basic idea in which each edge is subdivided employing its neighbor information.

The main disadvantage of these subdivision schemes is their high memory requirements when finer meshes are considered. The number of mesh vertices and computations grows by a constant factor from iteration to iteration. Processing of a high detailed mesh in real time could be
achieved by its execution on a set of processors. This feature is definitely useful in a lot of computer graphics engines - including global illumination methods as radiosity - as a part of the rendering process. Thus, the original coarse mesh would be partitioned and, as a result, the smooth mesh calculated on each processor would be sent to the application in order to continue the rest of the rendering process.

In this paper, we consider the parallelization of surface subdivision algorithms for triangular meshes. Specifically, we have used the Modified Butterfly algorithm although similar results could be obtained using the Loop subdivision algorithm and, in general, with all those subdivision algorithms, including adaptive subdivision strategies \textsuperscript{1,3}, in which the first order neighbor information is employed.

We have used a novel mesh partitioning algorithm to subdivide the coarse mesh into smaller groups. This grouping algorithm \textsuperscript{1,3} covers the full mesh in an efficient way, produces a balanced number of triangles per group, and reduces the number of boundary triangles per group. Load balancing among processors together with the reduction of the communications are algorithm design goals. Thus, here we propose two different parallel implementations on distributed memory systems: on the one hand a static and simple version, with hardly any communication, and on the other hand a dynamic version, which implements a load balancing scheduling. Experimental results have been obtained using two different systems; a SUN cluster and a SGI Origin 2000. On both machines we have achieved good results, and algorithm scalability is shown.

This work is organized as follows: in Section 2 we introduce the Modified Butterfly scheme for subdividing triangle meshes; in Section 3 we present the mesh grouping algorithm; the parallel implementations are described in Section 4, and experimental results on our target machines are shown in Section 5; finally, in Section 6 we present the conclusions.

2. The Modified Butterfly Algorithm

In this section we briefly describe the Modified Butterfly algorithm \textsuperscript{11}. In this method the triangulation scheme is refined through splitting every edge into two parts and reconnecting. In each iteration all the edges in the mesh are split and four new triangles for each original one are introduced. The location of a new vertex is obtained according to specific rules and it is conditioned by the weighted average of the first order neighbor vertices, though extensions to second order neighbors have also been proposed \textsuperscript{8}.

According to its valence we can classify vertices as regular vertices (valence of 6, so the vertex is shared by 6 triangles) and extraordinary vertices (valence not equal to 6). This way, four different subdivision situations have to be considered:

1. The edge connects two regular vertices. The mask (graphical representation showing the vertices employed in the computation of a new point) to be employed is depicted in Fig. 1.a. The new point, \( V \), is computed through the weighted sums of neighbors’ values in this way:

\[
V = \frac{1}{2}(a + b) + \frac{1}{8}(c + d) - \frac{1}{16}(e + f + g + h) + 0(i + j)
\]

2. The edge connects an extraordinary vertex of valence \( K \) and a regular vertex. The mask to be employed for the extraordinary vertex is indicated in Fig. 1.b. The weights of the vertices \( w_{ij} \) (with \( 0 \leq j < K \)) used in the linear combination rule are in this case:

\[
K = 3 \Rightarrow \begin{cases} 
  w_{s0} = 5/12 \\
  w_{s1} = -1/12 \\
  w_{s2} = -1/12 
\end{cases}
\]

\[
K = 4 \Rightarrow \begin{cases} 
  w_{s0} = 3/8 \\
  w_{s1} = 0 \\
  w_{s2} = -1/8 \\
  w_{s3} = 0 
\end{cases}
\]

\[
K \geq 5 \ (K \neq 6) \\
 w_{s0} = \frac{1/4 + \cos(2\pi j/K) + 1/2 + \cos(4\pi j/K)}{K}, \ 0 \leq j < K
\]

3. The edge connects two extraordinary vertices. In this case we take the average of the values computed using the appropriate scheme from the previous case for each end-point.

4. Boundary edges are subdivided using the 1-dimensional 4 point scheme of Fig. 1.c. This scheme was proposed to avoid cracking effects between contiguous meshes. Thus, the new point will be computed as follows:

\[
V = \frac{9}{16}(a + b) - \frac{1}{16}(c + d)
\]
3. Mesh Partitioning: Grouping Algorithm

The solution we propose in order to achieve load balance is based on the mesh partitioning into groups to be processed in parallel by different processors. In this section the grouping algorithm to subdivide the full mesh into smaller groups is presented. A detailed analysis can be found in 1 and 3. The grouping algorithm we use attempts to reduce the number of boundary triangles, generate well balanced groups, and cover the full mesh in an efficient way.

The group construction algorithm is based on a central vertex search, and the selection of the contiguous concentric triangle strips around this central vertex. Likewise, the central vertex is selected in such a way that the successive processed groups are contiguous and without holes (non-assigned triangles) among groups. Next, we describe in details the grouping algorithm:

1. A list of working starting points, \( L^{\text{start}} \), is maintained. This list, initially empty, keeps a set of vertices starting from which we compute the central vertex for the successive groups to compute.
2. Another list, \( B^{\text{global}} \), with the boundary vertices of the global group (that is, the group resulting from the union of the computed groups), is also maintained and initially empty.
3. We take a random vertex as the central vertex for the construction of the first group (eg. \( v_1 \) in Fig. 2.a), since the \( L^{\text{start}} \) list is initially empty.
4. The group of triangles around the central vertex is then constructed. In Fig. 2 groups formed by three concentric triangle strips are built.
5. Once a group is constructed, the working starting points list, \( L^{\text{start}} \), is updated. This way, the vertices that are boundary vertices in the new group, that were in the previous \( B^{\text{global}} \), and that are characterized by not having all their triangles assigned to a group, are added to the \( L^{\text{start}} \) list. As an example, let us consider the boundary vertices in group B2 of Fig. 2.b, that also were in the previous \( B^{\text{global}} \) (dashed line in Fig. 2.a). In this case the point \( L_2^{\text{start}} \) would be included in the \( L^{\text{start}} \) list. Note that after the first group construction (Fig. 2.a) the previous \( B^{\text{global}} \) is still empty, so that we simply take a random vertex from \( B^{\text{local}} \) (the first one, for instance) as \( L^{\text{start}} \) (the point \( L_1^{\text{start}} \) in Fig. 2.a).
6. After updating the \( L^{\text{start}} \) list, the \( B^{\text{global}} \) list is also updated. Thus, each time a new group is attached to the previous computed groups, the boundary vertices of the group (\( B^{\text{local}} \)) are added to the \( B^{\text{global}} \) list, except the vertices being in the intersection of the two lists (\( B^{\text{local}} \cap B^{\text{global}} \)) and characterized by having all their triangles assigned to a group, which are removed from the resulting \( B^{\text{global}} \) list (these are interior vertices now). This way, for instance, we can see the \( B^{\text{global}} \) list after the construction of the first group with a dashed line in Fig. 2.a and the resulting \( B^{\text{global}} \) list once a second group is computed in Fig. 2.b.
7. The list of working starting points, \( L^{\text{start}} \), is used to compute the central vertex (eg. the point \( v_2 \) in Fig. 2.a and the point \( v_3 \) in Fig. 2.c) for the next group to be built. The basic idea behind our grouping algorithm consists of searching, from a starting vertex \( v \) of \( L^{\text{start}} \), a point that is surrounded by a specific number of triangles strips without overlapping with any group previously computed. The procedure is briefly indicated in (Fig. 3) were groups with three concentric strings were considered. From the starting point \( L_1^{\text{start}} \) the vertices at distance three are identified (Fig. 3.a). From this set of vertices, a set of candi-
dates to be a center point of a new group are identified. Specifically these points should be at distance three from both any vertex of the $B^\text{global}$ list and the border of the mesh. In this example only the point $v_2$ (Fig. 3.b) is a candidate. In case of obtaining more than one candidate vertex, one vertex would be randomly selected.

8. Once a new group have been built and the next central vertex has been chosen, the algorithm goes to the step 4 again, while non-assigned triangles remain in the mesh.

Once a group is built some additional steps have to be performed to assure a good coverage of the mesh. This way, we should take into account the presence of any of the following kinds of groups of triangles:

- **Case 1:** Triangles not included in the group with all their vertices located on the corresponding boundary vertices list (case 1 of Fig. 2.b). The group under construction, $B_2$, is drawn in dark gray and the triangle that is not included in the group but whose vertices are all in the boundary of the group is drawn with a different pattern.

- **Case 2:** Holes, that is, non-assigned triangles, that can also appear in the construction of a group. This can be observed in the case 2 of Fig. 2.b where there are some non-assigned triangles between the last built group ($B_2$, marked in dark grey) and the global group consisting of the previously built groups (only one in this case, $B_1$, and marked in light grey).

- **Case 3:** A strip of cave triangles can appear in the union between two groups. If these triangles are assigned to another new group, no neighbor information could be employed for their subdivision. This situation can be observed in the case 3 of Fig. 2.b.

Every patch in any of these three cases should be detected and included in the current group under construction.

4. Parallel Surface Subdivision Algorithm

In this section the parallel surface subdivision algorithm is presented. Two parallel implementations have been developed, both based on a coarse-grain approach, that is, each processor performs the whole computation of triangles subdivision for disjoint set of groups.

Fig. 4 shows the parallel mesh partitioning scheme for a simple system with two processors. The original coarse mesh is replicated in every processor (first row in Fig. 4); this fact does not represent a drawback due to the low storage requirements for coarse meshes. Next, each processor partitions the full mesh into groups employing the new grouping algorithm (second row in Fig. 4). Groups are assigned cyclically to the different processors in order to obtain load balancing, alternating the order of assignment in each wave for a better work balance. Once the groups are assigned to the processors (third row in Fig. 4) the subdivision procedure is carried out. This way, every processor just takes care of its assigned groups.

4.1. Static Distribution

In our first parallel implementation of the surface subdivision algorithm we have carried out a strict static assignment of the groups to the processors. The groups are sorted in decreasing order of number of triangles per group. We assume
groups. Specifically, the utilization of a mask, which only takes into account the local information of each edge, should be used in order to avoid incoherent information among different neighbor vertices. In order to avoid cracking effects, afterwards, during the recursive process each processor only signed groups.

Perhaps the masks used in each processor would imply different new vertex coordinates have to be assured in both groups. As a result, the number of groups assigned to each processor is minimized. Of course, during this scheduling stage, nonblocking communications (both send and receive primitives) are used to overlap communication and computation.

Let us consider the example of Fig. 5 in order to clarify the scheduling protocol. This figure shows four processors, being 4 different groups of the mesh assigned to each one of them (the list of assigned groups indicates in grey color the processed groups and in black the group being just processed). Every processor maintains a list with the processors with available unprocessed groups. That way, once processors 0 and 2 have finished the computations associated with their assigned groups, they search for the next processor with unprocessed groups. Processor 0 tries to get a group from processor 1 and processor 2 tries to get a group from processor 3. Then, Processor 1 and 3 send the groups 15 and 13, respectively. Additionally, as processor 1 has just finished with its unprocessed groups it sends this information to the rest of processors as well.

### 4.2. Dynamic Load Balancing

In addition to the previous static parallel version we have developed an enhanced implementation that includes a dynamic scheduling. The static implementation could achieve a good load balance in most cases, nevertheless, if any of the processors employed for the computation has a heavy workload, the global performance could be weakened. Thus, in order to avoid busy-waiting processors (while one or more processors are still processing their groups) we have added a dynamic scheduling to our parallelization. Static assignment is at this point complemented with a dynamic load balancing, which is going to allow idle processors to subdivide triangles of groups initially assigned to other processors.

Next, we describe in detail the dynamic scheduling we have developed to solve the load imbalance: First the groups are cyclically assigned to the processors, in the same way as in the static implementation. However, once a processor finishes with its corresponding computations the next step will be to check on unprocessed groups initially assigned to different processors. Then, each processor must keep updated the information about the groups that have not been still processed. In order to reduce the number of communications to a minimum, the processors do not send each other any message during the computation of the subdivision of their assigned groups. A processor only sends a message to every one of the others when it finishes the computation of its assigned groups, so each processor keeps a list of the processors with unprocessed groups.

In order to obtain a new group from another processor, the processor sends a request to the next processor (according to the processor’s identification number) with unprocessed groups. If the processor that receives the requesting message has some unprocessed group then the ownership of that group is transferred to the requesting processor (ACK).

If the group has been already given to any other processor, the group request is refused (NACK). Using this scheduling multiple assignment of one group to two or more processors is avoided. Of course, during this scheduling stage, nonblocking communications (both send and receive primitives) are used to overlap communication and computation.

3. In Table 1 we summarize the times measured on both machines -Origin 2000 and Sunfire 6800- and for the meshes of Fig. 6 after 5 iterations. In column labeled with “Parallel” we display the lower time obtained. This corresponds with the processors employed for the computation has a heavy workload, the global performance could be weakened. Thus, in order to avoid busy-waiting processors (while one or more processors are still processing their groups) we have added a dynamic scheduling to our parallelization. Static assignment is at this point complemented with a dynamic load balancing, which is going to allow idle processors to subdivide triangles of groups initially assigned to other processors.

5. Experimental Results

The Modified Butterfly subdivision algorithm has been implemented on two different machines: a SunFire 6800 cluster with 750MHz UltraSPARC-III processors, and a SGI Origin 2000 distributed-shared memory computer with 250MHz R10000 processors. Both implementations have been performed using the message passing programming model (specifically we have used the MPI library).

Three different models, shown in Fig. 6, have been rendered using the algorithm: *armadillo* (Fig. 6.a), *bunny* (Fig. 6.b), and *hypersheet* (Fig. 6.c). The original coarse meshes are shown in this figure with 799, 499 and 917 triangles, respectively. In Fig. 7 the subdivided meshes after 3 iterations are shown, having 51136, 31936 and 58688 triangles, respectively.

The number of triangles per group for 2, 3 and 4 rings of the three different models are depicted in Fig. 8, respectively: *armadillo* (Fig. 8.a), *bunny* (Fig. 8.b), and *hypersheet* (Fig. 8.c). As we can see, the grouping algorithm covers the full original mesh in an efficient way, producing a balanced number of triangles per group. Obviously, the increasing number of rings produces an increase in the number of triangles per group, and a reduction in the number of groups. The number of simple masks employed on the border and external edges is lower so higher quality are obtained in the resulting images. Anyway, no difference in quality can be appreciated in the final images.

From the results in Table 1 we can appreciate that the number of triangles per group for the *armadillo* model is around 75% lower than the *bunny* model, while the *hypersheet* model has the lowest number of triangles per group. This is due to the fact that the *armadillo* model has fewer vertices than the other two models, and the *hypersheet* model has the most vertices.

In Table 1 we summarize the times measured on both machines -Origin 2000 and Sunfire 6800- and for the meshes of Fig. 6 after 5 iterations. In column labeled with “Parallel” we display the lower time obtained. This corresponds with...
Figure 5: Example of load-balance algorithm (a) Processors 0 and 2 send a request (b) Processors 1 and 3 send the groups 15 and 13 to the requesting processor, respectively.

Table 1: Execution times (seconds)

<table>
<thead>
<tr>
<th>Target Machine</th>
<th>Armadillo</th>
<th>Bunny</th>
<th>Hypersheet</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Seq</td>
<td>Parallel</td>
<td>Seq</td>
</tr>
<tr>
<td>Origin 2000</td>
<td>7.28</td>
<td>0.68(20p)</td>
<td>4.47</td>
</tr>
<tr>
<td>Sunfire 6800</td>
<td>7.26</td>
<td>0.42(24p)</td>
<td>4.48</td>
</tr>
</tbody>
</table>

The configuration with the closest maximum number of processors with respect to the number of groups. For example, the number of groups for 2 rings of the bunny mesh is 19 then we have only considered configurations from 1 to 16 processors.

The results in terms of speedups are shown in Fig. 9 for the armadillo for 1 ring after 5 iterations. The results of the rest of the meshes are similar. These data have been measured with respect to the sequential algorithm without grouping, and show the low overhead associated with our method. As can be observed, we are close to the ideal speedup in both systems, but the speedups are better using the dynamic scheduling. Therefore, these results prove that the techniques employed to parallelize the algorithm permit to obtain an excellent performance on distributed memory systems. Besides, we want to emphasize the good behavior of the algorithms using the cluster. This system is a good, cheap alternative to more expensive systems because the ratio performance/cost is higher than the one obtained using classical multiprocessors. On the other hand, we can see that the dynamic algorithm achieves a better performance regarding to the static one on very loaded machines, for example in the Origin 2000 system (Fig. 9.b).

6. Conclusions

In this paper we have described a parallel implementation of the Modified Butterfly scheme for triangular meshes on distributed memory systems using the message passing programming model. Two versions, one static and one dynamic, have been presented, the latter specially useful in cases of running the algorithm on heavy loaded machines, since it carries out a load balancing approach. The original mesh is partitioned into small groups employing a grouping algorithm. These groups are sorted in decreasing order of number of triangles and cyclically distributed among the processors, trying to assign the same number of computations to each processor, that in the static implementation just carries out the subdivision procedure for its assigned groups. In the dynamic implementation also may compute groups initially assigned to other groups. So load balance and good speedups have been achieved through this approach, as we have experimentally shown on a SunFire 6800 cluster and a SGI Origin 2000.

The parallel method maintains a high quality in the resulting images, in spite of the greatest number of border edges and the simplification introduced by the grouping process. Furthermore, a good speedup was achieved, so we are close to the objective of real time rendering.

© The Eurographics Association 2002.

118
Figure 6: The original coarse meshes. (a) armadillo with 799 triangles, (b) bunny with 499 triangles and (c) hypersheet with 917 triangles

Figure 7: Subdivided meshes from Fig. 6 after 3 iterations. (a) armadillo with 51136 triangles, (b) bunny with 31936 triangles and (c) hypersheet with 58688 triangles

Acknowledgments
This work was partially developed during the stay of E. J. Padrón in the Edinburgh Parallel Computing Centre, Univ. of Edinburgh (EPCC), Scotland (stay supported by Training and Research on Advanced Computing Systems (TRACS)). We gratefully thank Centro de Supercomputación Complutense (Madrid, Spain) for providing access to the SGI Origin 2000. This work was supported in part by the Ministry of Science and Technology (MCYT) of Spain under the project TIC 2001-3694-C02. We acknowledge University of A Coruña (Spain) for the research funds that have also contributed to support this work.

References
Figure 8: Number of triangles per group for 2, 3 and 4 rings, (a) armadillo, (b) bunny and (c) hypersheet


© The Eurographics Association 2002.
Figure 9: Speedup for the armadillo model for the two different machines: (a) SunFire cluster and (b) Origin 2000