Implementation of a well vectorized Marching Cube Algorithm in a Parallel Machine

Prof. Dr. André Luiz Battaiola [1]

Dr. William Hibbard [2]

DC / UFSCar [1]
Rod. Washington Luiz, km 235
São Carlos - SP - 13565-905
andre@dc.ufscar.br

SSEC/University of Wisconsin [2]
1225 W. Dayton St
Madison, WI 53706
whibbard@macc.wisc

Abstract: This paper describes the results of the implementation of a well vectorized version of the marching cube algorithm, used to generate three-dimensional isosurfaces in scientific visualization applications, in a massive parallel machine called CM-5 (Connection Machine). The implementation was made in the message passing and data parallel programming methods.

Key words: Scientific Visualization, Parallel Processing, Isosurfaces

1- Introduction

Isosurfaces are an important tool in many scientific visualization systems. In order to increase interactivity, a fast version of marching cube algorithm [LOREN87] to run in computers with vector facilities was developed.

In an experiment with a virtual reality environment, a parallel machine was used to obtain performance compatible with the interactivity required by the application. The vectorized version of the algorithm was ported to a CM-5 parallel machine. This paper describes the main ideas of the vectorized version of the algorithm and the changes required to port it to the CM-5.

It will be used in this text the following convention:

Dgrid: is the data grid supplied by the user (data grid)
Dsubgrid: is a subset of this data grid (data subgrid)

This convention will avoid confusions with the term *subgrid* used to indicate a subset of data mapped in the CM-5 nodes with vector facilities. In the same way, we use the term *node* or *processor* to mean a CM-5 processor. This will avoid confusions with *vector processor*, the resource that some nodes have and that is responsible for vector processing.

2- The algorithm

The marching cube algorithm interpolates a surface through a 3D grid (Fig. 1) defined by all points at a specified value called an *isovalue* (Fig. 2).

The marching cube algorithm analyzes each grid cube and marks each cube vertex with a flag. The flag is 1 if the vertex has associated a value bigger than the isovalue, and the flag is 0 if the value is equal or lower than the isovalue (Fig. 2). Ordering the cube vertices, it is possible to associate a byte to each cube configuration (Fig. 3).

As the cube has eight vertices, there are 256 configurations. Using rotations of the cube, it is possible to reduce the number of configurations to 30 basic configurations (Fig. 4). Considering negative configurations as equivalent, the number can be reduced to 15 basic configurations. A negative configuration occurs when for each vertex of a cube A, the same vertex of a cube B has a negative logical value. In this implementation we kept the distinction between positive and negative configurations to avoid problems with the calculation of the normals of the polygons.

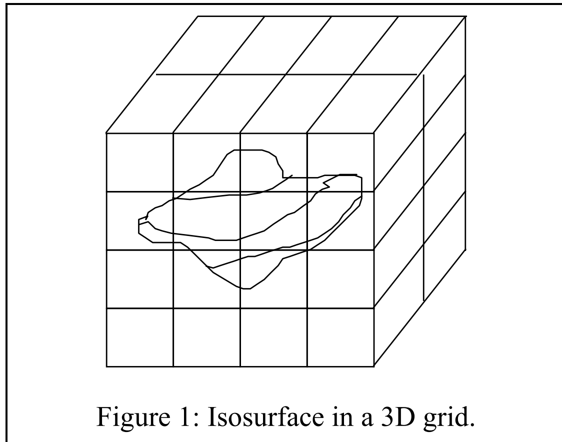

Figure 1: Isosurface in a 3D grid.

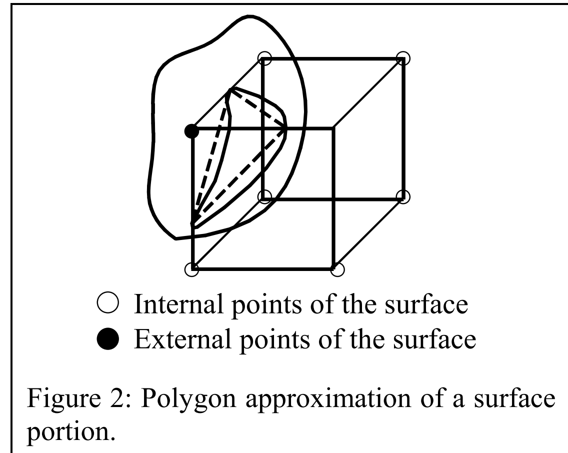

Figure 2: Polygon approximation of a surface portion.

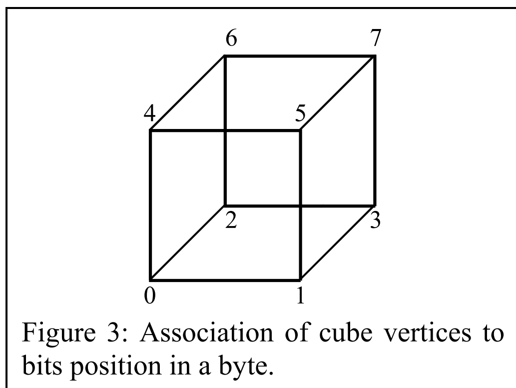

Figure 3: Association of cube vertices to bits position in a byte.

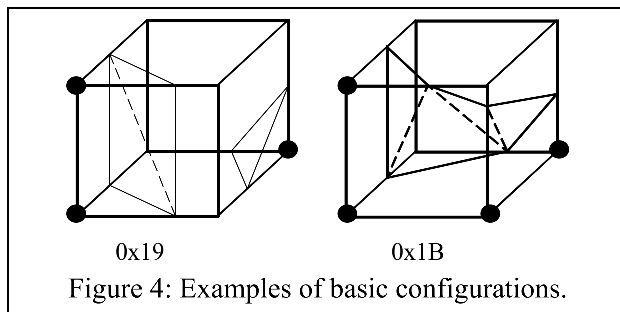

Figure 4: Examples of basic configurations.

To optimize flag processing a scheme of auxiliar planes is used. During the processing of the cubes in the bottom of the data grid, the flags calculated for the right faces of the cubes is stored in a area called pYZ_1, meaning plane YZ. In the next step, the configuration of the cubes in the next column is calculated using pYZ_1 as pYZ_0 (Fig. 5). During the processing of the cubes in the same plane XY, the flags associated with vertices in the top of the cubes are stored in a plane pXY_1. In the next step, the plane pXY_1 is changed to pXY_0 to be used in the processing of the cubes above (Fig. 6).

The process of flag calculation treats some problems related to ambiguities of the marching cube algorithm. When cubes with specific configurations are joined, hole in the surface can occur due the way the polygons are connected (Fig. 7). The flags calculation process recognizes these configurations and changes the type of polygon of one cube to avoid the hole. For example, in the case of Figure 7, the two triangles of the cube below are changed to a hexagon.

In each cube, the polygons of the surface are calculated based on the flag values. After that, a normals calculation process associates a normal to each polygon vertex. To calculate the normals a vector product method is used rather than a gradient method for better appearence with rough surfaces.

The normal of a vertex is calculated as a weighted average of the normals of the triangles that share it. The weight is in a inverse proportion to the the area of each triangle (Fig. 8). This approach gives a smooth appearance to the corners of the rendered surface.

In the end, a process is executed that connect all the polygons and generates a polytriangle strip. The process uses polygon masks that allows the conversion of the polygons to triangles and their connection in a efficient manner (Fig. 9).

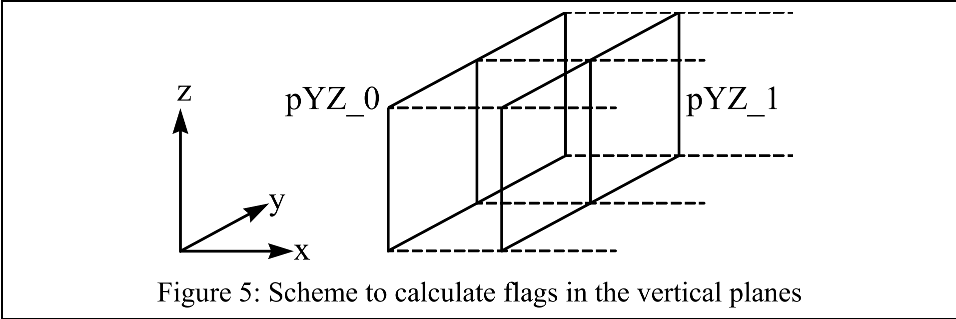

Figure 5: Scheme to calculate flags in the vertical planes

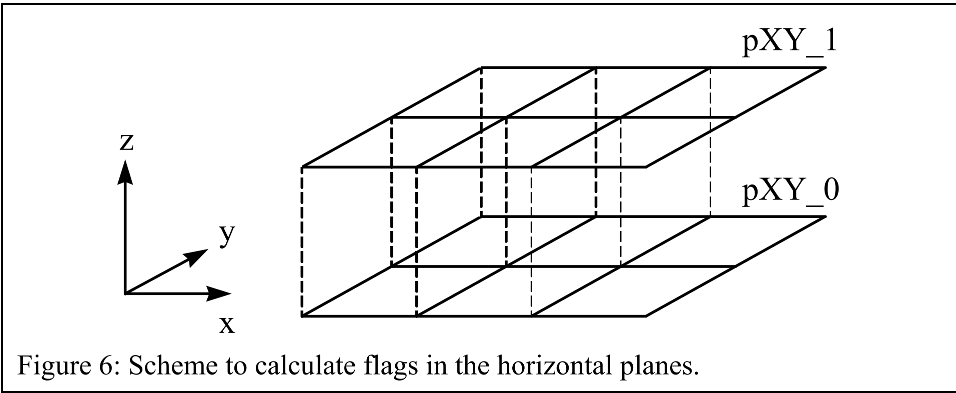

Figure 6: Scheme to calculate flags in the horizontal planes.

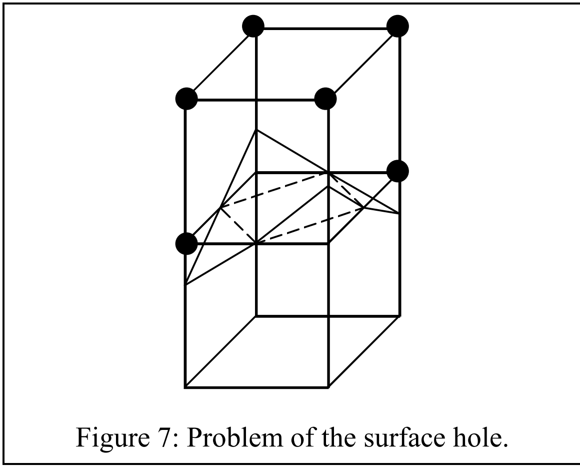
Figure 7: Problem of the surface hole.

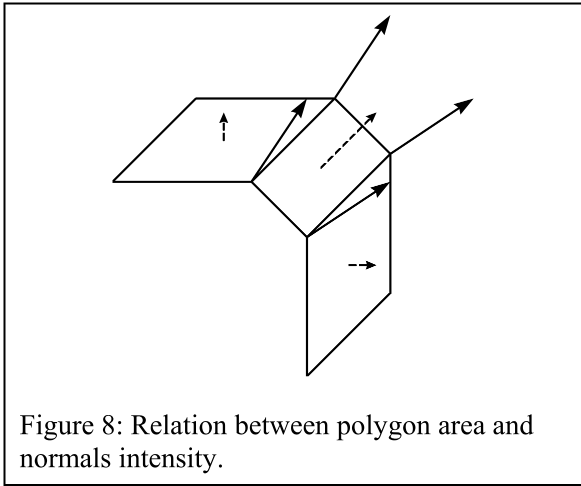
Figure 8: Relation between polygon area and normals intensity.

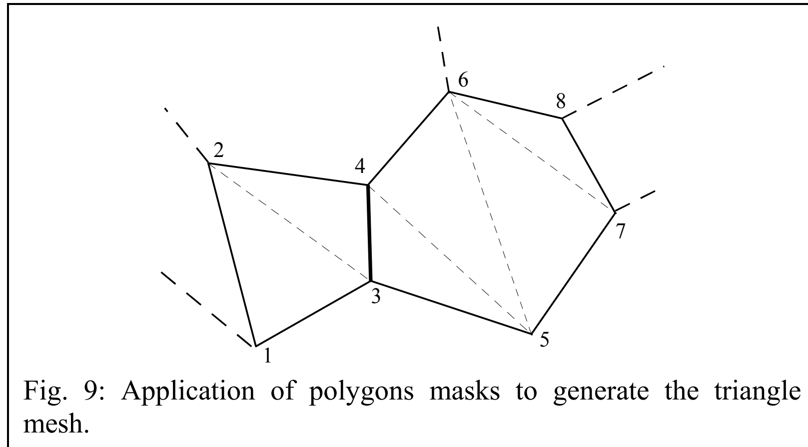

Fig. 9: Application of polygons masks to generate the triangle mesh.

All the algorithm process can be summarized in the C code described below. The names inside the comments will be used as references to each procedure in the rest of the text.

```
Marching_Cubes()
{       Read_Configurations_Table();
        Calculate_Cubes_Configurations();          /* Flags */
        Calculate_Vertice_of_Cubes_Polygons();     /* Marching */
        Calculate_Normals();                       /* Normals */
        Generates_PolyTriangleStrip();             /* Triangles */
}
```

Besides the techniques to optimize the algorithm, each procedure of the algorithm was optimized to run in vector machines. The good results of this implementation in terms of performance encouraged us to port it to a parallel machine.

3. Algorithm Parallelization Methodology

As mentioned before, the idea was not to deeply modify the algorithm to port it to a parallel machine. The idea is to divide the 3D data grid (Fig. 1) between all the processors of the machine.

The complexity of data division for parallel processing is proportional to the dependencies among data, in order to assure data consistency between processors. Fortunately, in the case of this marching cube implementation, the only dependency between adjacent grid cubes occurs in the normals calculation process.

As described before, the normal of a vertex is calculated considering a weighted average of the polygons normals that share it and a poor grid data division might cause vertices to lose connections with polygons (Fig. 9). The solution is to keep an extra layer of border grid points during data division (Fig. 10).

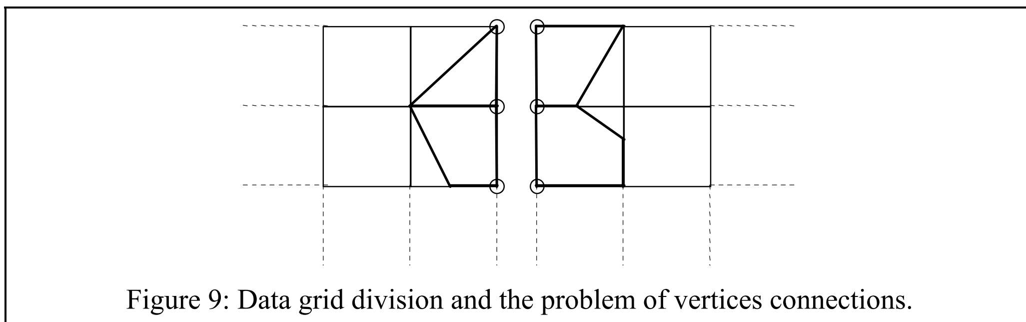

Figure 9: Data grid division and the problem of vertices connections.

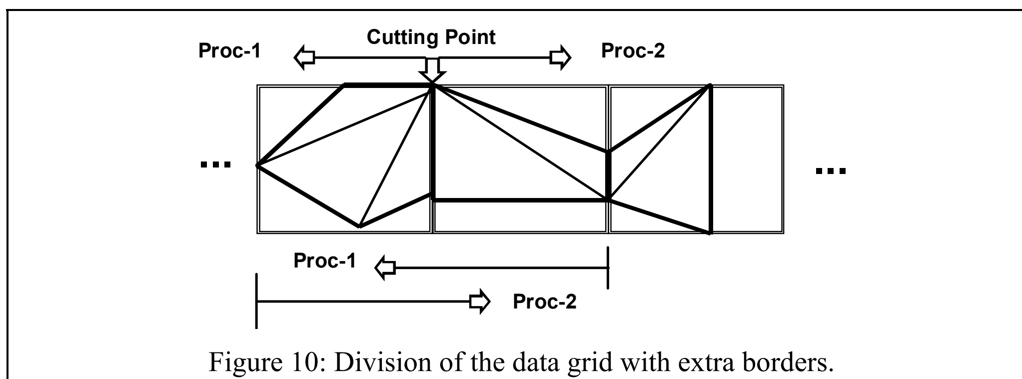

Figure 10: Division of the data grid with extra borders.

4. The machine and the programming methods

The *Connection Machine* [HILL93] uses massive parallelization and it was produced by the *Thinking Machine Company*. This machine has two models: CM-2 and CM-5. The CM-5, the last model, differs of the CM-2 in the ability of the nodes to run instructions in a independent way and in the organization of the input/output mechanism. Also, the CM-5 uses more powerful 64 bit processors.

This kind of machine is structured to allow the execution of a program in a massively parallel environment. As opposed to a conventional vector supercomputer that uses a few powerful and expensive processors, the CM works with a great number of cheap processors (the RISC Sparc) with local memory. The machine architecture does not impose a theoretical limit for the number of processors, but it was projected to support until 16000 processors and the largest machine built has 1024 processors.

The CM allows the creation of processor partitions. A partition is a set of processors controlled by a processor called *the controller*. The CM-5 at the University of Wisconsin, per example, has two partitions, each with 32 processors. One of these partitions has nodes with vector facilities.

There are two basic models to program in a CM: the MP (message passing) and the DP (data parallel). In the CM guides, these methods are referenced as MIMD (Multiple Instruction, Multiple Data) and SIMD (Single Instruction, Multiple Data), respectively.

For this application, MP programming is a simple job as it is only necessary to define how to divide the data among the nodes and how to execute the same copy of a program in each node. Each node processes a subset of the original data. The controller runs another program that controls the execution of the node programs. After data processing in each node, the results are sent to the controller. At this time, the controller can merge the output data. Another interesting option is to send the results directly to the network. This allows the execution of the node programs in an asynchronous manner but requires the receiving machine to merge output data from the nodes.

MP supports the programming languages C, Fortran and Pascal. The MP programs only call special communication and synchronization routines. Depending of the application, the node program can be the original sequential program with a few modification related to the calling of these communication and synchronization routines.

In the case of the marching cube implementation, the main problems of this method were related to the control of the node synchronization and the definition of the best way to divide the data among the nodes and to merge the output data. The synchronization was, in general, only necessary for communications and it was simple to insert this operation in the program. The data division and output merging were also simple and used a scheme similar to the data parallel implementation.

The data parallel implementation requires the use of a special programming language called C* (“C star”) that contains resources to allow the parallelization of the program in a semi-transparent way to the user. This means that the programmer does not need to worry about the communication and

synchronization control between different nodes, but it is required to rewrite the program to use special mechanisms of the C* language. Another advantage of the C* is the access to the vector facilities of the nodes. The languages used in the MP method does not allow the use of vector facilities.

5. The Data Parallel Implementation

The *for* loop described below was written in C* and it gives an idea of the implementation process in this language.

```
shape    [2048]shapeA;
shapeA  int:ptAUX_SP[1000];
shapeA  float:ptGRID_SP[1000]

for (ii = 0; ii < max_grid_dimen; ii++)
{     where (ptGRID_SP[ii] >= INVALID_VALUE)   ptAUX_SP[ii] = 0x1001;
      else where (ptGRID_SP[ii] >= isovalue)    ptAUX_SP[ii] = 1;
      else                                      ptAUX_SP[ii] = 0;
}
```

Considering a 32 nodes configuration, *shapeA* is a data type that will distribute 2048 components in this way: [0-63] in the node 0, [64-127] in the node 1, and sequentially following this scheme until the components [1984-2047] are mapped in the node 31.

ptAUX_SP and *ptGRID_SP* are respectively an integer and a real vector. Each element of these vectors (ptAUX_SP[i] or ptGRID_SP[i]) has 2048 components. In this case we use the term *element* rather than component in order to avoid confusions between vector elements and shape components.

The *where* command works similarly to the *if* command in a sequential program. It defines a selection condition for a shape component as it is exemplified below.

```
ii = n,  0 <= n <= 1000,        0 <= j <= 2047
   if      [j]ptGRID[ii] equal to INVALID_VALUE then     [j]ptAUX_SP[ii] = 0x1001
   else if [j]ptGRID[ii] equal to isovalue then          [j]ptAUX_SP[ii] = 1
   else                                                  [j]ptAUX_SP[ii] = 0
```

Program writing using *where* is not a difficult job, we encountered a serious performance problem using *where* statements. Despite the importance of the *where* command, Thinking Machines informed us that this problem would be solved only in the next version of operating system and unfortunately the available guide of the language did not have this information. We only discovered the reason for the poor performance when another guide became available [CM92]. The reasons for the low performance of the *where* command are explained in this guide and reproduced below.

a) C* blocks of code are loaded in each node of the machine and all nodes are executing the same code synchronously;
b) at the end of the processing of each block of code, all nodes are synchronized so that the results of the block can be used for the next block to be loaded;
c) when a *where* command is found by the used compiler, a new block is generated because the compiler can not distinguish situations that requires or not a new block;
d) so, if a program has a great number of *where* commands it will have a great number of blocks and consequentially a great number of synchronization operations that will decrease the performance of the program.

To solve this problem, the new guide recommended the use of logical variables or expressions rather than *where* commands. Using this approach, the loop described previously changes to:

```
for (ii = 0; ii < max_grid_dimen; ii++)
        ptAUX_SP(ii) = 0  +  (ptGRID_SP[ii] >= isovalue)  +
```

(ptGRID_SP[ii] >= INVALID_VALUE)*0x1000

In this case, the logical expression *ptGRID_SP[ii] >= isovalue* works like a logical parallel variable. Considering *temp_SP* the name of this variable, then:

[j]temp_SP **will be** "1" **if** "[j]ptGRID[ii] >= isovalue"
[j]temp_SP **will be** "0" **if** "[j]ptGRID[ii] < isovalue"

The second loop version is more efficient than the first one. Optimizing all the program using this method it was possible to increase the performance of the algorithm routines in the following rates: Flags 5.78, Marching 1.41, Normals 3.25 and Triangles 1.91. The routines Flags and Normals, with larger percentages of vectorized loops, yielded larger rates of optimization.

Another important factor for efficiency is the size of the subgrid shape that is mapped to each node or vector processor. When the machine has vector facilities, each node is equipped with 8 vector processors. A minimum of 8 components of the shape type is mapped in each vector processor. Then, in a configuration of 32 nodes, the mapping is made in the following way:

shape [2048]shapeA
2048 64 components/node in a 32 nodes machine configuration
64 components of type ShapeA mapped in each node
8 vector processors per node
8 components per vector processor

Table 1		
Routine	**CM-5** (seconds)	**SG Onyx** (seconds)
Flags	0.17	0.21
Marching	1.34	0.17
Normals	0.46	0.19
Triangles	1.30	0.07

The C* compiler always maps a number of *shape* type components that is a multiple of 8 to each vector processor. Then, if a *shape* type is defined with 2050 components, 4096 components will be mapped in the vector processors. The extra components do not cause errors in the output but, as the mapping of the components changes, do have an impact on the program performance. This is especially true in more complex cases such as shape types with two dimensions (example: shape [8][12]shapeA).

Unfortunately, even after the conclusion of all mentioned optimizations, the global performance was still unsatisfactory. Table 1 has a comparison between the time spent to run the routines on a CM-5 with 32 processors and vector facilities, and on a Silicon Graphics Incorporated (SGI) Onyx. These runs used a data grid with dimension 50x50x50 with a spherical isosurface.

The two implementations have a few differences in terms of algorithm process. The main difference is the calculation of normals and the elimination of polygons in the extra border in the CM-5 process. As the SGI Onyx does not have a vector processor, the processing time of the highly vectorized routine Flags is worse than the CM-5 time. On the other hand, the other routines run much faster on the SGI Onyx.

Some reasons for the low performance of the CM-5 implementation are:

a) The CM-5 C* compiler had many problems associated with code optimization, so any implementation on this machine requires many tests to check the performance of code sections. These problems were detected only in the end of the implementation.
b) The CM-5 implementation of the algorithm was based in the assumption that “each component of a shape type processed in a different node would be related to a different Dsubgrid”. For example, in the case of a 32 node configuration, a shape type would have 32 components and consequentially it would have 32 Dsubgrids. But if the machine has vector processors, the mapping of the components in the processors must obey the rule of multiples of 8 and then the number of Dsubgrids became 2048. In the case of a machine with 256 nodes configuration, there are 16384 component for the *shape* type and consequentially 16384 Dsubgrids. Supposing a Dgrid with 50x50x50 dimension, the size of the

Dsubgrid will be a maximum of a 4x4x6 considering the extra borders. Then if a machine configuration has a great number of nodes with vector processors there is a great number of small Dsubgrids with extra borders and this fact increases overhead. Considering also the time spent on node synchronization, it is not possible to decrease the time processing of the program in proportion to the number of nodes.

6. The Message Passing Implementation

The MP implementation used the same division of Dgrids into Dsubgrids that the DP implementation used. An advantage of the MP method is the possibility to use standard languages like C so that, in the case of the marching cube algorithm, few changes had to be done. The only big change was adding a new routine to divide the Dgrid in Dsubgrids plus a few small changes in the routines to calculate the normals and the polytriangle strip.

The MP implementation ran fast in the nodes, but global performance decreased due to the data exchange time between the controller and the nodes. To minimize this problem, we defined a scheme to divide numerous Dgrids into Dsubgrids in the nodes as a pre-process before the user begins interaction. Then, only the selection of a specific Dgrid and a isovalue need to be passed to the nodes during user interaction. An outline of program execution is described below. The text in normal and italic fonts indicate what happens in the controller-program (cp) and in the node-program (np), respectively.

cp Dgrids are divided in Dsubgrids
cp each node receives a set of Dsubgrids, each Dsubgrid related to a specific Dgrid
np nodes receive Dsubgrids

cp ** begin loop **
cp controller sends the insovalue and Dsubgrid index
np receives the isovalue and Dsubgrid index
np executes the algorithm for the specified isovalue and Dsubgrid
np sends the results to the controller
cp controller receives the results
cp ** end loop **

cp if necessary, it is re-executed the loop for other isovalue and Dsubgrid index

The communications between controller and nodes to send the isovalue and the Dsubgrid index, and to receive the results have a relevant impact in the algorithm performance (Tables 2, 3 e 4).

The program performance drops significantly when we consider communications during program execution, so the maximum number of surfaces generated per second are around 6 (Tables 2 e 3). This program performance is not satisfactory.

The communication problem required a precise analyze. Table 4 presents time measures of the total program and of some of its portions, all measured in different executions (that is, Tm_Total was measured in one execution, Tm_March in another and so on). Then:

$$Tm_Total \sim= Tm_Receb + Tm_March + Tm_Trans$$

The Table 4 column with Tm_Rec parameters (time that a node takes to receive two numbers from the controller) shows high values of time and that they are decreasing in a constant proportion.

Table 2					
Nodes Number	Dgrid Dimen. (Surf: Sphere)	Dsubgrid Dimension	Tm_March (seconds)	Ncubes/Sec	NP*NC/Sec (millions)
32	50x50x50	15x15x27	0.20	26000	0.83
32	100x100x100	28x27x52	0.63	57000	1.82
64	100x100x100	28x27x28	0.40	47385	3.03
256	100x100x100	15x15x27	0.16	31850	8.15
64	200x200x200	53x52x53	1.92	71825	4.60
256	200x200x200	28x28x52	0.68	54675	14.40

Tm_March: Time to execute the marching cube algorithm in a specific node
NCubes/Sec: Number of cubes processed per second
NP*NC/Sec: Total number of cubes processed per second considering all nodes

Table 3						
Nodes. Numb.	Dgrid Dimens. (Surf: Sphere)	Tm_RMT (seconds)	NC/Sec	NP*NC/Tm_RMT (millions/sec)	SPS_A	SPS_B
32	50x50x50	0.46	9200	0.30	6.6	2.40
32	100x100x100	2.21	15294	0.49	1.8	0.50
64	100x100x100	2.04	8600	0.47	3.0	0.50
256	100x100x100	2.56	1990	0.51	8.1	0.50
64	200x200x200	22.23	5960	0.38	0.6	0.02
256	200x200x200	24.86	1440	0.37	1.8	0.02

Tm_RMT: Time that a specific node takes to receive the isovalue and the Dsubgrid index plus the time to execute the marching cube algorithm plus the time to send the results
SPS_A: Surfaces per second, ((NP*NC/Tm_March)/Grid), considering only the processing time of the algorithm
SPS_B: Surfaces per second, ((NP*NC/Tm_RMT)/Grid), considering the processing time of the algorithm plus communication time between controller and nodes

As described below, the controller sends the grid_id (Dsubgrid index) and the isovalue to the nodes using the function *CMMD_bc_from_host()* that broadcasts a specific parameter to all nodes.

```
CMMD_bc_from_host (&isovalue, sizeof(float));          /* Transmission 1 */
CMMD_bc_from_host (&current_grid_id, sizeof(int));    /* Transmission 2 */
```

To identify the effect of each of these transmissions on computing time, we measured the time spent for each one (Table 5). Then, the analyses in Table 5 of Tm_Rec, Tm_Rec_1 and Tm_Rec_2 shows that the first nodes require around 1 second to receive the two values and that basically all the time is spent in the first parameter reception.

The same time measures were obtained when we inverted the parameters transmission (that is, the Dsubgrid index is transmitted first). Then it is reasonable to suppose that there is some problem related to sequential synchronization in the processors (Table 6).

To confirm that the time problem was really related to synchronization operations, we measured the node reception time with a user-forced synchronization before the communication process. Thus we used the functions *CMMD_sync_host_with_nodes()* and *CMMD_sync_with_host()*, as shown in the code fragment described below. Using these functions there is a guaranty that all the nodes are synchronized

before the communications and we can assume that the communications times measured do not reflect syncronization operations.

```
/* Fragment code of the controller program */
CMMD_sync_host_with_nodes();
CMMD_bc_from_host (&isovalue, sizeof(float));
CMMD_bc_from_host (&current_grid_id,sizeof(int));
/* Fragment code of the node program */
CMMD_sync_with_host();
CMMD_receive_bc_from_host (&f_aux, sizeof(float));
CMMD_receive_bc_from_host (&cur_grid_id,sizeof(int));
```

Table 4

Node	Dsubgrid (Sphere)	NumVert	Tm_Total (seconds)	Tm_March (seconds)	Tm_Rec (seconds)	Tm_Transm (seconds)
0	27x27x27	312	1.781	0.186	1.527	0.006
1	27x27x27	1115	2.455	0.371	1.504	0.459
2	27x27x27	1126	2.403	0.364	1.481	0.438
3	27x27x28	390	1.808	0.267	1.457	0.055
...	...	...	...	...	...	...
60	28X28X27	478	0.478	0.254	0.082	0.076
61	28X28X27	1315	1.157	0.416	0.058	0.565
62	28X28X27	1320	1.070	0.324	0.030	0.526
63	28x28x28	576	0.395	0.287	0.000	0.103

Dsubgrid: Dsubgrid dimension (surface Sphere, Dgrid dimension 100x100x100, 64 nodes)
NumVert: Number of polygons vertices in the Dsubgrid
Tm_Total Tm_March plus Tm_Rec plus Tm_Transm
Tm_March: Marching cube execution time in a "i" node
Tm_Rec: Time that a node "i" takes to receive the isovalue and the Dsubgrid index
Tm_Trans: Time that a node "i" takes to transmit the results to the controller

Table – 5

Node	Tm_Rec (seconds)	Tm_Rec_1 (seconds)	Tm_Rec_2 (seconds)
0	1.527	1.518	0.000027
1	1.504	1.495	0.000024
2	1.481	1.472	0.000022
3	1.457	1.447	0.000021
...	...	...	...
60	0.082	0.076	0.000021
61	0.058	0.051	0.000019
62	0.030	0.026	0.000023
63	0.000	0.000	0.000025

Tm_Rec: Time that the node "i" spent to receive the isovalue and the Dsubgrid index. It was considered a Dgrid with dimension 100x100x100, a sphere surface and a configuration with 64 nodes.
Tm_Rec_1: Time that the node "i" takes to receive the first parameter, in the case, the isovalue
Tm_Rec_2: Time that the node "i" takes to receive the first parameter, in the case, the Dsubgrid index

Table 7 presents the times measured for the communications with explicit synchronization. Comparing the values in table 5, 6 and 7, we observe that there is no difference in the communication times measured.

In fact, the same time measures were obtained when the two parameters were sent from the controller to the nodes using the following loops:

```
/* Loop in the controller program - Option 1 */
for (node = 0; node < NUM_PROC; node++)
{       CMMD_send_block (node, grid_vl_TAG,&isovalue,sizeof(float));
        CMMD_send_block (node, grid_id_TAG,&current_grid_id, sizeof(int));
}
/* Loop in the controller program - Option 2 */
for (node = NUM_PROC-1; node >= 0; node--)
{       CMMD_send_block (node, grid_vl_TAG,&isovalue,sizeof(float));
        CMMD_send_block (node, grid_id_TAG,&current_grid_id, sizeof(int));
}
```

Considering a sequential communication process, it should be expected that in the option 1 loop the node 0 would take less time in communication than the node with the highest index. But in any case node 0 presents the highest communication time.

Table – 6			
Nodes	Tm_Rec_N(i) (seconds)	Tm_Rec_N(i+1) (seconds)	Tm_Dif (seconds)
0 , 1	1.518	1.495	0.023
1 , 2	1.495	1.472	0.023
2 , 3	1.472	1.447	0.025
3 , 4	1.447	1.424	0.023
...	...	...	...
60 , 61	0.076	0.051	0.025
61 , 62	0.051	0.026	0.025
62 , 63	0.026	0.000	0.026

Nodes:

Node (i) and Node (i+1) (configuration with 64 nodes)

Tm_Rec_node(x):

Time that node "i" takes to receive the first parameter from the controller

Tm_Dif:

Tm_Rec_node(i) - Tp_Rec_no(i+1)

Table 7		
Node	Tm_Sinc_Rec (seconds)	Tm_Rec_P1 (seconds)
0	1.533	0.000027
1	1.510	0.000024
2	1.484	0.000022
3 ...	1.460 ...	0.000021 ...
61	0.059	0.000019
62	0.031	0.000023
63	0.005	0.000025

Tm_Sinc_Rec: Time that node "i" requires to synchronize and receive the two parameters

Tm_Rec_P1: Time that node "i" takes to receive the first parameter from the controller

The analysis of all these experiments points to the following conclusions:

a) The nodes work like a multi-tasking processor that execute portions of programs in conformity to a priority schedule.
b) Program portions that the machine runs in different stages require synchronization with other nodes.
c) The machine allows the configuration of the node numbering and thus varies the node with the highest index. Probably, for some reason related to this fact, the machine initiate the synchronization following a sequence that starts in the node with highest index and finish in the node with index 0. Consequently the machine spent k seconds to allocate node n, k+0.024 to allocate node n-1, k+0.048 to allocate node n-2 and so on. Thus, in a machine with a 64 node configuration, the time to allocate the node 0 is around 1.5 seconds, in accordance with the measurements presented in Table 7.
d) Probably this problem can be avoided if the machine can be used in an exclusive execution mode. In this case the syncronization process occurs only when it is required by the program, and the impact of synchronization can be decreased significantly.

A final conclusion of this analyze is that the CM-5 has an efficient communication process between controller and nodes, but the machine requires node and controller synchronization that decreases communications efficiency.

An option to increase program performance is to transmit results from the nodes directly to workstations where visualization occurs. Unfortunately it was not possible to check this option.

The CPU time required by the program to generate a sphere in a Dgrid with dimension 100x100x100 with 32, 64, 128 and 256 nodes were aproximatelly the same, 50sec. The values have a variation of 40% because the elapsed time routine had precision problems during controller time measuring.

7. Conclusions

Besides the machine problems detected, other reasons for the low performance of the CM-5 version of the marching cube are:

a) The super estimation of the CM-5 performance caused a basic error in all this work. Due to the fact that the user guides did not declare the problems associated with the *where* statement and with synchronization, any implementation in this machine requires a deep study of all its capabilities and definition of a specific algorithm for its environment.
   This fact is more relevant when it is considered the performance reached by a CM-2 implementation of the marching cube algorithm [HANSE92]. Despite the fact that CM-2 is slower than CM-5 and that CM-2 version of the algorithm does not treat problems like surface holes and more precise normals, the rate of 170.000 polygons processed per second is still significant.
b) During the time of this implementation only the reported related work was available, so it was not possible to have good clues pointing to the best direction to follow.

An important conclusion of all this work was that any algorithm parallelization requires a deep study of the target machine resources. Based in this study and in a good analyze of the algorithm working process, a specific scheme of parallelization is defined.

8. References

[CM92] *CM-5 C* Performance Guide* - version 7.1

[HANSE92] C. D. Hansen, P. Hinker, *Massively Parallel Isosurface Extraction*, Computer Graphics (1992), 77-82

[HILLI93] W. D. Hillis, L. W. Tucker*, The CM-5 Connection Machine: A Scaleable Supercomputer*, Communications of the ACM (Nov., 1993), v.36, n.11, 31-40

[LOREN87] W. E. Lorensen, H. E. Cline, "*Marching Cubes: A High Resolution 3D Surface Construction Algorithm*", Computer Graphics (July, 1987), v.21, n.4.